A Compact Petri Net Representation for Ada Tasking Programs

Matthew B. Dwyer ⋆
Kari A. Nies †
Lori A. Clarke ⋆

CMPSCI Technical Report 94–16
Feb. 7, 1994

⋆ *Software Development Laboratory*
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

† Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92717

Submitted to International Symposium on Software Testing and Analysis

---

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under grant MDA972-91-J-1009. The content does not necessarily reflect the position or policy of the U.S. Government and no official endorsement should be inferred.

A Compact Petri Net Representation for Ada Tasking Programs [1]

Matthew B. Dwyer ⋆
Kari A. Nies†
Lori A. Clarke ⋆

⋆Department of Computer Science
University of Massachusetts, Amherst
†Dept. of Information & Computer Science
University of California, Irvine
email: dwyer@cs.umass.edu

Abstract

This paper presents a compact Petri net representation for concurrent programs with explicit tasking and rendezvous style communication that is efficient to construct. These Petri nets are based on task interaction graphs and are called TIG-based Petri nets (TPN)s. They form a compact representation by abstracting large regions of program execution and summarizing the information from those regions that is necessary for performing program analysis. We present a flexible framework for checking a variety of properties of concurrent programs using the reachability graph generated from a TPN. We also discuss the applicability of state space reduction techniques to TPNs. We present experimental results that demonstrate the benefit of TPNs over alternate Petri net representations.

[1]This work was supported by the Defense Advanced Research Projects Agency under Grant MDA972-91-J-1009 and the Office of Naval Research under Grant N00014-90-J-1791.

1 Introduction

An important goal of software engineering research is to provide cost-effective techniques that allow developers of concurrent software to gain confidence in the quality of their programs. Towards this end many researchers [16, 17, 14, 7, 3] have investigated state reachability analyses. Theoretical results on the complexity of basic questions about concurrent software [15] imply that reachability analysis can be prohibitively expensive. Recent experimental results [17, 12, 2] suggest that reachability analysis that considers the entire concurrent program is practical only for small to medium size programs of moderate complexity. One approach to extending the applicability of reachability analysis, called *state space reduction*, is to [3, 12] create a more compact program representation that preserves only the information necessary to analyze a restricted class of properties.

In this paper, we describe a Petri net representation for concurrent programs, with explicit tasking and rendezvous style communication, called *TIG-based Petri nets* (TPN)s. This representation summarizes large regions of program execution and makes available information from those regions that is important for program analysis. The result is a representation that is compact and unlike reduction techniques, has no loss of information. This representation appears to be amenable to reachability analysis for larger programs than previously proposed Petri net reachability techniques. TPNs are also amenable to property specific state space reduction techniques.

Given the large amount of information required to reason about properties of concurrent programs, there is a fundamental choice about where this information is encoded. It can be explicitly represented in the program representation or in the analysis algorithms that operate on the representation. Choosing the former increases the space requirements of analysis but simplifies the algorithms; the later decreases space requirements but increases algorithm complexity. The major limiting factor in performing reachability analysis is the construction of the reachability graph. Our hypothesis is that using a program representation that reduces the size of the reachability graph at the expense of increased cost in analysis of reachable program states, will allow practical analysis of programs for which reachability analysis is otherwise impractical. To evaluate this hypothesis we have constructed a set of tools to gather data on TPNs and reachability graphs generated from TPNs.

In the following section we give a brief overview of Petri nets and TIGs. Section 3 shows how a TPN is constructed and discusses the semantic content of TPN places and transitions. Section 4 describes analysis of state reachability properties using TPNs. We discuss how reachability analysis of TPNs differs from reachability of most other Petri net representations. We also describe how reductions can be applied to TPNs. In section 5 we describe a prototype toolset that we have built to evaluate the TPN representation. We present experimental data on the size of the reachability graphs generated from TPNs and on the cost of checking the graph for desired properties. Section 6 mentions directions for future work.

2 Overview

This section defines general Petri net and TIG terminology and introduces a simple example to illustrate the concepts presented in the paper. We assume that the concurrent programs being represented are Ada tasking programs. In principle the representations and algorithms described are applicable to programs written in any procedural programming that supports explicit tasking and rendezvous style communication.

Petri Nets

Petri nets are a formalism used for modeling concurrent systems[9]. A Petri net is a directed bipartite graph with nodes called *places* and *transitions*. Typically, places are drawn as circles and transitions as bars. The edges of the graph are called *arcs*. Arcs are labeled with a positive integer representing their *weight*. A *marking* is an assignment of an integer to each place in the net and represents the number of *tokens* at that place. Tokens are drawn as black dots inside of places. A marking is given by a k-vector, M, where k is the number of places in the net and $M(i)$ denotes the number of tokens at place i. Formally, a Petri net is a tuple (P,T,F,W,M_0), where P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, W is a function assigning weights to arcs, and M_0 is the initial net marking. In this paper all of the Petri nets discussed are *ordinary*, having arc weights of 1, and *safe*, having a maximum of 1 token per place. Associated with each transition is a set of *input places*, places at the head of incoming arcs, and *output places*, places at the tail of outgoing arcs. 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 *fires* by removing tokens from each input place and adding tokens to each output place. A transition may not fire unless it is enabled.

