A Compact Petri Net Representation for Concurrent Program

Matthew B. Dwyer
Lori A. Clarke

CMPSCI Technical Report 95–61
July 1995

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137.

A Compact Petri Net Representation for Concurrent Programs [*]

Matthew B. Dwyer
Lori A. Clarke

Department of Computer Science
University of Massachusetts, Amherst

Abstract

This paper presents a compact Petri net representation for concurrent programs. These Petri nets are based on task interaction graphs and, thus, are called *TIG-based Petri nets* (TPN)s. They form a compact representation by summarizing the effects of large sequential regions of a program and making useful information about those regions available for program analysis. TPNs and their associated analyses represent a tradeoff between encoding information about program behavior in the program representation or in the analysis algorithms. To evaluate the cost-effectiveness of this tradeoff, we have developed a flexible framework for checking a variety of properties of concurrent programs using the reachability graph generated from a TPN. We present empirical results that demonstrate the benefit of TPNs over alternate Petri net representations and discuss techniques to further reduce the cost of TPN-based analysis.

1 Introduction

An important goal of software engineering research is to provide cost-effective analysis techniques that allow developers of concurrent software to gain confidence in the quality of their programs. Researchers have proposed a variety of techniques for analyzing concurrent programs. Although these techniques vary in the time and space required for analysis, in the accuracy of their results, and in the types of questions that each can address, the nature of this variation is not well-understood, particularly when the techniques are applied to real production software systems. Most techniques have theoretical bounds that are daunting, but preliminary experimental results seem to indicate that there are applications for which each might be cost-effective. One of the goals of our work is to understand the practical limitations of a variety of static program analysis techniques as applied to "real" programs.

We have developed a framework for experimenting with a variety of state space enumeration analyses based on *task interaction graphs* (TIG)s [LC89] and Petri nets. Petri nets are a well-studied model for concurrent systems [Mur89]. This paper presents a coarsened Petri net model, called *TIG-based Petri nets* (TPN)s, that is efficient to construct. This model summarizes the effects of large regions of a program and makes useful information about those regions available for program analysis. The result is a representation that is compact, but, unlike many state space reduction techniques, there is no loss of information. The TPN representation appears to be amenable to reachability analysis for larger programs than previously proposed Petri net reachability techniques.

The major limiting factor in performing state space analysis is the enumeration of the reachable program states. Our hypothesis is that using a program model that reduces the size of the state space, even at the expense of increased cost in analysis of reachable program states, will allow analysis of programs for which reachability analysis is otherwise impractical. One of the goals of this research is to evaluate this hypothesis. In support of this we have constructed a set of tools to gather data on TPNs and reachability graphs generated from TPNs. These tools accept Ada programs, which explicitly denote tasks and use a rendezvous style of inter-task communication. We compare our results to recent work on alternate Petri net representations for Ada programs.

In the following section, we describe the nature of the tradeoff between encoding program information in a model versus in an analysis algorithm. Section 3 discusses related work. Section 4 provides a brief description of Petri

[*]This work was supported by the Advanced Research Projects Agency under Grant F30602-94-C-0137.

net and TIG models which, in Section 5, are combined into the TPN model. Section 6 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 present empirical data on the size of the reachability graphs generated from TPNs and on the cost of checking properties over those graphs. In section 7, we describe how TPNs can be reduced prior to reachability analysis. Section 8 summarizes the contributions of this work and concludes.

2 Rationale

All static program analysis techniques gather information about executable program behavior by reasoning using a model of program executions. Ideally, the model captures essential details of program executions that allow an analysis algorithm to distinguish between executions that satisfy or fail to satisfy the property being analyzed. For example, if we want to analyze a concurrent program for the absence of deadlock, the model should provide sufficient information so that we can distinguish those executions that deadlock from those that do not.

In general, representing complete information about the possible executions of a program is infeasible. A program can have an infinite number of executions, for example, if it contains an infinite loop. Thus, a program model only captures partial information about the set of possible program executions. In some cases, even capturing the appropriate partial program information can be impractical.

In designing program analyses, a tradeoff is made between the information encoded in the program model and the computations that must be encoded in the analysis algorithm itself. In general, there are a number of different ways we can choose to make this tradeoff. We can have large information rich models and relatively simple analysis algorithms. Alternatively, we can have very lean models and complex analysis algorithms that derive the necessary program information. The question is: what is the most cost-effective way to make this tradeoff. This question is complicated by the fact that the answer may not be universal; it may depend on the program being analyzed and the kind of analysis being performed.

The idea of shifting information between the model and analysis algorithms is not new. Early compilers used *control flow graphs* (CFG)s whose nodes represented program statements annotated with simple forms of analysis-specific information. The cost of data flow analysis grows rapidly with the number of CFG nodes and analyses over these statement CFGs was expensive. Basic block CFGs [ASU85] were introduced to coarsen the statement-level CFG by collapsing multiple nodes into a single node. This essentially shifted information from the CFG to the analysis algorithms. The algorithms now had to compute and manipulate more complicated analysis-specific node information. Thus, the decision was to reduce graph size at the expense of increasing the cost of processing each individual CFG node. The payoff was a net reduction in total analysis cost, because the additional node cost was more than compensated for by the reduction in the number of nodes processed.

The TPNs we present in Section 5 make a similar tradeoff; information is shifted from the TPN to the analysis algorithms. The benefit of this tradeoff may vary with the program and analysis being considered. Therefore, we demonstrate empirically that this results in a net reduction in analysis cost for a collection of different programs and analyses. In fact, in many cases this tradeoff enables analysis of programs for which analysis of a statement-level Petri net representation is infeasible.

3 Related Work

State space enumeration methods consider each reachable program state to determine whether a program satisfies a given property [MR87, SMBT90, Tay83, YTL$^+$95]. Unfortunately, in general, as programs increase in size and complexity, the state space grows exponentially and the space/time requirements of these analysis methods becomes impractical. The state space considered by these methods can be reduced by maintaining only the parts of the state space that are relevant to the analysis of a particular property, such as deadlock freedom [GW91, DBDS94]. For some programs, state space reduction is able to decrease analysis cost considerably but, in general, the cost of these techniques grows exponentially with the size of the program.

Symbolic model-checking techniques use a fix-point computation over an encoding of the state transition relation to determine reachability of a given state [BCM$^+$90]. For some systems this encoding is very compact, allowing time-efficient analysis. Finding a compact encoding can be difficult, however, and for some systems no compact encoding exists, resulting in a worst case state transition relation that is exponential in size.

Integer linear programming techniques avoid consideration of the state space entirely. They formulate a set of necessary conditions related to the property of interest and analyze the satisfiability of those conditions by the program [ABC$^+$91]. Unfortunately, in the worst-case, the integer programming algorithm for performing this analysis requires exponential time.

