A Compact Petri Net Representsation and Its Implications for Analysis

Matthew B. Dwyer ⋆
Lori A. Clarke †

CMPSCI Technical Report 96–21
February, 12 1996

⋆ Department Of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
(913)532-6350

† Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

This work was supported by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under contract F30602-94-C-0137. other works, must be obtained from the IEEE.]

A Compact Petri Net Representation and Its Implications for Analysis *

Matthew B. Dwyer

dwyer@cis.ksu.edu

Dept. of Computing and Info. Sciences
Kansas State University
Manhattan, KS 66506
(913)532-6350

Lori A. Clarke

clarke@cs.umass.edu

Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
(413)545-1328

Abstract

This paper explores a property-independent, coarsened, multi-level representation for supporting state reachability analysis for a number of different properties. This multi-level representation comprises a reachability graph derived from a highly optimized Petri net representation that is based on task interaction graphs and associated property-specific summary information. This highly optimized representation reduces the size of the reachability graph but may increase the cost of the analysis algorithm for some types of analyses. This paper explores this trade-off.

To this end, we have developed a framework for checking a variety of properties of concurrent programs using this optimized representation and present empirical results that compare the cost to an alternative Petri net representation. In addition, we present reduction techniques that can further improve the performance and yet still preserve analysis information. Although worst-case bounds for most concurrency analysis techniques are daunting, we demonstrate that the techniques that we propose significantly broaden the applicability of reachability analyses.

1 Introduction

An important goal of software engineering research is to provide software developers with cost-effective techniques for evaluating software. Concurrent systems, which are becoming increasingly common, are particularly difficult to reason about. To address this problem, researchers have proposed a variety of static analysis techniques to help validate properties of concurrent systems. These techniques vary in the time

*This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

and space required for analysis, in the accuracy of their results, and in the types of questions that each can address. Unfortunately, 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, although 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 comparative strengths and weaknesses of these techniques as applied to "real" programs.

Static analyses are typically composed of two fundamental components: a representation, or model, of the executable behavior of a program and an algorithm for extracting information about actual program behavior from that representation. Formalisms such as formal semantics, finite automata, control flow graphs, and Petri nets have all served as the basis for such representations. With each of these well-studied formalisms comes a collection of existing algorithms for manipulating and analyzing a represented program. Developers of static analyses are still left with many decisions about how to encode the selected representation and associated algorithms and these decisions impact the cost and effectiveness of the analysis.

In this paper, we explore some of these decisions for state reachability analysis. The cost of reachability analysis is dominated by the fact that, in general, reachability graphs can grow to be exponential in the size of the source program [Tay83a]. Thus, great efforts are taken to reduce the size of the reachability graph. One approach that has been explored is to reduce the representation of each task in isolation before forming the reachability graph from these individual task representations. This was the approach taken in [LC89] using *task interaction graphs* (TIG)s, where the nodes in the task representation represent maximal sequential regions. Another approach is to form a model of the concurrent system that can be optimized or reduced before forming the reachability graph. This is the approach taken when the reachability graph is derived from a Petri net representation [Pet81]. There has been extensive research on Petri net reduction techniques, e.g., [Ber87, $ST^{+}94$]. We explore an approach that uses the TIG representation of each task to create a Petri net representation of the concurrent system. The resulting representation, called a *TIG-based Petri nets* (TPN)s, combines many of the benefits of TIGs and Petri nets. TPNs coarsen the view of program execution by collapsing the effects of large regions of sequential execution. In order not to lose important information, relevant analysis information is summarized and associated with each region. The result is a representation that is compact, but, unlike many other coarsening and reduction techniques, there is no loss of analysis information. Furthermore, TPNs and the reachability graphs generated from them can be reused across different analyses by simply rebinding region summary information. Thus, the resulting set of graphs provides a *property-independent, coarsened, multi-level* representation. This paper explores the potential benefits of using such a representation as a basis for state reachability analyses.

Property Independent Ideally, the program representation captures essential details of the program's execution behavior that allow an analysis algorithm to distinguish between executions that are guaranteed to satisfy the property that is being evaluated from those that may fail to satisfy the property. We call such a representation *conservative* with regard to the property of interest. 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 may deadlock from those that can not. Since building the program representation can be expensive, we want to maximize the kinds of properties for which the representation is conservative without excessively inflating the cost of the analysis or the cost of constructing the representation. The TPN representation that we present does not lose any information about the executable behavior of the program and thus is a conservative representation with respect to any property.

We envision that users employing conservative static analysis techniques will be interested in validating a number of properties of their programs. Thus, the benefits of re-use should be factored into any comparison of property-independent representations to representations optimized for a specific property.

Coarsened In general, the size of the representation will determine a lower bound on the cost of executing an analysis algorithm, since analyses over graph-based program representations typically perform some processing for each node or edge in the graph. One well-understood technique for reducing the size of the representation is the notion of *coarsening* the representation to increase the information modeled by each node or edge in the graph. This is precisely the approach taken in constructing basic-block control flow graphs, e.g., [ASU85]. Savings accrue from this approach by moving information from those parts of the representation on which complex algorithms operate to those parts of the representation on which simpler algorithms operate. For example, for many traditional data flow analysis problems, the computation of the summary information for a block is linear in the size of the block, whereas the propagation of this information through the flow graph is typically quadratic. Similarly for a TPN, shifting information to reduce the number of the reachability graph nodes and increasing the information modeled by each node should result in a net reduction in analysis cost when the computation of the summary information is less costly then the computation over the representation.

Multi-level When dealing with a coarsened representation, different analyses will require different kinds of summary information. Therefore, to support a variety of analyses, a representation should allow the binding of summary information to the nodes, or edges, of the graph to be light-weight, flexible and easily changed. We term such a representation *multi-level* since each level explicitly captures only a portion of the information necessary for the analysis. The main benefit of a multi-level representation is that construction of a high-level representation can be amortized across multiple analyses by rebinding its components to property-specific, lower-level representations. In practice, this can yield significant reduction in analysis cost, especially when construction of the high-level representation is expensive.

The benefits of coarsened, multi-level representations have been demonstrated in optimizing compilers for sequential languages. For example, the same basic block control flow graph can be used to compute reaching definitions or available expressions by binding the appropriate summary information to each node. The utility of such representations for the analysis of concurrent programs does not follow directly from the experience with sequential programs, because the complexities of the analyses are different. For state reachability analysis, where the size of the graph might be prohibitively large, we might consider more complex analysis algorithms in exchange for a significant reduction in graph size.

Thus, 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. We compare our results to recent work using an alternative Petri net representation for Ada programs. Our empirical data suggests that the TPN representation offers reduced analysis cost and enables reachability analysis for larger programs than previously proposed property-independent techniques. Building on our initial work [DCN95], we have also been able to adapt some property-specific Petri net reduction techniques into property-independent TPN reduction techniques, with the potential to further improve performance.