In [7, 11, 5] Petri nets have been used to model and analyze concurrent programs that use rendezvous style communication. The reachability graph of such a Petri net has been used to perform analysis of Ada tasking programs [11, 7].

Task Interaction Graphs

TIGs have been proposed by Long and Clarke [4] as a compact representation for tasks. TIGs divide tasks into maximal sequential regions, where such task regions define all of the possible behaviors between two consecutive task interactions. The TIG abstraction hides control flow information internal to a task, leading to a smaller graph than a traditional control flow graph. A task interaction is any point where the behavior of one task can be influenced by the behavior of another task. To support efficient analysis, task regions summarize groups of exiting task interactions for which the same control flow path through the region is taken into *edge groups*[2]. Task interactions are *blocking* if their execution is unavoidable given a particular control flow path through the task region. An example of *non-blocking* interactions are accept statements in a select statement with an else branch.

Formally, a TIG is a tuple (N, E, S, T, L, C), where N is the set of nodes representing task regions, E is the set of edges representing task interactions, S is the start node, T is the set of terminal nodes, L is a function assigning labels to edges, and C is a function assigning code fragments to nodes. The start node represents the region where task execution begins and the terminal nodes represent regions where task execution potentially ends. Each node has a fragment of code associated with it that represents Ada statements in the task region plus two non-executable statements, ENTER and EXIT, that mark region entry and exit points. The ENTER and EXIT statements take a description of the task interaction as an argument and ENTER takes a second argument describing the successor TIG node. The edges of a TIG are labeled with the tasking interactions that cause transitions from one region to another. The tasking interactions we consider are Ada entry calls and accept statements. There are four distinct kinds of tasking interactions: starting an entry call,

[2]The situation is somewhat more complicated as not all control flow choices are relevant. Only the branch choices made at conditional statements on which entry calls and accept statements are control-dependent are necessary for accurate analysis.

```
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 1: 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 2: Code fragments for task T1 of example

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 accept bodies are themselves task regions that perform sequential computation that must be captured in the representation.

Figure 1 presents a simple Ada program that will be used as an example throughout the rest of the paper. Task T1 consists of 5 sequential regions. To illustrate the idea of maximal sequential regions, consider the initial region of T1. 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. The code fragments for task T1 are given in figure 2. The TIGs for tasks T1 and T2 are given in figure 3, where an entry call is denoted by the task and entry name and an accept is denoted by the entry name alone. Given that regions represent all sequential execution paths between pairs of consecutive tasking

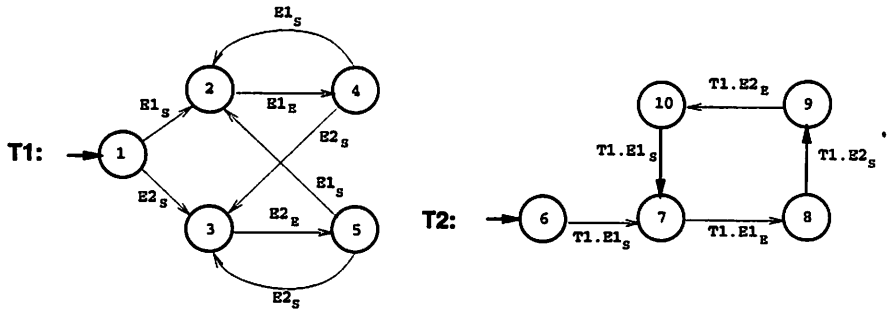

Figure 3: TIGs for example

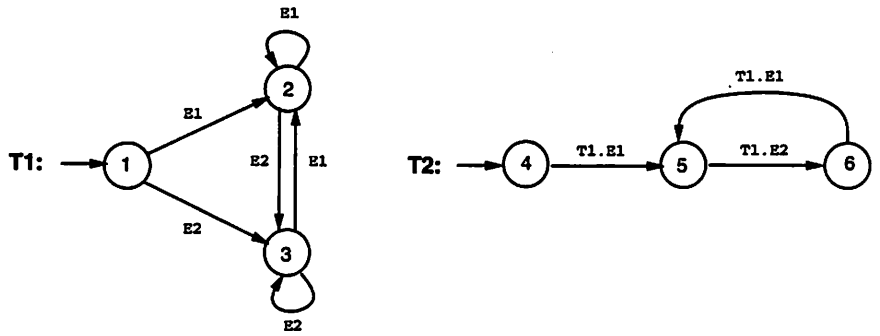

Figure 4: Reduced TIGs for example

interactions, it is possible for distinct TIG nodes to contain the same program statements. When EXIT statements are duplicated in this way, multiple TIG edges may be used to represent a single Ada communication statement in the source program. This is illustrated by the duplication of the statement `EXIT(ACCEPT_START(E1),2)` in regions 1, 4 and 5 of task T1 in figure 3. Note that a TIG represents a single task instance. The potential behaviors of a collection of tasks can be modeled by matching edges from different TIGs, whose labels represent calls and accepts of the same task entry.

If the accept statement of a rendezvous has no accept body then we can reduce the size of the TIG representation without loss of information. A single interaction, comprising both start and end of a rendezvous, is used to model such an accept statement and any entry calls made on it. Since the accept statements given in task T1 of figure 1 have no accept bodies, the TIGs for tasks T1 and T2 can be reduced as shown in figure 4.

3 TIG-based Petri nets

We propose a Petri net model for Ada tasking programs that is constructed from a set of TIGs and therefore hides the details of task control flow. A TPN maintains a strong relationship with the set of TIGs; each place in the Petri net has a one-to-one correspondence with a tasking region and each transition represents a potential task interaction. TPNs can be constructed quite simply by creating a transition for each pair of TIG edges whose labels represent a call and accept on the same task entry and by making the source and destination regions of these edges the input and output places of the new transition. More precisely:

```
Input:  a set of TIGs
Output:  (P,T,F,W,M0) a Petri-net
Algorithm:
   W = 1
   for each node, n, in the set of input TIGs
      create a place and add it to P
   end for
   for each edge, c, in the set of input TIGs that represents an entry call
      for each edge, a, in the set of input TIGs that represents an accept
         if L(c) is a call to task entry L(a) then
            create a transition, t, and add it to T
            add the following arcs to F:
               (place(head(c)), t)
               (place(head(a)), t)
               (t, place(tail(c)))
               (t, place(tail(a)))
         end if
      end for
   end for
   M0 = mark each place corresponding to a TIG start region with 1 token
```

The head and tail functions access the source and destination of a TIG edge, L(e) is the label for TIG edge e, and place returns the TPN place associated with a given TIG region. The resulting Petri nets are ordinary, by definition, and safe, as M_0 has only values of 0 or 1, and for all transitions t, |input places of t| = |output places of t| = 2, so the number of tokens is preserved across transitions. We note that Pezzè, Taylor and Young [10] independently developed a similar algorithm for constructing Petri nets from labeled flow graphs.

This algorithm constructs a Petri net that overestimates the possible task interactions of the program. All potential task interactions are included as a result of the simple and efficient matching of TIG edge labels, but some of these interactions can never be executed. From the algorithm description it is clear that the cost of constructing a TPN from a set of TIGs is dependent on the product of the number of calling and accepting TIG edges. For all of the examples discussed in section 5 this product is less than 10000 and as a result the TPN construction is quite fast in practice.

As we can see from the above algorithm the total number of places in a TPN is the sum of the number of nodes of the TIGs representing the program. A single Ada task may have a number of entry calls and accept statements, these may be either a synchronizing rendezvous, where an accept has no body, or a remote procedure call (RPC), where an accept has a body. Consider a task with c_{synch} synchronizing entry calls, c_{rpc} RPC entry calls, a_{synch} synchronizing accept statements and a_{rpc} RPC accept statements. The TIG representing such a task has $\leq 2(c_{rpc}) + 2(a_{rpc}) + c_{synch} + a_{synch} + 1$ nodes. There is a TIG region for every task interaction contained in the modeled task, except for the interaction that represents task termination. An RPC rendezvous models start and end interactions separately and contributes two regions to the TIG. A synchronizing rendezvous models start and end interactions as a single interaction and contributes a single region to the TIG. The interaction modeling task activation contributes an additional region to the TIG. We note

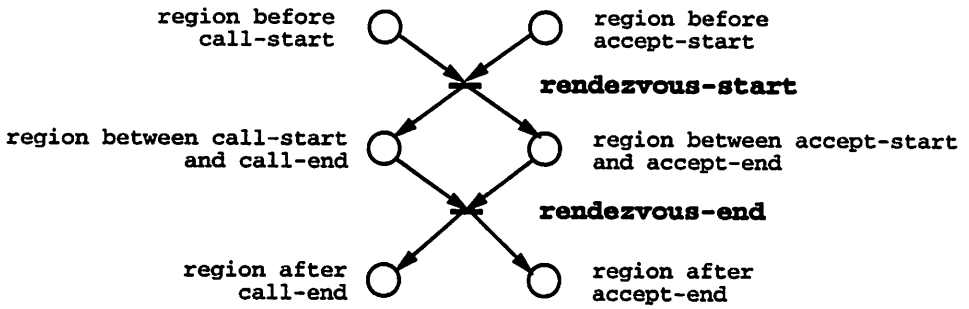

Figure 5: TIG-based Petri net representation for Ada rendezvous

that this is a strong upper bound. Furthermore this bound demonstrates that the number of TPN places is linear in the number of communication statements in the program.

In this algorithm, a TPN transition is created for each syntactic matching of edge labels. Ideally this would correspond to a syntactic matching of each pair of call and accept for the same task entry in the source code. However, there may be multiple TIG nodes that represent task regions on separate branches of a control flow statement. At the point where control flow merges, each of these TIG nodes must have an edge to the common successor TIG node. The potential for having multiple TIG edges corresponding to a single call or accept statement in the source program results in extra TPN transitions. There are pathologic examples where the number of TPN transitions used to represent communication through a given task entry is quadratic in the number of call and accept statements of that entry in the program. We note that many of these transitions are unreachable, and hence do not contribute to the complexity of TPN based reachability analysis.

Figure 5 illustrates the Petri net fragment that represents a single Ada rendezvous between a calling and accepting task. In the case of multiple callers this fragment is replicated with the accepting task participating in all potential rendezvous.

Continuing with our example, figure 6 illustrates the TPN constructed from the reduced TIGs in figure 4 where the executable transitions and arcs are in bold. This example illustrates a number of the benefits of the TPN representation. Each task communication has a simple representation as a single TPN transition that has a calling and accepting input place and two output places. There is a single marked place in the set of places associated with each task in the program that keeps track of individual task states. We have found that this regular structure of TPNs simplifies reasoning about the correctness of the TPN representation and TPN based analysis. TPNs are smaller than some other Petri net representations, for the above example the Ada-net, as defined by Shatz et. al. [11], would have 21 places and 16 transitions as compared to 6 places and 9 transitions for the TPN in figure 6. As we will see in section 5, in practice TPNs and the reachability graphs generated from them are smaller than Ada-nets and Ada-net generated reachability graphs.