Data flow analysis techniques are one of the few concurrency analysis approaches that do not have exponential cost [CK93, DS91, DC94, MR91]. These techniques formulate a set of conditions, related to the property to be analyzed, as a set of data flow problems whose solution provides information about the validity or satisfiability of those conditions by the program. These conditions must be strong, so that the number of spurious analysis results is small. Finding conditions that are strong enough for the analysis problem at hand yet amenable to a polynomial-time data flow formulation can be difficult.

Compositional approaches decompose the original analysis problem into smaller problems on which the above techniques can be applied [YY91]. This approach relies on finding a decomposition of the original problem that significantly reduces the cost of analysis for the subproblems. For many programs, such a suitable decomposition may be difficult to find, if one exists at all.

It has been demonstrated that each of the analysis techniques described above is capable of cost-effectively producing analysis results of sufficient accuracy to verify non-trivial properties of selected concurrent programs. Such results are useful as an initial indication of the feasibility of an analysis technique. The cost of an analysis technique can vary greatly from program to program. The control and communication structures that are used in real concurrent programs[And91] can also vary greatly. Therefore, a thorough understanding of the practical benefits of an analysis technique requires evaluation of that technique over a wide range of real concurrent programs.

To date, there has been little empirical work in evaluating concurrency analysis techniques. Experimental results suggest that despite the rapid growth of the state space, enumeration methods that consider the entire concurrent program can be practical for small to medium size programs of moderate complexity [YTL+95] and that state space reduction techniques can increase the size of the programs that can be considered still further [DBDS94]. A recent study [Cor94] has compared the cost-effectiveness of state space enumeration, reduction, model-checking and integer programming analysis techniques. Although the study considered only a small set of programs, one conclusion was that state space enumeration techniques can be more effective for programs with relatively few tasks, where the tasks contain significant control and data structures. The other techniques excelled when other types of programs were analyzed. Clearly much more work is needed before we understand the relative strengths and weaknesses of each analysis technique and before software developers are able to choose the most appropriate technique for the analysis task at hand.

In this paper, we explore a technique that reduces the size of the program state space considerably; consequently, a larger class of programs can be evaluated using this approach.

4 Background

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

Petri Nets

Definition 1 *A Petri net is a directed bipartite graph that can be written as a tuple* (P, T, F, 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, and* M_0 *is the initial marking.*

A *marking* is an assignment of an integer to each place in the net that represents the number of *tokens* at that place. Tokens and markings are used to record the state of a Petri net. In this paper, all of the Petri nets discussed are *safe*, having a maximum of 1 token per place. 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. 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 a token. An enabled transition *fires* by removing a token from each input place and adding a token to each output place. A transition that is never enabled is called *dead*. A marking of a Petri net is *reachable* if there exists a chain of transition firings that leads from M_0 to the marking.

Figure 1 presents a simple Ada program that will be used as an example throughout the rest of the paper. Task T1 of this example uses a *select-else* statement to poll for the presence of callers on entry a and, if none are present, blocks waiting for a caller on entry b.

Petri net models of concurrent programs have existed for some time; they are usually constructed from the set of control flow graphs for the tasks of the program [MZGT85, MR87, PTY92, SMBT90]. We call a Petri net that explicitly represents the possible control flow branch and merge points in each program task a *control flow graph Petri net* (CFGPN). Figure 2 illustrates a typical CFGPN for the example, where rendezvous start and end are

```
task body T1 is                    task body T2 is
begin                              begin
    loop                               loop
        select                             T1.a;
            accept a;                      T1.b;
        else                           end loop;
            accept b;              end T2;
        end select;
    end loop;
end T1;
```

Figure 1: Ada tasking example

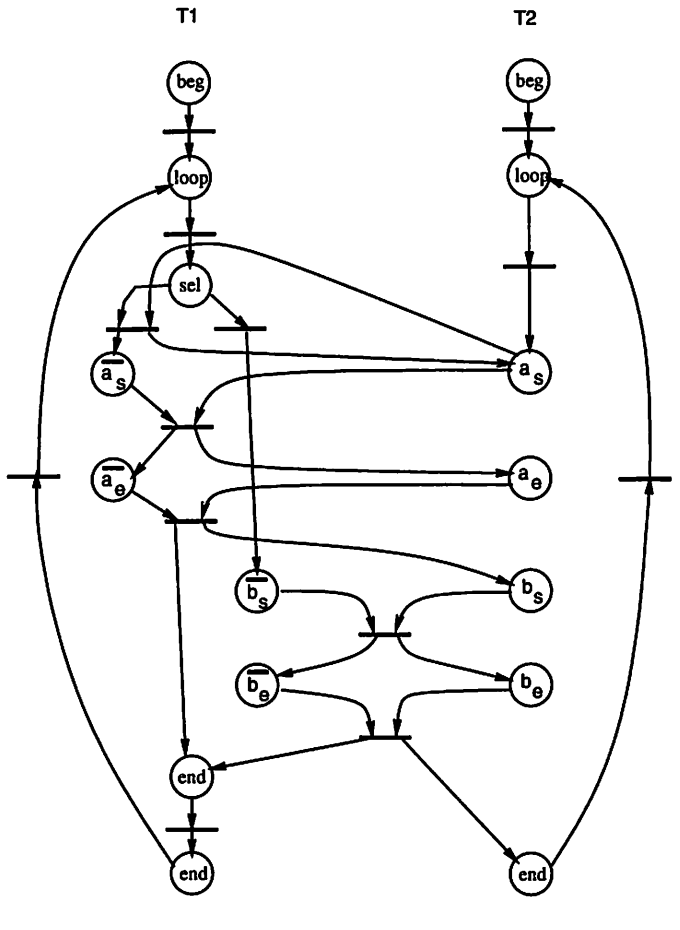

Figure 2: Control Flow Graph Petri Net for example

represented by separate transitions. Places are depicted as circles and transitions are depicted as bars. We denote start(end) of an entry call by the name of the entry subscripted by s(e), e.g., a_s. We denote start(end) of an accept statement by putting a bar over the name of the entry subscripted by s(e), e.g., $\overline{a_s}$. Note that in the example, the output transition of the *sel* place leading to the $\overline{a_s}$ place requires a token from the calling place in task T2; however the output transition leading to the $\overline{b_s}$ place requires no such token as it represents a control flow choice that is internal to T1. Because the net represents control flow choices explicitly, the set of reachable markings that have no successor marking is a conservative approximation of the set of program deadlock states. Reachability graphs for this type of Petri net have been used to perform analysis of Ada tasking programs [MR87, SMBT90].

```
C(1) = task body T1 is
       begin
         loop
           select
             EXIT(a̅_s,2);
           else
             EXIT(b̅_s,3);
```

```
C(2) = ENTER(a̅_s);
       EXIT(a̅_e,4);
```

```
C(3) = ENTER(b̅_s);
       EXIT(b̅_e,5);
```

```
C(4) = loop
         select
           EXIT(a̅_s,2);
           ENTER(a̅_e);
         else
           EXIT(b̅_s,3);
         end select;
       end loop;
```

```
C(5) = loop
         select
           EXIT(a̅_s,2);
         else
           EXIT(b̅_s,3);
           ENTER(b̅_e);
         end select;
       end loop;
```