In the following section, we briefly overview the major approaches to concurrency analysis. Section 3 describes Petri nets, the TIG model, and reachability graphs. In Section 4, we describe how TIGs and Petri nets are combined into the TPN model. Section 5 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 6, we describe some property-independent Petri

net reductions and assess the potential for these techniques to improve the cost-effectiveness of TPN-based reachability analysis. Section 7 summarizes the contributions of this work and potential future directions.

2 Related Work

In this section, we give a broad overview of static concurrency analysis techniques and focus on efforts to empirically evaluate the relative cost-effectiveness of these methods.

State space enumeration methods consider each reachable program state to determine whether a program satisfies a given property [MR87, SMBT90, Tay83b, 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.

A variety of methods for reducing the cost of state space enumeration analyses have been explored. Valmari [Val91] and Godefroid and Wolper [GW91] have developed methods for determining when states are equivalent and avoiding the generation of redundant states. An alternate method of reducing the cost of reachability analysis for Petri net based analyses is termed *Petri net reductions.* Berthelot [Ber87] describes a variety of transformations that can be applied to Petri nets so as to reduce their size yet maintain certain properties. Duri et. al. [DBDS94] have demonstrated the potential of this approach to reduce the cost of analyzing deadlock freedom for selected programs. Furthermore, they demonstrate that Petri net reductions and other reduction methods, including those of Valamari and Godefroid and Wolper, can be combined to further decrease analysis cost. For some programs, state space reduction is able to decrease analysis cost considerably but, in the worst case, even with reductions the cost of state space enumeration remains exponential in 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 analyses whose solution provides information about the validity or satisfiability of those conditions by the program. Combining a variety of different sub-analyses provides a flexible method for improving the precision of the analysis results at a cost bounded by a low-order polynomial in the size of the program [DC94]. More experimental studies are needed to determine if such flow analysis techniques are capable of producing as precise results as more costly techniques over a broad range of programs.

Compositional approaches decompose the original analysis problem into smaller problems on which many of 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 for analyzing deadlock freedom. 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. Empirical evaluation of TPN-based reachability analysis and reduction techniques will add to this growing body of data. 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.

3 Background

This section presents Petri net, TIG and reachability graph models of concurrent programs. We introduce a simple example to illustrate the differences between the models presented in the 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.

3.1 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. Similarly, with each place is a set of *input transitions*, transitions at the head of incoming arcs, and *output transitions*, transitions 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. Thus, a Petri net, by successive firing of enabled transitions, generates a graph whose nodes are reachable markings and whose edges represent transition firings.

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

```
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

are present, blocks waiting for a caller on entry b. We note that execution of this program can deadlock if task T1 proceeds until it blocks at the accept of entry b and task T2 has not yet called entry a. This leaves the program in a deadlock state where T1 is blocked waiting for communication on entry b and T2 is blocked waiting for communication on entry a.

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, PTY95, SMBT90]. To illustrate, we consider a Petri net that explicitly represents the possible control flow branch and merge points in each program task. We refer to such a net as a *control flow graph Petri net* (CFGPN). Figure 2 illustrates a typical CFGPN for the example, where rendezvous start and end are 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 for task T1 in this example, the accept statement for entry a is a non-blocking accept statement since it will only be executed if there is a blocked entry call on a and, thus, is never blocked itself. The accept statement for entry b is a blocking accept statement since it will block if there is no waiting entry call on b. This distinction is reflected in the structure of the Petri net for 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.

3.2 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 analogous 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,* $E : N \to N$ *is the set of edges representing task interactions,* $S \in N$ *is the start node,* $T \subseteq N$ *is the set of terminal nodes,* $L : E \to \Sigma_{label}$ *is a function assigning labels to edges, and* C *is a function assigning code fragments to nodes.*

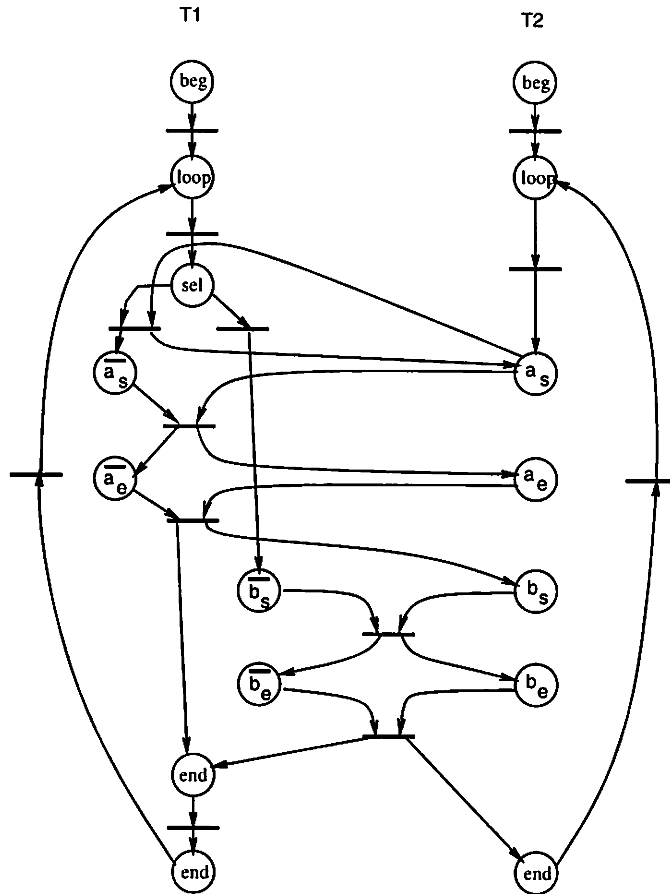

Figure 2: Control Flow Graph Petri Net for example

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.

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. 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

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

C(2) = ENTER(ā_s);
       EXIT(ā_e,4);

C(3) = ENTER(b̄_s);
       EXIT(b̄_e,5);

C(4) = loop
         select
           EXIT(ā_s,2);
           ENTER(ā_e);
         else
           EXIT(b̄_s,3);
         end select;
       end loop;

C(5) = loop
         select
           EXIT(ā_s,2);
         else
           EXIT(b̄_s,3);
           ENTER(b̄_e);
         end select;
       end loop;
```

Figure 3: Code fragments for task T1 of example

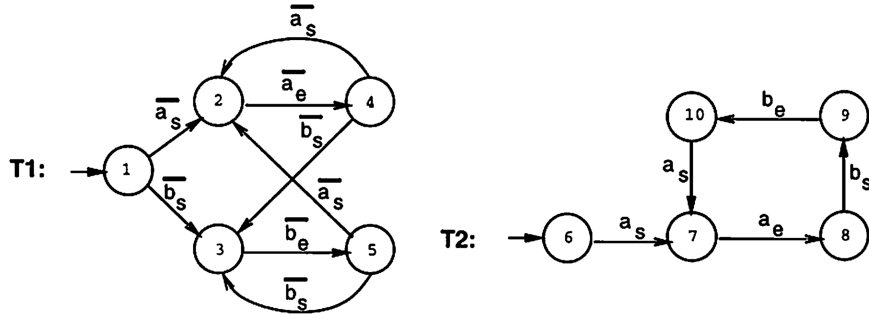

Figure 4: TIGs for example

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 $\overline{a_s}$.