4 Analysis using TPNs

Petri nets have been used for simulation, visualization, modeling and analysis [8]. We are primarily interested in analysis, so the appropriate measure of worth of the TPN representation is its suitability for analysis. TPNs are atypical of Petri net representations in that they implicitly represent control flow merge and branch points within a TPN place. In contrast most Petri net representations [7, 5, 11, 14] explicitly represent control flow merge and branch points using special purpose

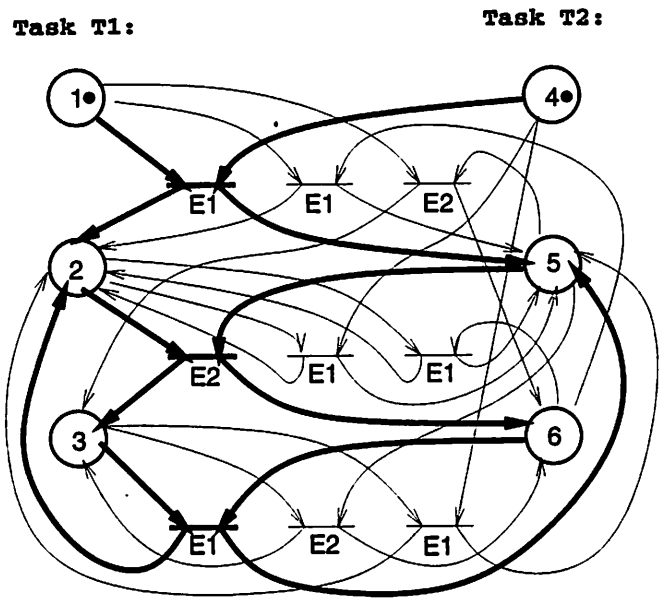

Figure 6: TIG-based Petri net for example

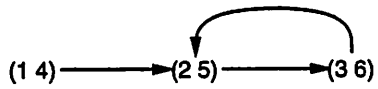

Figure 7: Reachability graph from Reduced TIGs and TPN for example

places and transitions. While the complexity of some structural Petri net analyses may depend on the size of the net [8], it is not a strong factor in determining the cost of reachibility analysis. Empirical data presented in section 5 bears this out. To judge the effectiveness of a representation for reachability analysis we need to consider both the size of the generated reachability graph and the cost of checking properties on that graph. In the remainder of this section we discuss these issues and how TPNs can be reduced prior to and during reachability analysis in the spirit of [12] and [3].

4.1 Reachability

Young et. al. [17] discuss separating the construction of the reachability graph from the process of checking a particular property. Experimental evidence suggests that construction of the reachability graph is the limiting factor in performing state reachability analyses. If we can construct the reachability graph it is often practical to check the property of interest by traversing the graph. We adopt this separation of generation and checking in our analysis.

The TPN reachability graph is generated using standard Petri net techniques [7]. The structure of TPNs is such that the number of marked places in any TPN marking is equal to the number of tasks in the program. So, TPN markings can be represented as an array of elements of length equal to the number of tasks in the program rather than as a bit vector of length equal to the number of Petri net places, which is needed for general ordinary, safe nets. The reachability graph for the TPN in figure 6 is given in figure 7, where nodes represent TPN markings.

Checking whether a program exhibits a desired property involves defining a *property predicate* that decides whether a TPN marking satisfies the property in question. The predicates must be defined to be conservative in the sense that they never return false when a marking corresponds to a state in which the desired property is true. To illustrate we discuss checking whether a marking corresponds to a program deadlock state and whether a marking corresponds to a violation of a region of exclusive access.

Freedom from deadlock is checked by determining whether some combination of the individual task states represented by the marked TPN places corresponds to a program deadlock. Conceptually we need to look at all of the possible control flow choices that can be made in the task regions associated with the marked TPN places. If we find a set of control flow choices such that no pair of tasks can successfully communicate, then the current TPN marking corresponds to a deadlock. To improve the efficiency of analysis TPN places summarize, through their associated TIG regions, the control flow choices that need to be considered to determine deadlock markings. Members of non-blocking exiting edge groups from a TIG region can never contribute to a program deadlock as they can always be bypassed. Thus, we restrict our attention to blocking edge groups. A TPN marking corresponds to a potential program deadlock if there is some combination of outgoing edge groups from the TIG regions of the marking, such that no pair of edges from these edge groups are matching communications [3]. We formulate this condition as a deadlock property predicate. For each exiting, blocking edge group of each TPN place we compute a pair of bit-vectors of length equal to the number of task entries in the program. A value of 1 in the ith bit of the `accept-vector` indicates that the TPN place has an outgoing edge that accepts the ith task entry. A value of 1 in the jth bit of the `call-vector` indicates that the TPN place has an outgoing edge that calls the jth task entry. We use bit-wise or, $\bigcup^{bit}$, and bit-wise and, $\bigcap^{bit}$, operations over collections of bit vectors in the following algorithm.