Figure 3: Code fragments for task T1 of example

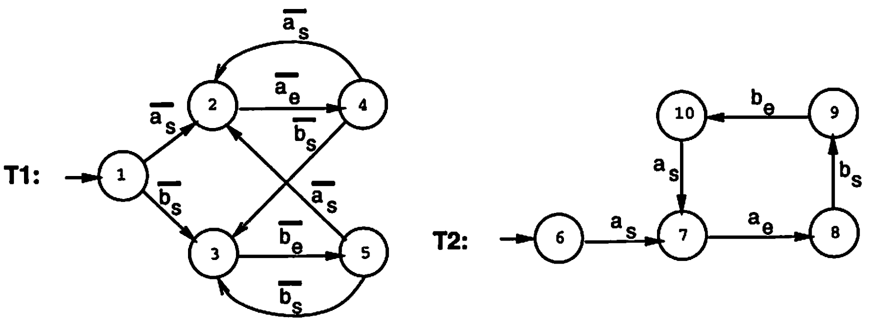

Figure 4: TIGs for example

Task Interaction Graphs

TIGs have been proposed by Long and Clarke [LC89] as a compact flow graph representation for concurrent programs. TIGs divide tasks into maximal sequential regions, where such task regions define all of the possible behaviors between two consecutive task interactions. TIGs are a coarsened flow graph representation analogous to basic block control flow graphs, where regions are blocks. Whereas basic block control flow graphs demark blocks by labels and branches, TIGs demark regions by inter-task communication statements.

Definition 2 *A task interaction graph (TIG) is a tuple (N, E, S, T, L, C), where N is the set of nodes representing task regions, $E : N \rightarrow N$ is the set of edges representing task interactions, $S \in N$ is the start node, $T \subseteq N$ is the set of terminal nodes, $L : E \rightarrow \Sigma_{label}$ 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 may begin and the terminal nodes represent regions where task execution may end. Each node has a fragment of code associated with it that represents Ada statements in the task region plus two types of non-executable statements, ENTER and EXIT, that mark region entry and exit points. The ENTER and EXIT statements take the task interaction as an argument and EXIT 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. Considering only Ada entry calls and accept statements, there are four distinct kinds of tasking interactions: starting and ending an entry call, and starting and ending an accept statement.

As with all coarsened representations, we need to provide access to relevant information about program behavior that has been abstracted in the representation. For TIGs, information about the potential for execution to permanently *block* is required for certain analyses, such as checking for deadlock freedom. To support efficient analysis, this information is summarized by labeling TIG edges as either *blocking* or *non-blocking*. If execution reaches an entry call, accept statement, or select statement without an else or delay alternative, then execution of the task blocks until another task reaches the rendezvous; edges representing these interactions are blocking. If execution reaches a selective entry call or a select statement with an else or delay alternative, then execution of the task does not block waiting for another task; edges representing these interactions are non-blocking.

To illustrate these ideas consider the initial region of T1, `C(1)` in figure 3. Region 1 is entered 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 non-blocking accept for a and the second is on the start of the blocking accept for b. A TIG represents the semantics of control flow branching, such as the select-else statement, within a TIG node. The TIGs for tasks T1 and T2 are given in figure 4. Since a region represents all execution paths between a given task interaction and any succeeding interactions, it is possible for program statements to be associated with multiple TIG nodes. In the example of figure 4, there are 3 edges corresponding to the statement EXIT($\overline{a_s}$,2) in regions 1, 4 and 5. 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, for example a_s and $\overline{a_s}$.

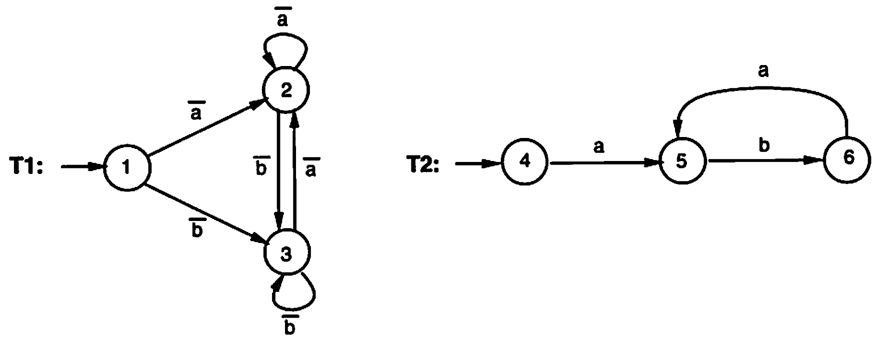

Figure 5: Reduced TIGs for example

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 5. We refer to these as *reduced TIGs* and drop the subscripts when referring to interaction names in this context.

5 TIG-based Petri nets

Petri nets are one way to model the behavior of concurrent program; we propose a Petri net model for Ada tasking programs that is constructed from a set of TIGs and therefore hides many of the details of task control flow.

Definition 3 *A* ***TIG-based Petri net*** *is a tuple (P, T, F, M_0), where $P = N$ is a set of TIG regions, $T \subseteq E \times E$ is a set of pairs of TIG edges, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, and $M_0 = (S_1, S_2, \ldots, S_k)$ is the initial marking representing the start nodes for each of the k tasks.*

It is clear that 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 task region and each transition represents a potential task interaction. Intuitively, the TPN is a merge of a collection of TIGs where transitions represent joint communication events between some pair of tasks. TPNs can be constructed using the following algorithm:

Algorithm 1 (TPN Construction)
Input:

A set of TIGs $\{T_1, T_2, \ldots, T_k\}$.

Output:

A TPN.