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. Whenever possible, we show TIGs in this reduced form and drop the subscripts when referring to interaction names.

3.3 Reachability Graphs

Reachability graphs for concurrent systems are well-understood. They can be derived from a variety of different representations of the behavior of collections of individual tasks.

Definition 3 *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 $F \subseteq S$.*

For Petri net representations the reachability graph corresponds to the transitive closure of the Petri net firing rules over M_0, the initial marking. 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. Reachability graphs for Petri nets have been used to perform analysis of Ada tasking programs [MR87, SMBT90].

For a forest of TIG representations, the reachability graph is constructed by simulating the execution of each individual TIG and matching edges whose labels correspond to a call and accept of the same commu-

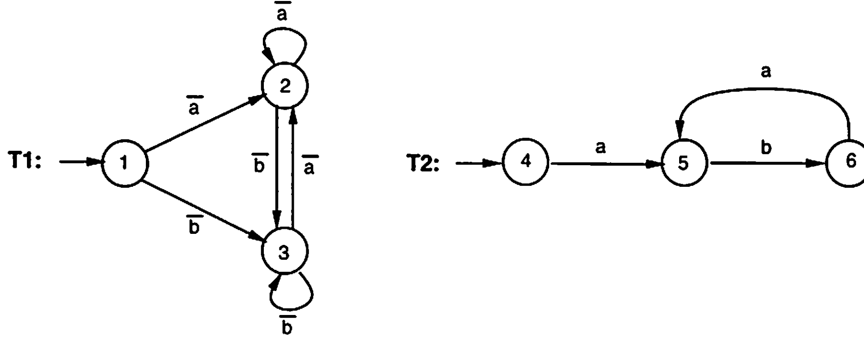

Figure 5: Reduced TIGs for example

nication channel.

4 TIG-based Petri nets

TIGs are a coarsened representation of the behavior of program tasks. They have been shown to enable significant reduction in the size of reachability graphs in practice [YTL+95]. TIG nodes correspond to regions of sequential program execution and are natural binding points for property-specific information summarizing the effects of a region. Thus, a forest of TIGs could be used to construct a coarsened, multi-level program representation for concurrency analysis.

Petri nets can also be used to represent the behavior of concurrent programs. Unlike a forest of TIGs a Petri net explicitly represents the interactions between tasks. We have combined the aggressive coarsening of TIGs and the explicit view of Petri nets to yield TIG-based Petri nets. Pezzé et. al. [PTY95] have demonstrated how one can construct a Petri net equivalent representation from a graph-based representation of concurrent programs. TPNs are not just *an* equivalent representation, however, they have been designed to be a property-independent, coarsened, multi-level representation for which we can easily associate property-specific summary information to facilitate a range of analyses. In addition, Petri net analysis techniques, such as accuracy improving techniques [CC96] and net reduction techniques, can be applied to TPNs. Some possible TPN reductions are discussed in Section 6.

Definition 4 *A **TIG-based Petri net** is a Petri net where the tuple, (P, T, F, M_0), is defined, based on a set of k TIGs, $(N_i, E_i, S_i, T_i, L_i, C_i)$, so that:*

$$\begin{aligned}
P &= \bigcup_{i=1}^{k} N_i \text{ is the union of all TIG regions} \\
T &\subseteq \bigcup_{i=1}^{k} E_i \times \bigcup_{j=1}^{k} E_j, j \neq i \\
&\quad \text{is a set of pairs } (x, y) \text{ of edges from different TIGs} \\
&\quad \text{where the labels } L(x) \text{ and } L(y) \text{ correspond to call and accept of the same entry,} \\
F &\subseteq (P \times T) \cup (T \times P) \\
&\quad \text{is the set of arcs linking places, that model TIG regions, with transitions representing} \\
&\quad \text{co-execution of pairs of TIG edges exiting and entering those regions} \\
M_0 &= (S_1, S_2, \ldots, S_k) \text{ is the initial marking representing the start nodes for each of the } k \text{ tasks}
\end{aligned}$$

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) $F = \emptyset$
(5) *for each edge,* $x \in \cup_{i=1}^{k} E_i \wedge L(x) = Call_a$ *loop*
(6)   *for each edge,* $y \in \cup_{i=1}^{k} E_i \wedge L(y) = Accept_a$ *loop*
(7) $T = T \cup (x, y)$
(8) $F = F \cup (Src(x), (x, y)) \cup (Src(y), (x, y)) \cup$
                $((x, y), Dest(x)) \cup ((x, y), Dest(y))$
    end loop
end loop

A TPN marking corresponds to a program termination state if all the marked places correspond to terminal TIG nodes. A similar algorithm developed independently by Pezze et. al. [PTY95] can be used to construct a Petri net from a set of task flow graphs.

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 execute. 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 5 is maximized when the total number of TIG edges is split evenly between calls and accepts of the same entry. Since each task has at least one node and each node has at least one incident edge, the cost of the loop on line 5 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 that interaction is the entry point. The number of task interactions in a TIG is linear in the number of communication statements in the task, since 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 3, results in additional TPN transitions. There are pathological examples where the

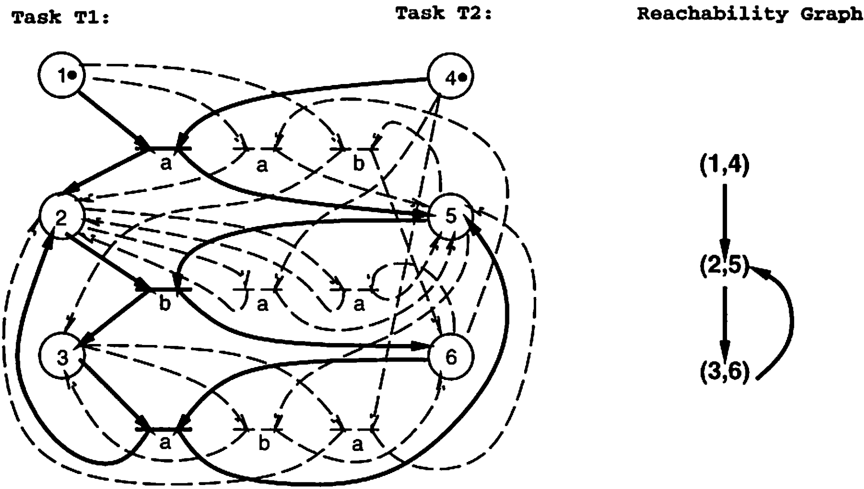

Figure 6: TIG-based Petri net and Reachability Graph for Example

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 potential task communication has a simple representation consisting of a single TPN transition with a pair of calling input places and a pair of accepting output places. Also when tracing executable behavior, each task will only have one place marked at a time. We have found that the regular structure of TPNs simplifies reasoning about the correctness of the TPN representation and TPN-based analysis[1].