```
Input:  TPN marking M = [n_1, n_2, ..., n_k]
Output:  True if the marking may correspond to a potential program deadlock
Algorithm:
   for each combination, C, of blocking, exiting edge groups from M
      all-accepts := ⋃^bit_{p∈C} accept-vector of p
      all-calls := ⋃^bit_{p∈C} call-vector of p
      if (all-accepts ⋂^bit all-calls) = (0, ..., 0) then
         return TRUE
      end if
   end for
   return FALSE
```

To illustrate, consider the deadlock property applied to the reachable TPN marking $(2, 5)$ in figure 7. For this example, see figure 6, TPN place 2 has a single exiting, blocking edge group whose value is $\{Accept(E1), Accept(E2)\}$ with an `accept-vector` of $(1, 1)$ and an `call-vector` of $(0, 0)$ and TPN place 5 has a single exiting, blocking edge group whose value is $\{Call(T1.E2)\}$ with an `accept-vector` of $(0, 0)$ and an `call-vector` of $(0, 1)$. The single edge group combination

[3] A degenerate case of this condition is that terminal reachable TPN markings, i.e., markings that have no successor, correspond to potential program deadlocks, since the set of edges in any edge group combination is empty there can be no pair of edges in it that correspond to matching communications.

considered by the deadlock predicate for TPN marking $(2,5)$ computes the bit vector expression $(((1,1) \vee (0,0)) \wedge ((0,0) \vee (0,1))) = (0,1)$ so the predicate returns FALSE.

This algorithm is exponential in the size of the TPN marking, which is always equal to the number of tasks in the program, where the base of the exponent is related to the number of exiting blocking edge groups. Young et. al. [17] have shown that this problem is NP-hard, but they have found through experimentation that for many programs checking this condition is practical. In contrast to TPNs, Petri nets that explicitly represent control flow can use a test for terminal reachable markings as a conservative property predicate for checking deadlock.

The need for exclusive access to selected regions of program execution is common in concurrent programs. We develop a flexible property predicate for checking violation of a mutual exclusion properties. Collections of tasks within a program may arbitrate exclusive access to regions of execution within each task by using semaphore tasks, or similar constructs. We call such a collection an *arbitration group*. A program may consist of many arbitration groups and an individual task may be a member of multiple arbitration groups. Ideally a region of mutual exclusion is a represented by a TPN place with a single input transition, representing acquisition of the semaphore, and a single output transition, representing release of the semaphore. When there is a program error, such as an omitted acquire or release operations, the region in the source code may span a number of TPN places. If there are m arbitration groups in the program we can associate with each TPN place a bit vector, called `in-region`, of length m. A 1 in the ith bit of this vector indicates that the TPN place is part of or contains a task region associated with the ith arbitration group. A bit value of 0 indicates that the TPN place is not in a region of mutual exclusion. We use bit-wise and, $\bigcap^{bit}$, over a set of bit vectors in the following algorithm.