Main Loop:
Let $Call_a$ and $Accept_a$ be the labels of communication statements for an entry whose name is a. Let $Src(x)$ and $Dest(x)$ be the source and destination TIG nodes for a TIG edge x.

```
(1) $M_0 = (S_1, S_2, \ldots, S_k)$
(2) $P = \cup_{i=1}^{k} N_i$
(3) $T = \emptyset$
(4) for each edge, $x \in \cup_{i=1}^{k} E_i \wedge L(x) = Call_a$ loop
(5)     for each edge, $y \in \cup_{i=1}^{k} E_i \wedge L(y) = Accept_a$ loop
(6)         $T = T \cup (x, y)$
(7)         $F = F \cup (Src(x), T) \cup (Src(y), T) \cup$
                $(T, Dest(x)) \cup (T, Dest(y))$
        end if
    end loop
```

A TPN marking corresponds to a program termination state if all the marked places correspond to terminal TIG nodes. A similar algorithm can be used to construct a CFGPN from a set of task control flow graphs.

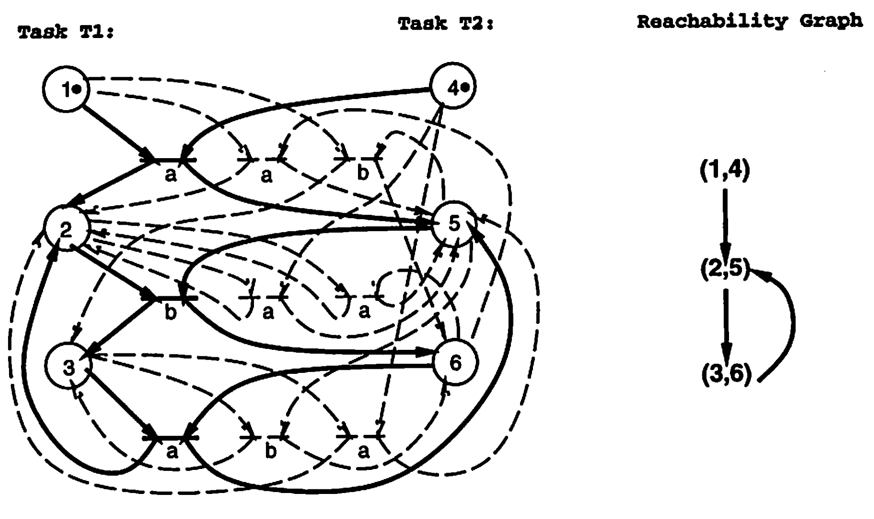

Figure 6: TIG-based Petri net and Reachability Graph for Example

Algorithm 1 constructs a Petri net that overestimates the possible task interactions of the program. All potential task interactions are included as a result of the exhaustive matching of TIG edge labels; however, some of these interactions may never be executed. Algorithm 1 is $O(CA)$ where C is the set of entry call edges and A is the set of entry accept edges in the set of input TIGs. The cost of the loop on line 4 is maximized when the total number of TIG edges is split evenly between Calls of a and Accepts of a; since each task has at least one node and each node has at least one incident edge, the cost of the loop on line 4 dominates. We note that, in general, the cost of constructing the TPN will be much less than this bound. This is due to the fact that in most programs there are multiple entries and calls and accepts to different entries will not match.

As we can see from Algorithm 1, the total number of places in a TPN is the sum of the number of nodes in the TIGs representing the program. For every task interaction contained in a task there is a single TIG region, for which the interaction is the entry point. The number of task interactions in a TIG is linear in the number of communication statements in the task, as we can have at most 2 interactions for a single communication statement in the case of an accept with a body. Thus the number of TIG nodes and hence the number of TPN places is linear in the number of communication statements in the program.

In Algorithm 1, a TPN transition is created for each syntactic matching of edge labels. The potential for having multiple TIG edges corresponding to a single call or accept statement in the source program, as described in section 4, results in additional TPN transitions. There are pathological 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 dead, and hence do not contribute to the complexity of TPN based reachability analysis.

Continuing with our example, figure 6 illustrates the TPN constructed from the reduced TIGs in figure 5, where the firable transitions and arcs are in bold and the dead transitions and arcs are dashed. This example illustrates a number of the benefits of the TPN representation. Each task communication has a simple representation; a single TPN transition that has calling and accepting input and output places. There is a single marked place in the set of places associated with each task in the program that keeps track of the local state of each task. We have found that the regular structure of TPNs simplifies reasoning about the correctness of the TPN representation and TPN based analysis[1]. TPNs typically contain fewer places and transitions than CFGPNs. For comparison, the TPN for the example in figure 1 has 6 places and 9 transitions. The example CFGPN in section 4 has 16 places and 13 transitions. An Ada-net [SMBT90] is a CFGPN designed to model Ada programs. An Ada-net for this example has 21 places and 16 transitions [For91]. In the next section, we will see that for a number of examples the smaller TPNs result in a significant reduction in the size of the reachability graph.

6 Evaluating TPNs for Concurrency Analysis

Experimental evidence suggests that construction of the reachability graph is the limiting factor in performing state reachability analyses; if we cannot build the program model then we have no hope of reasoning about it. If

[1]The structure and semantics of TPNs is also conducive to visualization, but in program analysis applications the size of the nets are usually too large to allow for effective visualization of program behavior.

we can construct the reachability graph, then we at least have the opportunity to reason about desired program behaviors. In fact, it is often practical to check the property of interest on each of the states in the graph. Thus, to judge the effectiveness of TPNs as the basis for practical reachability analysis we need to consider both the benefit of reducing reachability graph size and the (potential) increase in the cost of analyzing that graph.

In this section, we demonstrate that the size of a TPN-based reachability graph is much smaller than an equivalent statement-level model derived from a CFGPN. We then discuss the cost of reasoning over these models. We present algorithms for checking two different properties of concurrent programs over a TPN-based reachability graph. Finally, we evaluate the cost of performing this reasoning both analytically, where possible, and empirically.

6.1 Methodology

We collect data on the size of TPNs, the size of reachability graphs, and the cost of two different analysis algorithms applied to reachability graphs. This data was collected using the *TPN toolset*. With this toolset, constructing a TPN from Ada source code involves executing the Arcadia [TBC+88] language processing tools to generate a collection of CFGs. The CFGs are then converted to TIGs from which the TPN is constructed. Transformations can be applied to reduce the size of the TPNs. A reachability graph is generated from a TPN using standard Petri net techniques [MR87]. A variety of analysis algorithms can then be applied to the TPN-based reachability graph including checking for deadlock freedom, checking for freedom from critical races, and performing data flow analyses to check for event or state sequencing properties.

The goal of this work is to understand the circumstances under which TPN-based representations are beneficial for concurrency analysis. For the purpose of comparative evaluation, we need an alternative approach against which to measure potential improvements. Finding a suitable approach for comparison, however, is difficult for a number of reasons.

