A Compact Petri Net Representation for Concurrent Programs [1]

(Research Paper)

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

email: **dwyer@cs.umass.edu**
⋆Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

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

Abstract

This paper presents a compact Petri net representation that is efficient to construct for concurrent programs that use explicit tasking and rendezvous style communication. These Petri nets are based on task interaction graphs and are called *TIG-based Petri net*s (TPN)s. They form a compact representation by abstracting large regions of program execution with associated summary information 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 present experimental results that demonstrate the benefit of TPNs over alternate Petri net representations and discuss the applicability of Petri net reduction techniques to TPNs.

[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 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, it is not known how they differ, particularly when applied to real production software systems. Most have theoretical bounds that are daunting, but preliminary experimental results seem to indicate that there are applications in which they might be cost-effective. One of the goals of our work is to understand the practical limitations of state space enumeration techniques for "real" programs and to compare that to other techniques, such as symbolic modeling checking, integer linear programming techniques, data flow analysis, and compositional approaches.

State space enumeration methods consider each reachable program state to determine whether a program satisfies a given property [15, 21, 23, 25]. 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 [10, 22]. For some programs, state space reduction is able to decrease the cost considerably but, in general, the cost of these techniques grows exponentially.

Symbolic model-checking techniques use a fix-point computation over an encoding of the state transition relation to determine reachability of a given state [3]. 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 [2]. For this approach to be successful the necessary conditions must be strong so that the number of spurious analysis results is small. Unfortunately, in the worst case, the algorithm for analyzing these conditions requires exponential time.

Data flow analysis techniques are one of the few concurrency analysis approaches that do not have exponential cost [4, 6, 8, 14]. 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 [24]. 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. Given that the structure of real concurrent programs [1] and the behavior of analysis technique may vary greatly from program to program, however, we need to evaluate

analysis techniques over a range of real concurrent programs.

To date, there has been little empirical work in this area. 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 [25] and that state space reduction techniques can increase the size of the programs that can be considered still further [7, 22]. A recent study [5] 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.

We have developed a framework for experimenting with a variety of state space enumeration analyses that is based on *task interaction graphs* (TIG)s [11] and Petri nets [17]. Petri nets are a well-studied representation for concurrent systems [16]. This paper presents a Petri net representation, called *TIG-based Petri net*s (TPN)s, that is efficient to construct for concurrent programs that use explicit tasking and rendezvous style communication. This representation summarizes large regions of program execution and makes the relevant information available for 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 representation that reduces the size of the state space, 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 and compare our results to recent work on control flow graph based Petri net representations.

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 present experimental data on the size of the reachability graphs generated from TPNs and on the cost of checking the graph for desired properties. In section 5 we describe how TPNs can be reduced prior to reachability analysis. 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. In principle the representations 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 represented are Ada tasking programs.

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

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

Figure 1: Ada tasking example

Petri Nets

A Petri net is a directed bipartite graph with nodes called *places*, depicted as circles, and *transitions*, depicted as bars.. The edges of the graph are called *arcs*. A *marking* is an assignment of an integer to each place in the net that represents the number of *tokens*, depicted as black dots, at that 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. Formally, a Petri net is 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. In this paper all of the Petri nets discussed are *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 a token. An enabled transition *fires* by removing tokens from each input place and adding tokens to each output place. A transition that is never enabled is called *dead*.

Figure 1 presents a simple Ada program that will be used as an example throughout the rest of the paper. 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 [12, 15, 18, 19]. We call Petri nets that explicitly represent the possible control flow paths in each program task *control flow graph Petri nets* (CFGPN)s. Figure 2 illustrates a typical CFGPN for the example, where rendezvous start and end are represented by separate transitions. We denote start(end) of an entry call by the name of the entry subscripted by $\mathtt{s(e)}$, e.g., a_s. We denote start(end) of an accept statement by the putting a bar over the name of the entry subscripted by $\mathtt{s(e)}$, e.g., $\overline{a_s}$. 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. The reachability graph of this type of Petri net has been used to perform analysis of Ada tasking programs [15, 19].