TPN Place Summaries

For each region of code associated with a TIG region we can compute a property-specific summary of the behavior of that region. For example, when checking for freedom from deadlock, the summary would indicate the potential for execution to block within the region. As another example, when checking for freedom from critical races the summary would indicate which, if any, shared variables are accessed within the region. Given the one-to-one correspondence between TIG regions and TPN places, we can bind TIG region summary information to the TPN places. These summaries and their uses in analyses are described in detail in Section 5.

A Multi-level Representation

As with other Petri net representations, TPNs can be used to generate a reachability graph. TPN reachability graph nodes correspond to net markings, which in turn represent global execution states of the modeled program. A TPN marking is a collection of places, one for each program task. Given the one-to-one

[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.

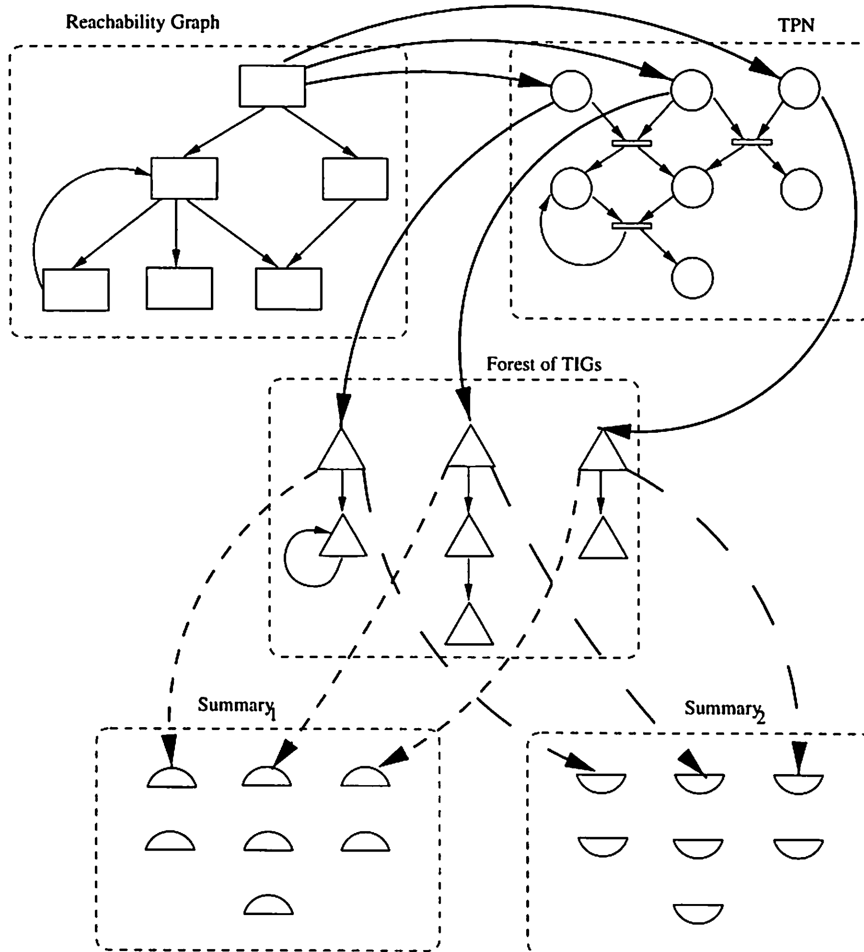

Figure 7: Levels of Representation in a TPN

correspondence between TPN places and TIG nodes we can map a node of the reachability graph to a collection of TIG nodes. As discussed above, TIG nodes could serve as binding points for property-specific summary information. The regions of code associated with TIG nodes are analyzed to extract property-specific summaries of the behavior of each region. The analysis algorithm can then combine the collection of summaries related to the TIG nodes associated with each reachable TPN marking to infer global properties of a potentially executable program state. Figure 7 illustrates how the information about a program's behavior is decomposed and spread across different levels of the TPN-based representation. By disassociating property-specific information from the reachability graph, the graph can be reused across a variety of analyses. Figure 7 illustrates two collections of summary information for use in distinct analyses.

In practice, a TIG region summary can be bound directly to its corresponding TPN place, thereby eliminating the necessity of traversing multiple levels of the representation during analysis. Furthermore, if modification of summaries is required, as described in Section 6, the modified summaries can be linked to TPN places without affecting the original region summaries. In the remainder of this paper we assume summary information is associated with TPN places.

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.

Designing a representation targeted for a specific analysis may allow improvement in the cost-effectiveness of that analysis. Those benefits, however, are restricted to that analysis and usually cannot be leveraged for other analyses. As will be shown in the next section, TPN-based reachability graphs enable significant reduction in analysis cost over uncoarsened representations. In addition, this benefit is not restricted to a single analysis. Furthermore, TPNs are designed to allow summary information in support of new analyses to be incorporated easily into reachability-based analyses.

5 Evaluating TPNs for Concurrency Analysis

Experimental evidence suggests that the 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 we can construct the reachability graph, then we at least have the opportunity to reason about desired program behaviors. Furthermore by employing a multi-level representation we can amortize the cost of constructing the reachability graph over a number of analyses. 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 an uncoarsened Petri net. We then discuss the cost of reasoning about properties of reachable markings of these net representations. 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. Before we present these results, we first discuss the methodology employed.

5.1 Methodology

We have collected 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, one for each program task. The CFGs are then converted to TIGs from which the TPN is constructed. 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. Although there has been some recent work on empirically evaluating static concurrency analysis techniques, 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 compare the cost of analysis techniques. 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.

To address these difficulties we compare the cost of performing reachability-based analyses using TPNs and an existing Petri net model. We separately compare the size of the generated reachability graphs and the cost of checking an individual graph node for the desired analysis. The first comparison is based on the number of nodes in reachability graphs generated from the two models for the same source programs. The second comparison is based on an analytical model of the cost of performing two different analyses over those reachability graphs. This allows us to focus on the essential work that is required to perform the analyses, rather than the details of toolset implementations.

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 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. TPNs and Ada-nets, and their reachability graphs, contain the same information about program behavior, although it is represented in quite different ways. Given this, we can compare the size of the models directly since neither is optimized for a specific analysis. Since the analysis algorithms traverse the reachability graphs and perform local checking on each node, we can also compare the cost of reasoning about reachable states. Ada-nets were originally conceived as a rich model of Ada tasking behavior and as such are not optimized for reachability analyses. We compare our results with equivalent analyses for Ada-nets because they are the only Petri net representation we are aware of for which a range of reachability analysis data have been published.

We have chosen to compare the performance of the analyses on standard examples from the literature. Analysis data based on Ada-nets is available for four example Ada tasking programs [DBDS94], `BDS`, versions of `Gas-1`, `Phils` and the `RW` examples. `BDS` is a simulation of a border defense system. 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. `Phils` are versions of the basic dining philosophers example with deadlock. `RW` are versions of the readers/writers problem. 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.