A fair comparison requires that both techniques be equivalent in the kinds of information they model and the kinds of analyses they support. There are a great variety of program models and analysis algorithms. As Corbett discusses [Cor94], different models and algorithms can be sensitive to subtle variations in the input program. Thus, a comparative evaluation can easily lead to unintended biasing of the results. Moreover, program analyses are not useful in the abstract; their worth derives from application to "real" programs. So, it makes sense to compare the effectiveness of analyses on "realistic" programs. While empirical evaluation of static concurrency analysis techniques has increased in recent years, there is still relatively little data, especially for "realistic" programs. This is due, in part, to the high cost of constructing a robust set of analysis tools with which to conduct such evaluations. Another concern is, how to go about measuring the cost of analysis techniques for the purpose of comparing them. While it may seem natural, from a users perspective, to compare analysis run-time, the results can be misleading. In comparing two analyses by studying the time it takes to check the same behavior on the same program, we end up comparing two "implementations" of the analyses. It can be difficult to tell whether the model, analysis algorithm, or implementation decisions are the key factors in determining analysis cost. Unfortunately, for very different models and reasoning algorithms there is no alternative. We cannot compare the models directly or analytically derive the amount of work required to reason about the model.

We are fortunate to have access to analysis results from the TOTAL toolset [SMBT90]. TOTAL performs state space analysis by constructing the reachability graph from a statement level Petri net model of Ada tasking programs, called *Ada-nets*. Ada-nets are a kind of CFGPN since they explicitly represent control flow decisions in the structure of the Petri net. We compare our empirical findings for TPN-based analyses to those for Ada-net-based analyses. Given that TPNs and Ada-nets, and their reachability graphs, are equivalent in information content we can compare the size of the models directly. Since the analysis algorithms traverse the reachability graphs and perform local checking on each node, we can compare the cost of reasoning about reachable states directly. We use Ada-nets, their associated reachability graphs, and the appropriate analysis algorithms for comparison.

Analysis data based on Ada-nets is available for four example Ada tasking programs [DBDS94, Sha93], `BDS`, versions of `Gas-1`, `Phils` and the `RW` examples. `BDS` is a simulation of a border defense system [DBDS94]. It contains 15 tasks and has entry calls and accept statements nested within complicated control flow structures. `Gas-1` are versions of the one pump gas-station example without deadlock and with the operator task unrolled to accept separate customer entries [ABC+91]. `Phils` are versions of the basic dining philosophers example with deadlock [ABC+91]. `RW` are versions of the readers/writers example presented in [ABC+91]. The last three examples are scalable; for the gas station the number of customers, for dining philosophers the number of philosophers and forks, and for readers/writers the number of reader and writer tasks can all be varied.

Table 1: TPN and Ada-net data

Example	Tasks	Ada-net				TPN			
		Petri net		Reachability Graph		Petri net		Reachability Graph	
		Places	Transitions	States	Arcs	Places	Transitions	States	Arcs
BDS	15					107	135	-	-
BDS opt	15	263	220	-	-	96	128	285006	1952588
Gas-1 3	7					60	89	934	1764
Gas-1 3 opt	7	157	141	79153	293490	39	75	493	987
Gas-1 5	9					90	185	20141	50140
Gas-1 5 opt	9	313	309	-	-	59	163	9746	26785
Phils 3	6					43	36	268	576
Phils 3 opt	6	72	54	18900	79083	25	24	84	186
Phils 5	10					71	60	11744	42440
Phils 5 opt	10	120	90	-	-	41	40	1653	6130
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	93	92	-	-	17	48	41	119
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	138	143	-	-	23	81	579	2884
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

6.2 Building Reachability Graphs

Reachability graphs for concurrent systems are well-understood. Reachability graphs are derived from Petri net representations using the transitive closure of the Petri net firing rules over M_0, the initial marking.

Definition 4 *A **reachability graph** is a directed graph (S, T, s, F) with a set of nodes S, called **states**, a set of edges T, called **transitions**, a distinguished start state $s \in S$ and a set of terminal or final states $F \subseteq S$.*

States of the graph are reachable markings of the Petri net. A transition in the reachability graph corresponds to the firing of a single Petri net transition. The start state corresponds to M_0. Final states of a reachability graph correspond to Petri net markings in which all of the marked places model the termination of a task.

The structure of a TPN 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 next to it, where nodes represent TPN markings.

Evaluating the Size of Reachability Graphs

Table 1 presents data on the size of TPNs built from unreduced and reduced TIGs, in terms of places and transitions, and on the size of the corresponding reachability graphs, in terms of states and arcs. We indicate with opt that the TIGs used to construct the TPN have been reduced as discussed in section 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 are available. For scalable examples, the number of tasks is given to indicate the scale of the program that was analyzed.

We present Ada-net results for the same examples. The two examples for which data are available from reachability analysis of Ada-nets and TPNs are the `Gas-1 3` and `Phils 3` examples. Comparison of these data illustrates the reachability graph compaction that can be gained by using TPNs; 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 [DBDS94]. If we assume that reachability graphs are at least that large for the examples where reachability graphs for Ada-nets could not be generated, then our results for the `Gas-1 5`, `Phils 5`, and `Phils 7` examples also 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. In Section 7 we discuss extending the applicability of TPN and Ada-net analyses by transforming the Petri nets prior to reachability graph construction.

6.3 Reasoning over Reachability Graphs

Analysis of TPNs involves first computing summary information and then checking the intended behavior over the representation using the summarized information. For TPN-based reachability graphs a state in the graph represents a collection of marked TPN places; the summary information for such a state is derived from the summary information of the marked places. Different analysis algorithms will require different kinds of summary information.

Checking reachability properties of a program involves defining a *property predicate* that decides whether a TPN marking satisfies the property in question. We evaluate this predicate for each state of the reachability graph to determine if any reachable TPN marking violates the property. Property predicates make use of problem-specific summary information. These predicates are defined to be conservative, in the sense that they never return a false result when a marking corresponds to a state in which the property in question is true.

For many property predicates the summary information required is TPN place summary information. Thus, to analyze different properties over a single reachability graph we need only compute different TPN summaries, rather than reachability graph state summaries. This can yield significant savings in practice; the cost of state summaries can be much more expensive than TPN summaries, and the cost of constructing the reachability graph can be amortized over multiple analyses.

We illustrate summary information and property predicates by presenting two examples: checking whether a TPN marking indicates the existence of a critical race in the program and checking whether a TPN marking corresponds to a program deadlock state.

Checking for Critical Races

An important global property of concurrent programs is freedom from critical races. *Write-write* critical races occur when tasks that define the value of a shared variable execute such that the writes in one task may either precede or follow writes in another task. This can be problematic, as the value subsequently read from the shared variable depends on the order of the writes.