```
Input:  TPN marking M = [n_1, n_2, ..., n_k]
Output: True if the marking may correspond to a violation of exclusive
        access to a region associated with some arbitration group.
Algorithm:
   result := ⋂^bit_{i∈1...k} in-region of M[i]
   return (result = (0, ..., 0))
```

The result computed by this algorithm provides sufficient information to point the user at regions of source code that may be in violation of mutual exclusion properties. If we make the assumption that the number of arbitration groups is reasonable, e.g., less than 128, then the bit-vector operations used in the predicate are efficient. Checking this property predicate for a given TPN marking requires time that is linear in the number of tasks in the program. A number of other co-executability properties can be checked using similar property predicates, e.g., critical races. This type of property predicate can also be applied to Petri nets that explicitly represent control flow.

As discussed in section 1, TPN based analysis represents a tradeoff in encoding information in the program representation versus analysis algorithms. In section 5 we will see that TPN reachability graphs are relatively small. The two property predicates described above illustrate that checking properties of a TPN marking can range in cost from linear to exponential in the number of tasks. Checking properties of reachable TPN markings for which efficient predicates are available provides the benefit of TPN reachability graph compaction without increase in the cost of analysis. It remains to be seen whether the additional cost of checking the deadlock predicate is compensated for by TPN reachability graph compaction.

4.2 TPN based Reduction

The goal of reduction techniques is to make reachability graphs smaller. Reductions can be applied prior to or during reachability graph generation.

The theory of Petri net reductions [8] allows a given net to be replaced by a different net where this reduced net maintains certain desired properties of the original net and has a smaller reachability graph. Net reductions typically replace a selected Petri sub-net with a new sub-net. The key to net reductions is the preservation of the desired properties. Reachability analysis of some Petri net representations can test for the existence of terminal markings to conservatively detect program deadlock. A net reduction must preserve this information so that each reachable terminal marking of the original net corresponds to some reachable terminal marking of the reduced net. Recent experimental data has demonstrated that net reduction techniques are an effective approach to extending the size of programs for which deadlock checking is practical[2, 12].

Unfortunately, program deadlocks are not conservatively represented by the set of reachable terminal TPN markings, so we cannot directly apply existing deadlock preserving Petri net reductions. However, net reductions can be developed that are applicable to TPNs. For deadlock preserving reductions the idea is that if we can find a necessary condition for our deadlock property predicate to hold, and test for that condition over parts of the TPN, then when we find the condition is false we know that the part of the TPN we tested cannot participate in a reachable TPN marking that corresponds to program deadlock. We discuss two reductions: *synchronizing accepts* and *forced communication pair*. The first is formulated as a net reduction and the second is formulated as a reduction that can be applied during reachability graph construction.

The synchronizing accept reduction preserves all information in the reduced net, including program deadlocks. When all accept statements for a given task entry have an empty body we can eliminate all ACCEPT_END and ENTRY_CALL_END transitions that correspond to accepts and entry calls of that task entry. We also merge each output place of those transitions with the input place in the same task. This is equivalent to the TIG reduction mentioned in section 2. This reduction cannot take place until all potential communication partners are known. Since communication partners are explicitly represented in TPNs we have all the necessary information to perform this reduction prior to construction of the reachability graph.

The forced communication pair reduction preserves program deadlocks in the reduced TPN. This reduction makes use of knowledge about the representation of deadlocks in TPNs and the semantics of the deadlock property predicate. If there are a pair of TPN places that each have a single common output transition, this transition must represent the execution of a blocking call and accept statement to the same task entry. The task regions associated with the TPN places each have a single blocking exit edge group and those groups each contain a single edge. Every edge group combination for any TPN marking in which this pair of places is marked must contain the pair of exiting blocking edge groups mentioned above. This pair of edge groups has matching call and accept edges, therefore the deadlock property predicate can never return TRUE for any TPN marking containing such a pair of places. If we collect and record all pairs of TPN markings that constitute a forced communication pair then, during reachability analysis when a marking is encountered for which both places of such a pair are marked, we can skip that reachable marking and generate its successors directly. As with the synchronizing accept reduction the reason we can detect forced communication pairs prior to reachability graph construction is because all possible communication partners are explicit in the TPN. This reduction can be generalized to more than a pair of TPN places. For example, if there is a set of TPN places where all of the output transitions

Example	Tasks	Places TPN	Transitions TPN	States TPN	Arcs TPN	Places Ada-net	Transitions Ada-net	States Ada-net	Arcs Ada-net
DARTES opt	31	528	668	*	*	*	*	*	*
Q	18	245	231	*	*	*	*	*	*
BDS	14	105	131	*	*				
BDS opt	14	93	112	*	*	263	220	-	-
Hartstone	5	46	27	29	29				
Hartstone opt	5	46	27	29	29	*	*	*	*
Gas-1 3	7	53	73	566	897				
Gas-1 3 opt	7	35	62	323	526	157	141	79153	293490
Gas-1 5	9	79	137	11831	24004				
Gas-1 5 opt	9	53	120	6304	13397	313	309	-	-
Phils 3	6	43	36	268	576				