The degree to which the results of an empirical comparison of analysis costs are generalizable is highly dependent on the population of programs to which the analyses are applied. It may be possible to select programs on which one technique exceeds its "typical" performance and on which another technique falls short of its "typical" performance and, thus, unintentionally bias the results of the comparison. For the comparison in this paper, the lack of available data makes it impossible to judge the presence of bias since we do not understand the "typical" performance of TPN and Ada-net-based reachability analysis. Thus,

Table 1: TPN and Ada-net data

Example	Tasks	Ada-net Reachability Graph States	Ada-net Reachability Graph Arcs	TPN Reachability Graph States	TPN Reachability Graph Arcs
BDS	15	-	-	285006	1952588
Gas-1 3	7	72121	265578	493	987
Gas-1 5	9	-	-	9746	26785
Phils 3	6	18981	79371	84	186
Phils 5	10	-	-	1653	6130
Phils 7	14	*	*	32063	166502
RW 2/2	5	-	-	175	692
RW 2/3	6	-	-	609	3031
RW 3/2	6	-	-	579	2884
RW 5/2	8	-	-	5811	40660
RW 2/5	8	-	-	6229	43571

the results presented in this section are intriguing, but should not be generalized from this relatively small sample of programs.

5.2 Evaluating the Size of Reachability Graphs

Table 1 presents data on the size, in terms of the number of states and arcs, of reachability graphs generated from TPNs and from Ada-nets for the same source programs. 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.

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 the TPN-based analyses for the `Gas-1 5`, `Phils 5`, `Phils 7`, and `RW` 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. The use of TPNs yields smaller graphs and can produce reachability graphs for larger problems than are possible with Ada-nets.

5.3 Reasoning over Reachability Graphs

Analysis of TPN-based reachability graphs involves first computing summary information and then checking the intended behavior over the representation using the summarized information. As discussed in Section 4, for TPN-based reachability graphs, the summary information for a state is derived from summaries of the behavior of each code region associated with the marked places of the state. Analysis of different properties may require different kinds of summary information. Property-specific code region summaries bound to TPN

places provide the mechanism for associating information for different analyses to the same reachability graph nodes. This can yield significant savings in practice by amortizing the cost of constructing the reachability graph over multiple analyses.

Checking reachability properties of a program involves defining a *property predicate* that decides whether a TPN marking satisfies the property in question. For many kinds of analyses, this predicate is evaluated for each state of the reachability graph to determine if any reachable TPN marking violates the property; two such examples are shown below. Analyses that reason about properties of sequences of reachable markings require a slightly different approach, which for simplicity is not described here. Property predicates make use of the appropriate 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.

We illustrate the use of summary information and property predicates by checking whether a TPN marking corresponds to a program state that is free of critical races and by checking whether a TPN marking corresponds to a deadlock-free program state.

Analyzing for Freedom from 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. Since it is quite likely that a given TPN place will model a part of multiple reachability graph states we choose to pre-compute the summaries. As a practical matter, we can bind reachability graph states directly to a collection of TPN place summaries rather than having to traverse the multiple levels of the TPN representation to access the summaries. In the following, we have assumed that this summary information is pre-computed and bound directly to reachability graph nodes.