Shared variables can be identified by scanning the set of variables defined and used by each task in the program. For each shared variable and for each TPN place, we summarize whether a write to that variable is contained in the program region corresponding to the place.

Definition 5 *A critical race summary is bit-vector of length v, where v is the number of shared variables in a program and a $TRUE$ value in the ith element indicates a write to the ith shared variable.*

We compute critical race summaries for each place in a TPN using a simple depth-first walk of the corresponding code fragment. The total cost of computing all summaries is linear in the number of program statements.

Note that we can compute the summary information *a priori* of any analysis, and thus amortize that cost over many analyses. Alternately, we can compute the summary information during analysis. The decision depends on whether we will use each piece of summary information once or multiple times; in the latter case, pre-computing summaries will reduce overall analysis cost. In the following, we have assumed that TPN summary information is pre-computed.

The following algorithm determines whether a critical race on any shared variable occurs in a given TPN marking.

Algorithm 2 (Critical Race Property Predicate)
Input:

A TPN marking $M = [s_1, s_2, \ldots, s_k]$ and critical race *Summary* information for each TPN place.

Output:

$TRUE$ if the marking may correspond to a critical race.

Main Loop:
Let *WriteFound* be a bit-vector of length v that records writes to shared variables associated with the marked places.

```
(1) WriteFound := $(0, 0, \ldots, 0)$
(2) for i in 1..v do
(3)    for j in 1..k do
(4)       if $Summary(M[j])[i] = 1$ then
(5)          if $WriteFound[i] = 0$ then
(6)             $WriteFound[i] = 1$
             else
(7)             return TRUE
             end if
          end if
       end for
    end for
(8) return FALSE
```

Intuitively, for each shared variable, the algorithm checks whether some TPN place in the marking writes it; if so, it records that fact. Upon encountering a write to a variable, if a write to that same variable has already been recorded, then the algorithm indicates the potential for a critical race. A slight variant of this algorithm can be used to point the user at regions of source code that may contain critical races. We note that other co-executability properties, such as mutual exclusion, can be checked using similar property predicates.

Checking this property predicate for a given TPN marking requires $O(vk)$ steps since the critical race summaries are pre-computed.

To the best of our knowledge, checking for freedom from critical races is not implemented in the TOTAL toolset. The approach described above, however, could be easily adapted to Ada-nets. Computing summary information would proceed as in the case of TPNs. Once the summary information is computed, Algorithm 2 can be applied to an Ada-net marking. Unlike TPNs, an Ada-net place corresponds to a single program statement; thus, the cost of computing summary information for an Ada-net place will be cheaper than for a TPN place. The total cost for computing all summary information, however, is linear in the number of program statements for either representation. Thus, the cost of checking for critical races using either a TPN or Ada-net marking is equal.

Checking Deadlock

Freedom from deadlock is checked by determining that no combination of the individual task states for a reachable TPN marking correspond to a program deadlock state. 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 may correspond to a deadlock. TPN places summarize, through their associated TIG nodes, the control flow choices that need to be considered to determine deadlock markings. For the TIG nodes associated with a TPN marking we reason about all possible combinations of blocking exiting edges, where one edge is taken for each TIG node [2].

Definition 6 *A choice combination for a TPN marking $M = [s_1, s_2, \ldots, s_k]$ is a vector $(e_1, e_2, \ldots, e_k)$ where e_i is an exiting blocking edge of the TIG region corresponding to place s_i.*

The choice combinations are easy to compute, but, in the worst-case, there can be many of them. Under standard assumptions about program control flow branching, i.e., that the number of branches is bounded by some constant, the number of exiting edges from a region is $O(c)$, for some constant c. The total number of combinations of exiting TIG edges across all k of the regions associated with a TPN marking is therefore $O(c^k)$.

Intuitively, a TPN marking corresponds to a potential program deadlock if the marking does not represent a terminal state of the program and if there exists a choice combination such that no pair of edges in the combination are matching communications. Non-blocking edges exiting a TIG node can never contribute to a program deadlock, since they can always be bypassed, thus they are not included in choice combinations.

We compute the following summary information:

[2] For clarity our presentation is in terms of edges. In practice, we group together edges that are branches of the same select statement and select choice combinations appropriately. This improves both the efficiency and accuracy of checking a TPN marking for deadlock.

Definition 7 *A deadlock summary for a TPN place is a pair of bit-vectors of length e, where e is the total number of task entries in the program. A value of TRUE in the ith bit of the accept summary vector indicates that the TPN place has a blocking communication that accepts the ith task entry. A value of TRUE in the jth bit of the call summary vector indicates that the TPN place has a blocking communication that calls the jth task entry.*

Using choice combination and summary information we can check for the possibility of deadlock at a TPN marking.

Algorithm 3 (Deadlock Property Predicate)
Input:

A TPN marking $M = [s_1, s_2, \ldots, s_k]$ and deadlock summary information, *Accept* and *Call*, for each TPN place.

Output:

TRUE if the marking may correspond to a program deadlock.

Main Loop:
Let *AllCalls* and *AllAccepts* be a bit-vectors of length e that record blocking entry calls and accepts at a TPN place.

```
(1) if M is a terminal marking then
(2)    return FALSE
    end if
(3) for each choice combination, C, of M loop
(4)    AllAccept = $\bigcup_{p \in C} Accept(p)$
(5)    AllCall = $\bigcup_{p \in C} Call(p)$
(6)    if (AllCall $\cap$ AllAccept) = $(0, 0, \ldots, 0)$ then
(7)       return TRUE
       end if
    end loop
(8) return FALSE
```

To illustrate, consider the deadlock property applied to the reachable TPN marking $(2, 5)$ in Figure 6. For this example, TPN place 2 has a single exiting, blocking edge for the accept of b with an *Accept* summary of $(0, 1)$ and a *Call* summary of $(0, 0)$ and TPN place 5 has a single exiting, blocking edge for the call of b with an *Accept* summary of $(0, 0)$ and a *Call* summary of $(0, 1)$. The single edge choice combination considered by the deadlock predicate for TPN marking $(2, 5)$ computes the bit-vector expression $(((0, 1) \vee (0, 0)) \wedge ((0, 0) \vee (0, 1))) = (0, 1)$ so the predicate returns FALSE.

The cost of Algorithm 3 is dominated by the cost of the loop. The body of the loop is $O(k)$ under the assumption that the bit-vector operations are $O(1)$; this is reasonable for a wide variety of programs since the number of communication channels typically grows slowly with the size of a program. Thus, the total cost of Algorithm 3 is $O(c^k)$. We note that Young et. al. [YTL+95] showed this problem to be NP-hard, but they have found empirically that for a number of programs, checking this condition is practical.