Phils 3 opt	6	25	24	84	186	72	54	18900	79083
Phils 5	10	71	60	11744	42440				
Phils 5 opt	10	41	40	1653	6130	120	90	-	-
Phils 7	14	99	85	-	-				
Phils 7 opt	14	57	56	32063	166502	*	*	*	*
RW 2/1	4	29	56	85	163				
RW 2/1 opt	4	17	48	41	119	93	92	-	-
RW 2/2	5	34	78	383	900				
RW 2/2 opt	5	20	66	175	692	-	-	-	-
RW 2/3	6	39	100	1413	3835				
RW 2/3 opt	6	23	84	609	3031	-	-	-	-
RW 3/2	6	39	95	1339	3644				
RW 3/2 opt	6	23	81	579	2884	138	143	-	-
RW 5/2	8	48	129	15221	50060				
RW 5/2 opt	8	28	111	5811	40660	-	-	-	-
RW 2/5	8	48	144	16433	53775				
RW 2/5 opt	8	28	120	6229	43571	-	-	-	-

Figure 8: Unreduced TPN and Ada-net data

have input places in the given set we can apply similar reasoning as above. This would allow the reachability graph of programs with multiple callers blocking on an entry for which there is a single blocking accept statement to be reduced.

We are currently developing a TPN net reduction for forced communication pairs. Compaction of the reachability graph is accomplished by adding transitions to the net that bypass all markings that contain the communication pair. We have found the regular structure of TPNs to be advantageous in developing and reasoning about the conservativeness of reductions.

5 Experimental Evaluation

We have constructed a set of tools to evaluate the suitability of TPNs for analysis of realistic programs. This toolset is built from components produced by the Arcadia consortium. At present the TPN tools are immature and we are only able to gather data on the size of the generated reachability graphs. The cost of checking property predicates can only be estimated based on information extracted from the TIG representation.

We first present data on the size of unreduced TPNs and the generated reachability graphs for

a number of examples in figure 8 [4]. In this table we use the symbol - to indicate that the tools were unable to build the reachability graph for the example and the symbol * to indicate that no experimental data is available. DARTES and BDS are both large simulation programs [6]. They contain entry calls and accept statements nested within complicated control flow structures. DARTES consists of 31 tasks and BDS consists of 14 tasks, none of the tasks are identical in structure. Q is an instance of a client-server based inter-process communication facility. It consists of 18 tasks and contains complicated control flow structures in which communication statements are nested. Hartstone is a benchmark driver that has a very regular structure [6]. Gas-1 are versions of the one pump gas-station example without deadlock and with the operator task unrolled to accept separate customer entries. Phils are versions of the basic dining philosophers example with deadlock. RW are versions of the readers/writers example presented in [1]. The numbers next to the example names indicate the scale of the problem. For the gas station this is the number of customers, for dining philosophers the number of philosophers and forks, and for readers/writers the number of reader and writer tasks. We indicate with opt that the TPN has been optimized using the synchronizing accept reduction.

We compare TPNs to the Ada-nets generated using the TOTAL system [11]. Ada-nets explicitly encode program control flow in the net and as such represent a contrast to TPNs in terms of the tradeoff between encoding information in the program representation versus the analysis algorithms. TOTAL is one of the few Petri net based systems for analyzing Ada tasking programs for which a mature implementation and experimental data are available. Experiments using the TOTAL system were conducted for BDS, versions of Gas, Phils and the readers/writers examples [2, 12]. The size of reachability graphs generated from unreduced Ada-nets, along with data on the sizes of the unreduced nets in terms of places and transitions [13] are given in figure 8, next to the comparable TPN experimental results. We note that the number of arcs in a TPN is always 4 times the number of transitions, as each transition has 2 input and 2 output places. For many unreduced Ada-nets, the number of arcs is less than 4 times the number of transitions. The two examples for which data is available from reachability analysis of unreduced Ada-nets and TPNs are the 3 customer Gas-1 and 3 dining philosophers. Comparison of this data illustrates the reachability graph compaction that can be gained by using TPNs, as the number of states and arcs in the reachability graphs are two orders of magnitude less for TPN generated graphs. Although the maximum capacity of the TOTAL toolset is not stated, programs whose reachability graphs are as large as 200000 states and 750000 arcs have been analyzed [2]. If we assume that reachability graphs are at least that large for the examples where reachability graphs for unreduced Ada-nets could not be generated, then the results we obtained for those examples show a compaction on the order of two orders of magnitude.

A major limiting factor in performing reachability analysis is the ability to construct the reachability graph and in this respect TPNs are superior to Ada-nets. Comparing TPNs and Ada-nets is fair because they represent equivalent amounts of program information. While early work on analyzing Ada-nets relied on reachability of unreduced nets [14, 11], more recent work [2, 12] has demonstrated that if we are interested in analyzing programs for deadlock freedom, Petri net reduction techniques are capable of significantly extending the size of problems for which reachability analysis can be performed. Comparing analysis based on reduced TPNs and reduced Ada-nets is

[4] Analysis results for Q and BDS have not been presented as there are known bugs in one of the front end tools that affect these examples. These bugs are being fixed and we expect data for these examples to be included in the final revision of this paper.

Example	Task Name	Max Blocking Exiting Edge Groups	Total Entries
BDS (product)		32	18
	BDS	2	
	Graphics_x_Display	2	
	Mouse_Buffer_x_Save	1	
	Mouse_x_Char_In	1	
	Mouse_x_Char_In_GenReports	1	
	Rocket_x_Control_Type	2	
	Rocket_x_Rocket_Guide_1	1	
	Rocket_x_Rocket_Guide_2	1	
	Simulate_x_RDL_x_Guide_Buf	1	
	Simulate_x_RDL_x_Report_Buf	1	
	Simulate_x_RDL_x_Rock_Sup	1	
	Simulate_x_Sensor_x_Targ_Sup	1	