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. Note that *Summary* is the critical race information for each TPN place that is provided as input.

```
(1) WriteFound := (0, 0, ..., 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. In considering each of the places in a marking, upon first encountering a write to a variable that fact is recorded. Upon encountering a write to that variable in a second place, the algorithm indicates the potential for a critical race. Checking this property predicate for a given TPN marking requires $O(vk)$ steps since the critical race summaries are pre-computed.

To illustrate the checking performed by this property predicate consider the small example illustrated in Figure 8. The source code has been annotated to highlight the sequential code regions. In this example there are two shared variables x and y. Only variable x is referenced by the code and it is the first shared variable represented in the summary vectors. The TPN for this program is illustrated with the critical race summaries for each code region written next to its corresponding TPN place. The reachability graph illustrates the TPN places and race summaries that are processed by the above algorithm. In all cases, the predicate evaluates to false indicating that there is no executable program state in which a critical race occurs.

A slight variant of this algorithm would provide sufficient information to indicate both the variables involved in potential critical races and the statements in the source code in which the conflicting writes occur. The statements can be determined by scanning the regions of code associated with each TPN place in the marking that corresponds to the potential race. This provides the same level of error information as a statement-level Petri net representation.

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. 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. Once the summary information is computed, Algorithm 2 can be applied to an Ada-net marking. Thus, the cost of checking for the absence of a critical race at a reachable marking of either a TPN or Ada-net marking is comparable.

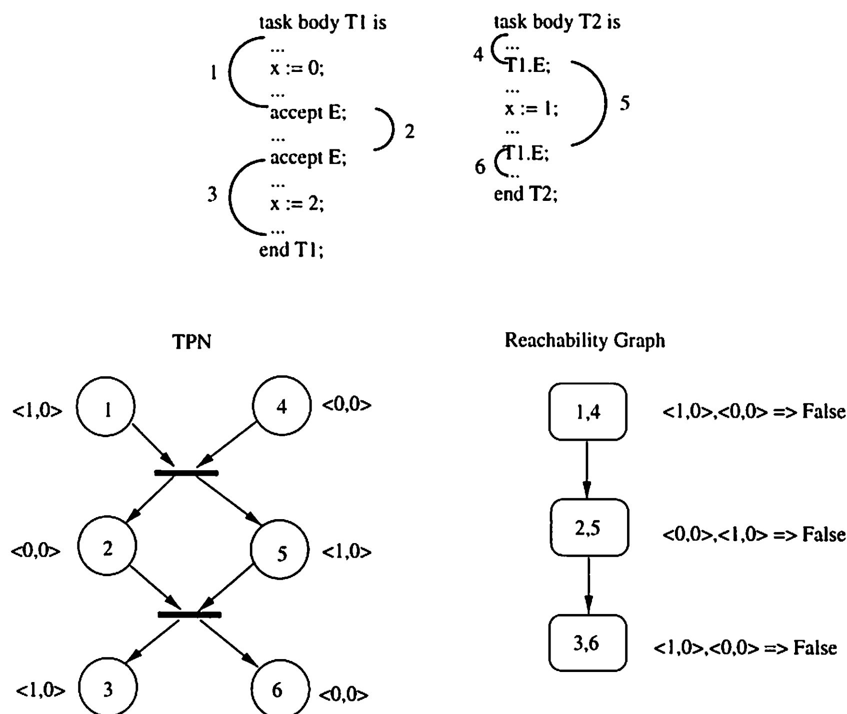

Figure 8: Using Critical Race Summaries

Analyzing for Freedom from Deadlock

Assuring that a program is free of deadlocks is one of the best studied concurrency analyses. As with other properties, checking for deadlock freedom requires processing summary information for reachable markings of a TPN.

Since we are interested in a conservative test that a marking is free of deadlock, a TPN place summary must include all of the exiting communications for the TIG region associated with that TPN place. We then must consider all possible combinations of exiting communications, where one communication statement is selected for each marked place. Each such combination must be examined to determine if there are no pairs of tasks that can successfully engage in communication.

In Ada, execution blocks when a task attempts to rendezvous, and thereby synchronize, with another task that is not ready to rendezvous. It is possible to execute a non-blocking communication in Ada by using selective entry calls or select statements with else or delay alternatives, as shown in Figure 1. In all other contexts communication statements, i.e., entry calls and accept statements, are blocking communications. Non-blocking communication statements cannot contribute to program deadlock. Furthermore, their execution cannot be guaranteed to release a blocking communication in another task since we can not be sure of the exact order in which the communications execute. Thus, the summary information for our deadlock freedom analysis represents only the set of blocking edges. This reduces the number of combinations of control flow choices that must be considered for checking deadlock-freedom. To illustrate, region 1 in Figure 5 has a non-blocking exiting edge, $1 \rightarrow 2$, representing the accept for a and a blocking exiting edge, $1 \rightarrow 3$, representing the accept for b. Thus, the TPN place summary for this region would indicate that there is a single blocking accept for entry b.

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 *represents the name of the entry associated with a blocking edge exiting the code region corresponding to place* s_i.

The choice combinations are easy to compute, but, in the worst-case, there can be many of them. If we make the standard assumptions that the number of branches of a control flow statement is bounded by some constant, c, the number of exiting edges from a region is $O(c)$. The total number of combinations of exiting TIG edges across all k of the regions associated with a TPN marking is therefore $O(c^k)$. [2]

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. The following summary information represents the matching communications that must be considered for each place during analysis:

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* j*th bit of the **call summary** vector indicates that the TPN place has a blocking communication that calls the* j*th task entry.*

An individual choice combination is used to filter only the portion of each place summary information that is relevant to a particular combination of intra-region control flow choices. We combine the choice combination and deadlock summary information to form a conservative test that a TPN marking is free of deadlock.

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 bit-vectors of length e, the number of entries in the program. Note that C is a vector whose elements describe the name of an entry. These choice combination elements can be also be considered as bit-vectors of length e.

(1) *if* M *is a terminal marking then*
(2) *return* $FALSE$
end if
(3) *for each choice combination,* C*, of* M *loop*
(4) $AllAccept = \bigcup_{i=1}^{k}(Accept(M[i]) \cap C[i])$
(5) $AllCall = \bigcup_{i=1}^{k}(Call(M[i]) \cap C[i])$

[2] For clarity our presentation is in terms of individual edges. In practice, we group together edges that are branches of the same select statement. These *edge groups* [YTL+95] are then used to compute choice combinations where each element of the combination may name multiple entries. This improves the efficiency of checking a TPN marking for deadlock freedom, although the bound remains unchanged.

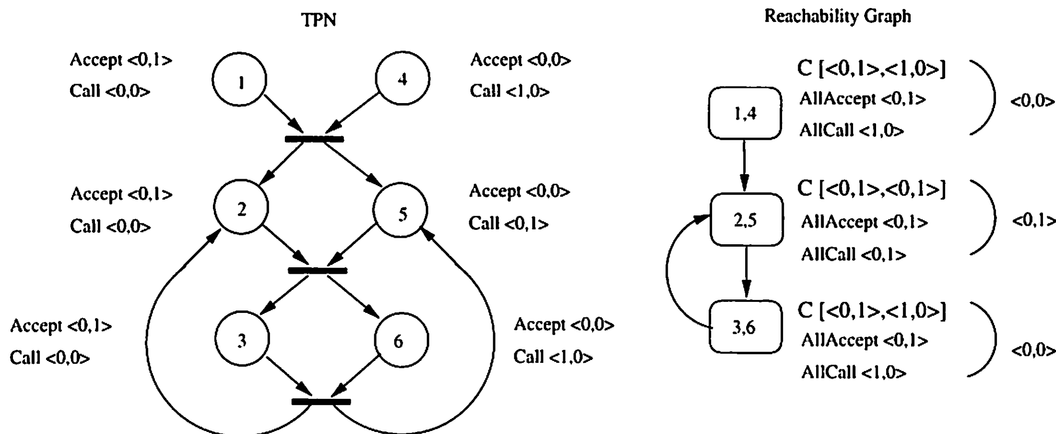

Figure 9: Using Deadlock Summaries

(6)   *if* $(AllCall \cap AllAccept) = (0, 0, \ldots, 0)$ *then*
(7)   *return* $TRUE$
     end if
   end loop
(8) ***return*** $FALSE$

To illustrate the checking performed by this property predicate consider the example from Figure 6 which is shown with choice combination and deadlock summary annotations in Figure 9. For this example there is a single choice combination for each marking, since each place in the marking has a single exiting blocking communication. The deadlock annotations represent the presence of blocking communications for entries a and b in the first and second bit respectively. The TPN for the program is illustrated with the deadlock summaries for each TIG region written next to its corresponding TPN place. The reachability graph illustrates the TPN places and the combined place summaries that are processed by the above algorithm. The results of this analysis indicate that states (1,4) and (3,6) correspond to potential deadlock states. In a conservative analysis we must assess whether these results are spurious or indicative of a program fault. For this example, there is an executable program behavior that can lead to deadlock states. If task T1 evaluates the select condition prior to task T2 executing the blocking call of entry a the program will deadlock as indicated by the analysis results.

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 state.

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.

Table 2: Deadlock Predicate Cost Data

Example	Tasks	Entries	Average CC	Predicate Cost Ratio	Graph Size 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 5/2	8	4	1	6.3	-

5.4 Evaluating Total Analysis Cost

We have presented TPN-based reachability graphs and compared them to reachability graphs built from the Ada-nets. We have also presented two property predicates for checking TPN markings and compared them to methods for checking the equivalent properties of Ada-net markings. To reason about program behavior we need to apply the property predicate, in the worst-case, to every reachable marking of the model. Therefore, the total cost of analysis is determined by the product of the number of reachability graph states and the cost of checking a property predicate at each state.