Task Interaction Graphs

TIGs have been proposed by Long and Clarke [11] as a compact representation for a rich class of tasking programs. 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 task control flow information and thus results in a smaller graph than a traditional control flow graph. An overview of the TIG representation for a subset of Ada is given below; a more

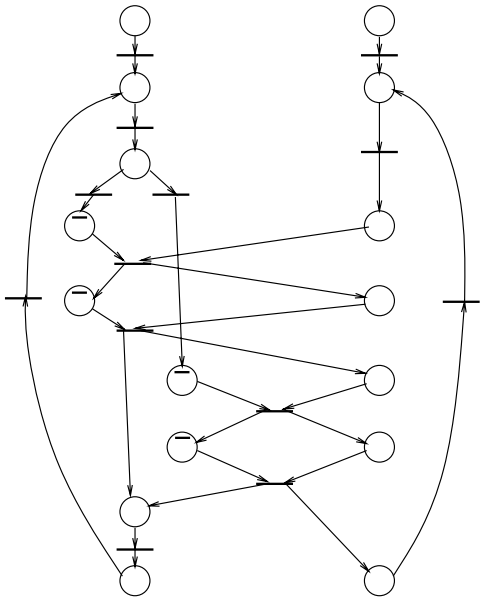

Figure 2: Control Flow Graph Petri Net for example

complete description is found in [11].

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 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 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.

To support efficient analysis, exiting task interactions are labeled 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 accept for `a` and the second is on the start of the accept for `b`. These edges are non-blocking and blocking respectively. In contrast to CFGPN representations 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 regions represent all

Figure 3: Code fragments for task T1 of example

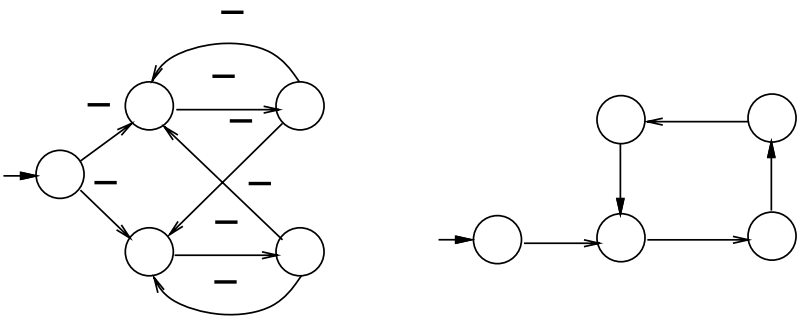

Figure 4: TIGs for example

execution paths between a given task interaction and any succeeding interaction, it is possible for distinct TIG nodes to contain the same program statements. 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. 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, for example a_s and $\overline{a_s}$.

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.

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 task region and each transition represents a potential task interaction. TPNs can be constructed using the following algorithm:

```
Input:  set of TIG
Output:  TPN
Algorithm:
   1) create a place for each TIG region
   2) for each pair of TIG edges with matching labels
      3) create a transition whose input(output) places are the
         source(destination) nodes of the TIG edges
```

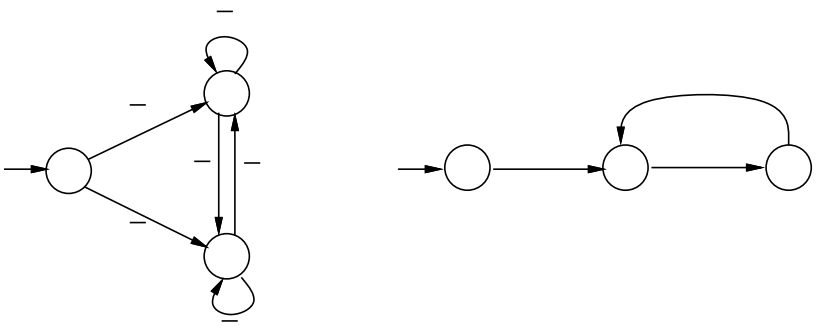

Figure 5: Reduced TIGs for example

The initial TPN marking has the place corresponding to each TIG's start region marked. 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.