Checking for freedom from deadlock is supported by the TOTAL toolset. For most statement-level Petri net representations, including Ada-nets, the deadlock predicate is very simple. We need only check whether a reachable marking has any outgoing arcs; if there are no outgoing arcs and it is not a terminal marking, then it is a potential deadlock.

Thus, the cost of checking for deadlock at an Ada-net marking appears to be considerably cheaper, in the worst-case, than checking for deadlock at a TPN marking. The next subsection, however, considers the total cost of analysis when the reachability graph and property predicate are both taken into consideration.

6.4 Evaluating Total Analysis Cost

We have presented the TPN-based reachability graph model and property predicates. We have also compared them to analysis using the Ada-net model. To reason about program behavior we need to apply the property predicate to, in the worst-case, every reachable marking of the model. Our analysis cost is bounded by the product of the number of reachability graph states and the cost of checking a property predicate at a state.

For checking that a program is free of critical races it appears that the TPN-based approach results in significant savings in analysis cost over Ada-net-based analysis. While the cost of the analysis algorithm is the same for TPNs

Table 2: Deadlock Predicate Cost Data

Example	Tasks	Entries	Average CC	Cost Ratio	Marking Ratio
BDS	15	18	3.83	40.2	-
Gas-1 3	7	10	1.31	7.2	.006
Gas-1 5	9	14	1.33	9.1	-
Phils 3	6	6	1	5.0	.004
Phils 5	10	10	1	7.7	-
Phils 7	14	14	1	10.3	-
RW 2/5	8	4	1	6.3	-

and Ada-nets, the size of the reachability graph is much smaller for TPNs. Thus, the TPN approach has a clear benefit for this analysis.

For checking that a program is free of deadlock, the comparison is not as clear. For TPNs, the deadlock property predicate can be costly to check. To reason about the cost of checking the deadlock property predicate over a TPN-based reachability graph, we compute the average number of operations required to check the predicate at a TPN place. We do this analytically based on Algorithm 3. We assume that each of the following operations has unit cost: bit-vector operations, accessing a Petri net marking, checking for the existence of successor markings, and checking that a given marking corresponds to a terminal program state. Under these assumptions we can define the cost of checking for deadlock at a TPN and Ada-net marking as:

$$\begin{aligned} Cost_{TPN} &= 2 + CC(2 * T + 1) \\ Cost_{Ada-net} &= 3 \end{aligned}$$

where CC is the number of choice combinations for the marking and T is the number of tasks in the program. For either model we need to access the marking and check if it is terminal. For Ada-nets we also check for the existence of successor states, and for the TPN case we have the cost of processing the loop from Algorithm 3. A component of the TPN toolset computes the average number of choice combinations over the set of reachable TPN marking for a given program; this data on choice combinations incorporates the notion of edge groups mentioned previously.

Table 2 presents the number of tasks and average number of choice combinations. We include the number of entries in order to validate our claim that the bit-vector operations are $O(1)$. The table includes the *cost ratio* which is the ratio of the cost of checking the deadlock predicate at a TPN marking to the cost of checking the deadlock predicate at an Ada-net marking. The table includes the *marking ratio* which is the ratio of reachable TPN markings to reachable Ada-net markings. For many of the example programs the Ada-net based reachability graph could not be generated; in those cases we mark the undefined ratio with '-'. We do not include the smaller RW examples; for each of those examples, the cost ratio was less than the ratio for RW 2/5 and the marking ratio was undefined.

Clearly, the ability to analyze examples for which existing control flow graph based Petri net analyses could not be constructed is a significant advantage of TPN-based analysis. In the cases where both analysis techniques provide results the TPN-based analyses enjoy a factor of 22 to 45 reduction in analysis cost.

The cost of checking the TPN deadlock predicate varies with the program under analysis and is a non-trivial increase over the cost of checking an equivalent predicate over Ada-net markings. The variation in cost depends both on the complexity of intra-task control flow, as in the case of the BDS program, and on the number of tasks in the program, as in the case of the scalable `Gas` and `Phils` programs.

As discussed in section 1, TPN-based analysis represents a tradeoff in encoding information in the program representation versus increased cost in the analysis algorithms. 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. Using the smaller TPN-based reachability graph is superior whenever efficient predicates are available. When the cost of checking a predicate on a reachable TPN marking is greater than the cost of checking the corresponding CFGPN predicate, this increased cost may be compensated for by the reduced number of states in the reachability graph itself, as was demonstrated by the data in this section. Of course, in cases where the CFGPN reachability graph is too large to construct and the TPN reachability graph can be generated, TPN based analysis is the better choice.

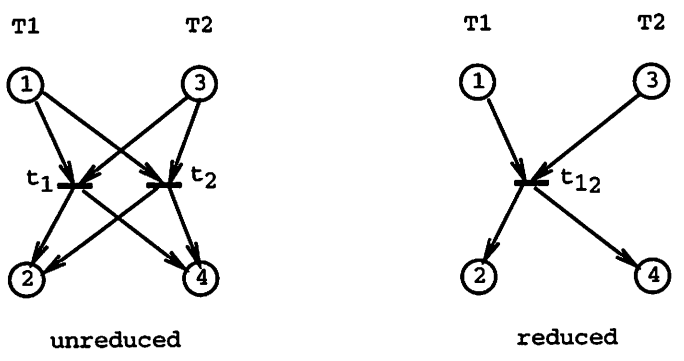

Figure 7: Example of parallel transitions reduction

7 Extending the Applicability of Reachability Analysis

Although TPNs appear to be an improvement over CFGPNs for reachability analysis, TPNs still suffer from explosive growth in the size of the reachability graph, which makes them impractical as a basis for analysis of large, complex concurrent programs. In this section, we consider techniques for reducing a TPN prior to reachability analysis.

The theory of Petri net reductions [Mur89] allows a given net to be replaced by a *reduced* net that maintains certain properties of the original net and has a smaller reachability graph. Most Petri net representations of concurrent programs, including TPNs, explicitly represent all potential inter-task communications. Unlike a collection of individual task representations, such as TIGs, this allows for the identification of reduction opportunities and transformation of the net prior to generation of the reachability graph. The decoupling of reduction considerations from graph generation allows a series of reductions to be independently applied before the reachability graph is constructed. For example, consider deadlock detection. As noted above, to conservatively detect program deadlock, reachability analysis of some Petri net representations can test for the existence of markings that have no successors. A net reduction must preserve this information so that each reachable marking without successors in the original net corresponds to some reachable marking without successors in 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 [DBDS94]. Here we discuss two TPN reductions: *parallel transitions* and *forced communication pairs.*

Parallel transition reductions preserve all information in the reduced TPN and thus can be applied to improve the effectiveness of analysis for any property. The parallel transition reduction merges transitions that have the same input and output places. These TPN structures arise when communication statements are nested within multiple control structures. The transitions represent different control flow choices that can be made within the TIG regions corresponding to the input places. In the context of our analysis, parallel TIG edges and their associated TPN transitions are redundant and can be deleted. Figure 7 illustrates the effect of the reduction. Transitions t_1 and t_2 are merged into a single transition t_{12}. This reduction has the potential to greatly reduce the number of arcs in the reachability graph, thereby reducing the time it takes to generate the set of reachable TPN markings.