For checking that a program is free of critical races it appears that the TPN-based approach results in significant savings in the overall cost of analysis compared with Ada-net-based analysis. The cost of analyzing reachable markings of TPNs and Ada-nets is comparable. The size of the reachability graph, however, 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. Evaluating a deadlock property predicate for a TPN marking can be costly, whereas checking an Ada-net marking is relatively inexpensive. 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 marking and to check the equivalent condition at an Ada-net marking. We do this analytically based on Algorithm 3 and descriptions of checking for deadlock in Ada-nets [SMBT90]. 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.

The cost of checking an Ada-net marking involves accessing the current net marking, checking if there are any successor markings, and checking if the marking corresponds to a terminal state. Given our assumptions the cost of this check is:

$$PredicateCost_{Ada-net} = 3$$

Checking the deadlock predicate for a TPN marking involves accessing the current net marking, checking if the marking corresponds to a terminal state, and, if not, executing the loop from Algorithm 3. The cost of the loop dominates the predicate check time. The loop body requires bit-union operations over the call and accept summaries, filtered through the choice combination, for each task, and a final bit-intersection operation. The number of loop iterations is equal to the number of choice combinations for the marking. Thus, the total cost of checking the predicate is:

$$PredicateCost_{TPN} = 2 + CC(2 * T + 1)$$

where CC is the number of choice combinations for the marking and T is the number of tasks in the program.

In order to compute the cost of checking this predicate we need to know the number of choice combinations for each reachable marking in our example programs. This value is highly dependent on the communication and control flow structure of a program. For this reason, we have constructed a tool that 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 for the example programs in our study. We include the number of entries in order to validate our claim that the bit-vector operations are $O(1)$. The table includes the *predicate cost ratio* which is $PredicateCost_{TPN}/PredicateCost_{Ada-net}$. The table includes the *graph size 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 indicate that data was not available for a problem with a '*'. We do not include the smaller RW examples; for each of those examples, the predicate cost ratio was less than the ratio for RW 5/2 and the graph size ratio was undefined.

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.

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. For the cases where both predicate cost and graph size ratios are available we can compute the ratio of the total cost of TPN-based analysis to Ada-net-based analysis as 1 / (predicate cost ratio $*$ graph size ratio). In the cases where both Ada-net and TPN analysis techniques provide results, the TPN-based analyses enjoy a factor of 22 to 45 reduction in the cost of checking for deadlock freedom. When analyzing for critical race freedom TPN-based analysis yields a factor of 167 to 250 improvement over Ada-nets. Based on this limited data, TPNs appear to be an improvement over Ada-nets for the analyses we considered.

Summary

As discussed in Section 4, 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 vary widely in cost. 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 predicate on an uncoarsened Petri net marking, 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 uncoarsened net's reachability graph is too large to construct and the TPN reachability graph can be generated, TPN-based analysis is the better choice.

6 Property-Independent Net Reductions

The previous section compares the cost of analysis for property-independent representations. One could argue that it is not fair to compare TPNs to Ada-nets since recent work has focused on using Ada-nets as an intermediate representation on which to develop deadlock-specific Petri net reductions. Thus, the reader should view the results not as a reflection on Ada-nets but more as a point of comparison to Petri net analyses that are not reduced for a specific property. Recent data [DBDS94] show that when deadlock-freedom

reduction techniques are applied to Ada-nets the size of the reachability graph is reduced considerably. Such reductions enabled larger instances of the scalable problems we considered to be analyzed: `Gas 1-10`, `Phils 20`, and `RW 10/10`. Both the predicate cost and graph size for analysis based on reduced Ada-nets are less than for TPN-based analysis. Specifically, for the example programs analyzed, deadlock-freedom reduced Ada-nets yielded decreases in total analysis cost ranging from a factor of 14 to 529 over TPN-based analysis. Although this comparison ignores the cost of performing Ada-net reductions, we would still expect deadlock-freedom reduced Ada-nets to be substantially more efficient.

It is not surprising that a representation optimized for the analysis of a specific property would be significantly smaller than a property-independent representation, especially for those properties for which large portions of the executable behavior can be ignored. If we assume that analysts want to validate many properties about a program, then a property-independent representation is preferable if overall costs are comparable. One approach for decreasing overall cost is to develop property-independent reductions to decrease the size of the property-independent representation. Although there has been a considerable amount of work on defining deadlock-freedom Petri net reductions, there has been little work that we are aware of for defining property-independent reductions. In this section, we describe our formulation of such reductions. We illustrate the ideas by considering three property-independent reductions and describe how summary information for two different analyses can be preserved in the reduced nets. We present these reductions in terms of TPNs but believe they would also be applicable to other Petri net representations, such as Ada-nets. Towards this end, we indicate where differences in the underlying Petri net model will influence the formulation of property-independent reductions.

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. The goal is to produce a Petri net which generates a significantly smaller reachability graph. Individual reduction steps can be defined as the replacement of a sub-graph of a given Petri net with a new graph. The conditions under which a reduction step may be applied are typically defined by the presence of a sub-graph with certain structural properties. Individual reduction steps tend to be small and inexpensive to perform. A sequence of reduction steps is applied iteratively to reduce the structure of the original Petri net, until no more reductions are applicable.

The applicability of Petri net reductions relies on preserving information in the net that is relevant to the specific analyses that are being applied. This information can vary with each analysis as illustrated in Section 5 where we describe call and accept summaries for analyzing deadlock freedom and race summaries for analyzing critical race freedom using TPNs. The key idea that enables property-independent reductions is the combining of summary information from places in the unreduced net and binding the combined summary to places in the reduced net. Hence, property-independent reductions require the ability to adjust the binding of property-specific summary information to net places and, thus, assume support for a multi-level representation. In this section we discuss ways to combine summary information for analysis of two different properties.

In the remainder of this section we discuss three TPN reductions: *parallel transitions*, *serial places*, and *parallel places*. These are adaptations of existing Petri net reductions proposed by Berthelot [Ber87] and used by Shatz et. al. [ST+94] for Ada-net reductions. In defining these reductions we describe: the structural properties of the sub-graph of the Petri net, which determines the applicability of the reduction, and the transformation of that sub-graph. In describing sub-graphs we use the following notation: for a place p, $Out(p)$ is the set of output transitions, and $In(p)$ is the set of input transitions, for a transition t, $Out(t)$ is the set of output places, and $In(t)$ is the set of input places.

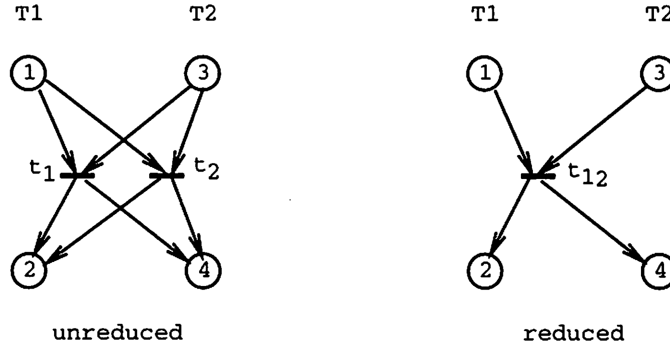

Figure 10: Example of parallel transitions reduction

6.1 Parallel Transitions

TPNs use transitions to represent the co-execution of inter-task communication statements in a pair of tasks. A transition may fire only when both its input places are marked. If a collection of transitions all share common input places and common output places, the firing of any of the transitions results in the same net marking. Since summary information is bound to net places, and not transitions, we can merge such a set of transitions. The resulting net is guaranteed to have the same reachable markings and furthermore the summary information available at those markings will be identical.