The above 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 exhaustive 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 4 the time required to construct the TPN is negligible.

As we can see from the above algorithm, the total number of places in a TPN is the sum of the number of nodes in the TIGs representing the program. There is a single TIG region for every task interaction contained in the modeled task. 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 this algorithm, 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 2, 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. 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 [2]. TPNs typically contain fewer

[2]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.

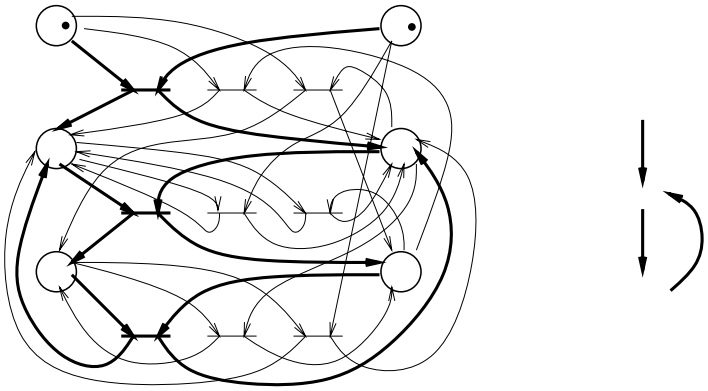

Figure 6: TIG-based Petri net for example

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 2 has 16 places and 13 transitions. An Ada-net [21] is a CFGPN designed to model Ada programs. An Ada-net for this example has 21 places and 16 transitions [9]. In the next section we will see that for a number of examples this reduction in the size of the Petri net representation results in a significant reduction in the size of the reachability graph.

4 Analysis using TPNs

Experimental evidence suggests that construction of the reachability graph is the limiting factor in performing state reachability analyses. Young et. al. [25] discuss separating the construction of the reachability graph from the process of checking a particular property. If we can construct the reachability graph, it is often practical to check the property of interest on each of the states in the graph. We adopt this separation of graph generation and property checking. Thus, 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.

4.1 Reachability

The TPN reachability graph is generated using standard Petri net techniques [15]. 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 next to it, where nodes represent TPN markings. For clarity, we refer to a node in the reachability graph as a *state*, e.g., (1 4); we refer to an edge in the reachability graph as an *arc*, e.g., (3 6) $\rightarrow$ (2 5).

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

Figure 7: TPN and Ada-net data

Evaluating the Size of Reachability Graphs

We have constructed a set of tools to evaluate the suitability of TPNs for analysis of Ada programs. This toolset is built from components produced by the Arcadia consortium. The current implementation recognizes almost all Ada language features, builds reduced and unreduced TIGs, builds a TPN from a set of TIGs, and then builds the TPN reachability graph. In figure 7, we present data for a number of examples on the size of TPNs built from unreduced and reduced TIGs, in terms of places and transitions, and 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 2. 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. `BDS` is a simulation of a border defense system [13]. 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 [2]. `Phils` are versions of the basic dining philosophers example with deadlock [2]. `RW` are versions of the readers/writers example presented in [2]. The number of tasks next to the example name indicates 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 compare TPNs to the Ada-nets generated using the TOTAL system [19] because Ada-nets are an example of the class of CFGPNs described in section 2 and 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-1`, `Phils` and the `RW` examples [7, 22, 20]. The size of Ada-nets, in terms of places and transitions, and the size of the corresponding reachability graphs in terms of states and arcs, are given in figure 7, next to the comparable TPN results. The Ada-net data is independent of reduced or unreduced TIGs so we only present that data once. 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, 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 [7]. 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. While early work on TOTAL relied on straightforward reachability of Ada-nets [19, 21], more recent work [7, 22] has demonstrated that if we are only 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 [3].

4.2 Checking Properties

Checking whether a program exhibits a desired property 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. These predicates are 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. The semantics of a property predicate can be used to construct TPN reductions that preserve the conservativeness of the predicate, as will be shown in section 5. We illustrate 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 writes. Shared variables can be identified by scanning the set of variables defined and used by each task in the program. For Ada tasking programs the language provides a mechanism for identifying shared program variables.