Forced communication pair reductions only preserve the deadlock property predicate in the reduced TPN. Unfortunately, program deadlocks are not conservatively represented by the set of reachable TPN markings without successors, so we cannot directly apply existing deadlock-preserving Petri net reductions. Net reductions can be developed, however, that are applicable to TPNs. We use the semantics of the property predicate to develop reductions by extracting necessary conditions for the predicate to hold. If we find that a necessary condition for the property predicate to hold is false for a TPN fragment, then we know that the tested fragment cannot participate in a reachable TPN marking that corresponds to that property.

The forced communication pair reduction takes advantage of the existence of a sequence of communications between two tasks. Here we illustrate how it can be applied to preserve deadlock in the reduced TPN. Informally, this reduction can be applied when no other tasks attempt to communicate with the pair during a sequence of communications and when all choice combinations for the communicating pair contain matching communications. Figure 8 depicts a simple example of forced communication, where tasks T1 and T2 engage in a series of 3 communications at `T2.start`, `T1.exchange`, and `T2.stop`. The reduction is based on the semantics of the deadlock property predicate. Given the conditions on applying the reduction, it is always the case that whenever we reach a TPN marking in which the start places of the forced communication are marked (in our example 1 and 5) we always execute the rest

```
task body T1 is
begin
 loop
   ...
   T2.start;
   — local work
   accept exchange;
   — local work
   T2.stop;
   ...
 end loop;
end T1;
```

```
task body T2 is
begin
 loop
   ...
   accept start;
   — local work
   T1.exchange;
   — local work
   accept stop;
   ...
 end loop;
end T2;
```

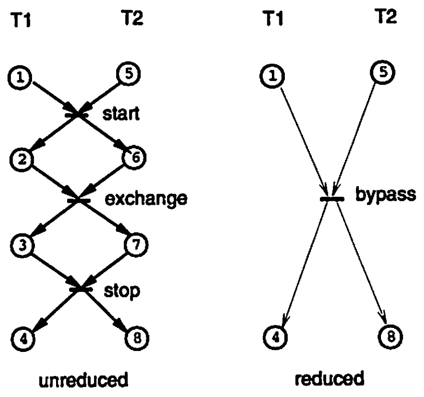

Figure 8: Example of a forced communication reduction

of the TPN fragment associated with the forced communication resulting in the end places (in our example 4 and 8) being marked. We introduce a transition into the TPN that bypasses the portion of the TPN representing the forced communication and delete the bypassed portion of the TPN. We can detect forced communication pairs by looking for the boundary communications, in our example *start* and *stop*, then verifying that the rest of the pattern is suitable for reduction. This reduction yields a decrease in both the number of reachable TPN markings and the number of arcs in the reachability graph.

We can generalize forced communication reductions to allow a more complicated pattern of communication between a pair of tasks, to consider more than a pair of TPN places, and to preserve properties other than deadlock. Consider a region of the reachability graph that is entered through a single marking and exited through a single marking. If we can verify that no markings in this region violate the property we are interested in, then we can bypass it.

Unlike existing Petri net reduction approaches that consider low-level detailed semantics, the forced communication reduction reasons about necessary conditions for executable program behavior at a very high-level and has the potential to collapse large portions of a Petri net.

8 Conclusion

In this paper we have presented a compact Petri net representation for Ada tasking programs. Empirical evidence shows that TPNs are 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 so. We introduced the concept of a property predicate and provide evidence that checking such predicates over the set of reachable TPN markings is practical. We showed how property-dependent and property-independent TPN reductions have the potential to improve the cost-effectiveness of reachability based analysis. Although those well-versed in concurrency analysis could define, and appropriately verify, property predicates and property-specific reductions, we envision that typical end users will select from a library of existing property predicates and reductions that address common properties of interest.

It has been suggested that no single technique is suitable for analysis of all properties of all concurrent programs. TPNs bring elements of Petri net and TIG-based reachability analysis together. TPNs represent a different tradeoff between encoding information in the program representation versus the analysis algorithms than has traditionally been made for Petri net representations. This paper has presented preliminary results to support our hypothesis that reachability analysis of a representation that reduces the size of the state space, perhaps by increasing the cost of checking properties of program states, is more practical than reachability analysis of non-reduced representations. This work has established a framework from which we intend to further explore the limits of practical state space analysis of concurrent programs.

Acknowledgments: The authors wish to acknowledge Kari Nies, whose contributions to earlier versions of this work were invaluable, and Michal Young, Tim Chamillard and Gleb Naumovich for helpful discussions and feedback.

References

[ABC+91] 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.

[And91] G.R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, mar 1991.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the European Software Engineering Conference*, 1993.

[Cor94] J.C. Corbett. An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. *Software Engineering Notes*, pages 204–215, August 1994. Proceedings of the International Symposium on Software Testing and Analysis.

[DBDS94] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in ada. *ACM Transactions on Software Engineering and Methodology*, 3(4):340–380, October 1994.

[DC94] M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[DS91] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification*, Victoria, Canada, October 1991.

[For91] K. Forester. TIG-based Petri nets for modeling Ada tasking. Master's thesis, University of Massachusetts, Amherst, MA, June 1991.

[GW91] P. Godefroid and P. 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.

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

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

[MR91] S.P. Masticola and B.G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, May 1991.

[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.

[PTY92] M. Pezzè, R.N. Taylor, and M. 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.

[Sha93] S.M. Shatz. Personal Communication, February 1993.

[SMBT90] S.M. Shatz, K. Mai, C. Black, and S. 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.

[Tay83] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TBC$^+$88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of SIGSOFT88: Third Symposium on Software Development Environment*, pages 1–13, November 1988. Published as ACM SIGPLAN Notices 24(2) and as SIGSOFT Software Engineering Notes, 13(5) November 1988.

[YTL$^+$95] M. Young, R.N. Taylor, D.L. Levine, K.A. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64–106, January 1995.

[YY91] W.J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification*, pages 49–59, Victoria, Canada, October 1991.