	Status_x_Update	2	
	Target_x_Track	2	
	Target_x_Track_Data	1	

Figure 9: Worst case number of blocking edge combinations for BDS

not possible at present because TPN reductions have not been fully implemented.

In section 4 we describe two property predicates. The deadlock predicate requires that in the worst case we check all combinations of blocking exiting edge groups associated with a TPN marking. If we consider the number of blocking, exiting edge groups for each TIG node, we can compute the maximum number of edge group combinations that the deadlock predicate must consider for any reachable TPN marking. We summarize this data for a number of the examples from figure 8. We also give the total number of task entries in the program, as this determines the practicality of the bit-vector operations used in implementing property predicates. Data for the 14 tasks of the BDS program are given in figure 9. Data for the other examples are given in figure 10. The product indicates the number of combinations of communication statements that would need to be considered for a TPN marking in which each marked place had the maximum number of blocking exiting edge groups. Its clear that the cost of checking the TPN deadlock predicate may vary, sometimes widely, with the program under analysis. The nesting of communication statements within complex control flow statements in the BDS example is clearly the most problematic. We note that these are indications of the worst case cost of checking the deadlock predicate. In fact, for the BDS system, in 3 of the 5 tasks with maximum edge group values of 2 that maximum occurs for a single TIG region. In the other 2 tasks it occurs in less than 3 TIG regions. This is suggestive that the average cost of checking the deadlock predicate over the set of reachable TPN markings for the BDS program is much less than the worst case. Furthermore, the number of task entries for these examples is small enough so that the bit vector operations used in property predicates execute in constant time. In contrast to the BDS example, the fork and philosopher tasks of the Phils examples have no complicated control flow. Because of this, all choice among potential communication events is encoded explicitly in the TPN, so we could just look for terminal reachable TPN markings to conservatively detect deadlock for these examples.

Example	Task Name	Max Blocking Exiting Edge Groups	Total Entries
Hartstone (product)		1	10
	T[1...5]	1	
	Hartstone	1	
Gas-1 w/k Customers (product)		k+2	2k+4
	Operator	k+2	
	Pump	1	
	Customer	1	
Phils w/k Forks and Philosophers (product)		1	2k
	Fork	1	
	Philosopher	1	
RW w/ n Readers and m Writers (product)		1	4
	Reader	1	
	Writer	1	
	Controller	1	

Figure 10: Worst case number of blocking edge combinations

6 Conclusion

In this paper, we have presented a Petri net representation for tasking programs based on task interaction graphs that is efficient to construct. We have developed an upper bound on the number of places of these TIG-based Petri nets. Comparison with existing Petri net representations for Ada tasking programs provides evidence that the number of places in TPNs is relatively small. We have noted that in the worst case the number of TPN transitions corresponding to communication over a given task entry is quadratic in the number of syntactic calls and accepts of that entry in the program. Experimental evidence shows that in spite of this bound, TPNs are typically smaller than Petri nets that explicitly represent program control flow. More importantly for analysis, the reachability graphs generated from TPNs are also smaller, in some cases dramatically smaller.

It has been suggested that no one technique is suitable for analysis of all properties of all concurrent programs. TPNs bring elements of Petri net and TIG based reachability analysis together. They represent a different tradeoff between encoding information in the program representation versus analysis algorithms than has traditionally been made for Petri net representations. This paper has presented preliminary data to investigate our hypothesis that choosing a representation that reduces the size of the state space at the cost of checking properties of program states is more practical than reachability analysis using representations that are more explicit. For a class of properties, that includes checking for violations of mutual exclusion, the benefit of our approach is clear. More experimentation is required to assess whether the tradeoff is beneficial for properties that are more costly to check.

References

[1] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[2] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Using state space methods for deadlock analysis in ada tasking. *Software Engineering Notes*, 18(3):51–60, July 1993. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA93).

[3] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third Workshop on Computer Aided Verification*, pages 417–428, July 1991.

[4] 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.

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

[6] S.P. Masticola. *Static Detection of Deadlocks in Polynomial Time.* PhD thesis, Rutgers University, May 1993.

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

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

[9] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] Mauro Pezzè, Richard N. Taylor, and Michal Young. Graph models for reachability analysis of concurrent programs. Technical Report TR-92-27, Department of Information and Computer Science, University of California, Irvine, January 1992.

[11] 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.

[12] S.M. Shatz, S. Tu, , T.Murata, and S. Duri. 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, 1994.

[13] Sol Shatz. Personal Communication, February 1993.

[14] 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.

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

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

[17] Michal Young, Richard N. Taylor, David L. Levine, Kari Forester, and Debra Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. Technical Report TR-128-P, Software Engineering Research Center, 1398 Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, October 1992.