[3] We have not implemented the TPN reductions described in section 5, so comparing deadlock reduced TPNs to deadlock reduced Ada-nets is not possible at this time.

For each shared variable, for each TPN place we summarize whether a write to that variable is contained in the program region corresponding to the place. For a program with v shared variables, each TPN place has an array of v booleans, named `contains_write` in the algorithm below, that records this information. A true value in the ith element indicates that the region contains a write to the ith shared variable. An array of v booleans, named `write_found` in the algorithm, records writes to shared variables associated with the marked places. The following algorithm determines whether a critical race on any shared variable occurs in a given TPN marking.

```
Input:  TPN marking M = [n_1, n_2, ..., n_k]
Output:  True if the marking may correspond to a critical race.
Algorithm:
write_found[1..v] := FALSE
for i in 1..v do
   for j in 1..k do
      if M[j].contains_write[i] then
         if not write_found[i] then
            write_found[i] := TRUE
         else
            return TRUE
         end if
      end if
   end for
end for
return FALSE
```

It is easy to see that a slight variant of this algorithm can be used to point the user at regions of source code that may contain critical races if the predicate returns true for a marking. Checking this property predicate for a given TPN marking requires time that is linear in the product of the number of tasks and the number of shared variable in the program. A number of other co-executability properties, such as mutual exclusion, can be checked using similar property predicates.

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 node, 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. We call these the *choice* combinations for the TPN marking, as they represent the possible communication choices that can be made by the program. 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. [4]

[4] For clarity our presentation is in terms of edges. In practice, we group edges together 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.

We formulate this condition as a deadlock property predicate. For each blocking edge exiting the TIG node associated with a 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 exiting 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 exiting 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:
   if M is a terminal marking then
      return FALSE

   for each choice combination, C, of 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

   return FALSE
```

To illustrate, consider the deadlock property applied to the reachable TPN marking $(2,5)$ in figure 6. For this example, see figure 6, TPN place 2 has a single exiting, blocking edge for the accept of b with an `accept-vector` of $(0,1)$ and a `call_vector` of $(0,0)$ and TPN place 5 has a single exiting, blocking edge for the call of b with an `accept_vector` of $(0,0)$ and a `call_vector` 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.

This algorithm is strongly dependent on the number of choice combinations for a TPN marking. The number of choice combinations is, in the worst case, exponential in the number of tasks in the program. Young et. al. [25] have shown that this problem is NP-hard, but they have found through experimentation that for a number of programs checking this condition is practical.

Evaluating the Cost of Property Checking

For most CFGPNs, including Ada-nets, the deadlock predicate is very simple and only requires checking whether a reachable marking has any outgoing arcs. For TPNs the deadlock predicate is more costly, since in the worst case we must check all choice combinations associated with a TPN marking. This is, in the worst case, exponential in the number of tasks in the program. To get a sense of the cost of checking the deadlock property predicate over a TPN reachability graph in practice, we compute the average number of choice combinations over the set of reachable TPN markings for each example in section 4.1. This is a measure of the work required to check the deadlock property predicate at each reachable TPN marking. For the `Phils` and `RW` examples the computation is trivial, as all TPN places correspond to TIG nodes with a single blocking edge, thus there is a single choice combination to consider at each reachable TPN marking. For the `Gas-1` examples the Operator task has communication statements nested inside control structures. Consequently, a number of TIG nodes for the Operator task have 2 blocking exiting edges. For the

Example	Tasks	Entries	Maximum	Average Combinations
BDS opt	15	18	3	3.83
Gas-1 3 opt	7	10	2	1.31
Gas-1 5 opt	9	14	2	1.33

Figure 8: Choice Combination Data

BDS example there are a number of tasks that have nodes with more than one exiting blocking edge. A component of our toolset computes the number of choice combinations for each reachable TPN marking, and sums these values to get a measure of the work required for checking the deadlock predicate over the entire generated reachability graph. Figure 8 gives the number of tasks, the maximum number of blocking edges for any place in the TPN, and the average number of choice combinations to be considered by the deadlock predicate over the set of markings in the reachability graph for the Gas-1 and BDS examples [5]. We also note that the bit-vector operations described in the property predicates are appropriate since the longest vector for any of these example is 18 bits, the number of entries in BDS.

Its clear that the cost of checking the TPN deadlock predicate varies with the program under analysis. The data for the Gas-1 and BDS examples illustrates that the cost of checking the deadlock predicate is dependent on the complexity of control flow within which communication statements are nested. The work required for checking the deadlock predicate requires that 3 bit-vector operations be executed for each choice combination. Our data shows that for all of our examples the average number of choice combinations is less than 4. Therefore, we can expect that at most the work required to check the deadlock predicate at a reachable TPN marking is approximately 12 times more costly than checking for deadlock at a reachable CFGPN marking; for many of the examples the cost will be closer to 3 times the CFGPN cost. The data presented in section 4.1 indicates that the number of states in a TPN reachability graph is on the order of 100 times less than in a CFGPN reachability graph. So, for these examples, the extra cost of checking the TPN deadlock predicate is compensated for by the reduction in reachability graph size.

As discussed in section 1, TPN based analysis represents a tradeoff in encoding information in the program representation versus 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 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 is too large to construct and the TPN reachability graph can be generated, TPN based analysis is the better choice.

Further experimentation with a range of realistic examples will provide a better indication of the cost of checking TPN predicates versus the size of the TPN reachability graph.

[5] Our data on choice combinations incorporates the notion of groups of edges, mentioned above.

5 Reducing the Cost of 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 [16] 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 [7, 22].

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. 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. Forced communication pair reductions only preserve the deadlock property predicate in the reduced TPN.

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 9 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.

The *forced communication pair* reduction takes advantage of the existence of a sequence of communications between two tasks. 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 10 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

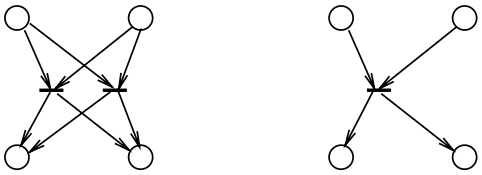

Figure 9: Example of parallel transitions reduction

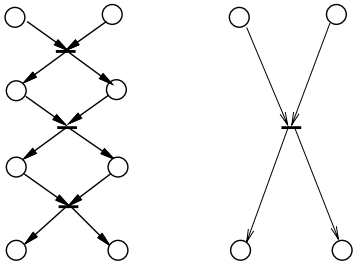

Figure 10: Example of a forced communication reduction

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 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 reduces 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. We have developed algorithms for detecting TPN fragments associated with such regions based on commonly occurring communication patterns in concurrent programs. We intend to explore their potential for reducing the cost of TPN based reachability analysis.

6 Conclusion

In this paper, we have presented a compact Petri net representation for Ada tasking programs. Experimental 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 preliminary 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.

We are continuing to develop and implement TPN reduction and decomposition techniques and tools. These promise to improve the cost-effectiveness of the analysis. For example, we are working on improving accuracy of the analysis results by eliminating some infeasible program paths. Intuitively, this would seem to increase the time and space requirements of analysis, but preliminary work has shown that, for some of the examples discussed in this paper, improved accuracy comes with little or no increase in analysis cost.

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 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.

Acknowledgements

The authors wish to thank Tim Chamillard and Gleb Naumovich for helpful discussions and feedback.

References

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

[2] 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.

[3] 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.

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

[5] J.C. Corbett. An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. 1994. Proceedings of the International Symposium on Software, Testing and Analysis.

[6] 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 (TAV4)*, October 1991.

[7] 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).

[8] M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (to appear).

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

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] W.J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, pages 49–59, Victoria, Canada, October 1991. ACM Press.

[25] M. Young, R.N. Taylor, D.L. Levine, K. Forester, and D. 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.