Formally, we define the reduction as follows:

Definition 8 *Parallel Transition Reduction*
Structure of sub-graph:

$\exists P_1, P_2 \subseteq P$	*two sets of boundary places*
$\quad \forall p, q \in P_1$	*upper boundary places share common outputs*
$\qquad Out(p) = Out(q)$	
$\quad \wedge$	
$\quad \forall p, q \in P_2$	*lower boundary places share common inputs*
$\qquad In(p) = In(q)$	
$\quad \wedge$	
$\quad \exists p \in P_1, q \in P_2$	*upper place's outputs and lower place's inputs*
$\qquad Out(p) = In(q)$	*are the same*

Transformation of sub-graph:

> *Replace all transitions bounded by the sets of places described above with a single transition whose input and output places are the same as any of those original transitions.*

Figure 10 illustrates the effect of this reduction for a pair of parallel transitions. 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 markings. In addition, it may simplify the net so as to expose opportunities for additional reductions.

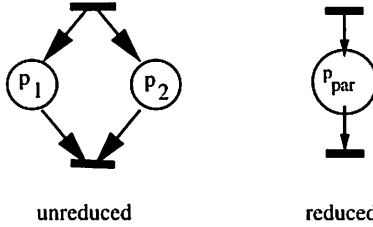

Figure 11: Example of parallel places reduction

6.2 Parallel Places

When program tasks engage in consecutive communications separated by regions of local computation, the TPN will contain pairs of places which share a single common input transition and a single common output transition. This reduction collapses such a pair of places into a single place, which will be marked whenever the original set of places would have been simultaneously marked. By merging the summary information for this pair of places and binding this new summary to the new place, we can preserve the information required by analyses.

Formally, we define the reduction as follows:

Definition 9 *Parallel Place Reduction*
Structure of sub-graph:

$\exists t_1, t_2 \subseteq T$	*two sets of boundary transitions*
$\quad In(t_1) = Out(t_2)$	*upper transition's outputs and lower transition's inputs are same*
$\quad \wedge$	
$\quad \forall p \in In(t_1)$	*all bounded places have only*
$\qquad In(p) = \{t_2\}$	*the designated boundary transitions*
$\qquad \wedge$	*as inputs and outputs*
$\qquad Out(p) = \{t_1\}$	

Transformation of sub-graph:

All bounded places are replaced with a single place whose input and output transitions are the designated boundary transitions.

Figure 11 illustrates the effect of this reduction for a pair of parallel places, p_1 and p_2, which are merged into a single place p_{par}. Thus, the reduction removes one place from the net and exposes opportunities for further reductions. To preserve analysis information in the net resulting from this reduction, we must combine the summary information from places p_1 and p_2 and bind it to place p_{par} [3]. We describe how this combination is performed on both of the analyses discussed Section 5.

For analyzing critical race freedom the race summaries for parallel places are unioned. This yields a conservative representation of the access to shared variables during execution of the sequence of TIG regions associated with places p_1 and p_2. Due to the fact that two tasks are involved in this pattern of execution it is possible that a critical race exists between these tasks. We check for such internal races by intersecting the critical race summaries for the places in the unreduced net. This allows us to detect that a potential

[3] Note that the correspondence between places of a reduced TPN and TIG regions is potentially different than for an unreduced TPN. In particular, the relationship between places and regions may be 1-many and a single place may correspond to regions from TIGs for different program tasks.

critical race may occur prior to reachability analysis. A more precise report of potential critical races would be delayed until it was determined that place p_{par} can actually appear in a reachable marking of the reduced net.

For analyzing freedom from deadlock, we can take advantage of information about the semantics of TPN places. This reduction requires that places p_1 and p_2 are the only outputs of a given transition. This implies that if one of the places is marked then the other must be marked, since the firing of the preceding transition will mark both. Furthermore, places p_1 and p_2 are the only input places of a transition. This pattern only occurs when places p_1 and p_2 correspond to a pair of TIG regions that each have a single exiting edge where those edges correspond to a call and accept of the same entry. Given that these regions are guaranteed to co-execute and form a matching communication pair, they cannot contribute to deadlock. Thus, the deadlock summary information would represent the fact that there are no blocking exiting calls or accepts for p_{par}. This not only preserves deadlock summary information in the reduced TPN, but also reduces the number of choice combinations that need to be considered in analyzing markings of the reduced net for deadlock freedom.

Note that for other Petri net representations, parallel places may represent both control flow choice between places of the same task and potential parallel execution among tasks. Thus, these two cases would need to be recognized in order to combine summary information appropriately during reduction. In addition, unlike TPNs, alternate Petri net representation may have more than two parallel places.

6.3 Serial Places

Application of a parallel place reduction may result in a net with a pair of places connected by a single transition. We can reduce such a pair of places to a single place to which we bind summary information combined from each of the original places.

Formally, we define the reduction as follows:

Definition 10 *Serial Place Reduction*
Structure of sub-graph:

$\exists p_1, p_2 \in P$	*two places*
$Out(p_1) = In(p_2)$	*where the outputs of one are the inputs of the other*
$\wedge$	
$\lvert In(p_2)\rvert = 1$	*and there is one such output/input transition*

Transformation of sub-graph:

The places and shared output/input transition are replaced with a single place.

Figure 12 illustrates the effect of this reduction. Places p_1 and p_2 are merged into a single place p_{ser}. Thus, the reduction removes one place and one transition from the net.

To preserve analysis information in the resulting net we must combine the summary information from places p_1 and p_2 and bind it to place p_{ser}.

For analyzing freedom from critical races, critical race summaries are combined by applying a bit-wise union. This yields a conservative representation of the access to shared variables during execution of the sequence of TIG regions associated with places p_1 and p_2. Furthermore, because of the restrictions on applying the reduction we can be sure that any place in the unreduced net that appears in a marking with p_1 (p_2) also appears in a marking with p_2 (p_1). In the reduced net, markings that only differ by having one of these places and not the other are collapsed to a single marking with p_{ser}. Thus, any critical races in the unreduced net will be reflected in the reduced net.

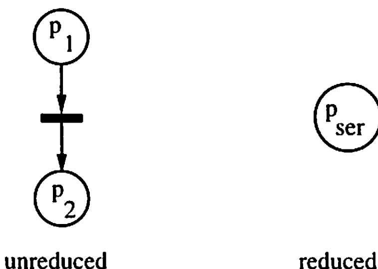

Figure 12: Example of serial places reduction

For analyzing deadlock freedom, the information about blocking edges exiting TIG regions represented in the call and accept summaries associated with places p_1 and p_2 is combined. If there is a blocking exiting communication on entry a associated with place p_1 and a blocking exiting communication on entry b associated with place p_2 then there are a pair of potential blocking exiting communications on entries a and b for the new place p_{ser}. This preserves the conservativeness of the reduced TPN for checking deadlock freedom since opportunities for causing deadlock by blocking communication are not removed from the representation.

6.4 An Example Reduction Sequence

As discussed above, reduction of a Petri net is performed by a series of reduction steps. We illustrate how the above reductions can be applied to produce a TPN that has a smaller reachability graph than the original TPN. We consider an example Ada tasking program, shown in Figure 13, that contains some simple client/server interactions. Each client task interacts with the server to acquire and then release access to locks managed by the server. The figure illustrates a sequence of parallel place (PP) and serial place (SP) reduction steps. The number of reachable markings is reduced from 5 for the original TPN to 3 for the final TPN. For this example, to perform analysis for critical race freedom we combine race summaries as described above for serial and parallel place reductions. The summaries are either 1, indicating a write to variable x, or a 0, indicating no write to variable x. For each reduction step we show the combined critical race summary for the new place. The final TPN has three reachable markings: $(\begin{smallmatrix}14\\25\end{smallmatrix}, 9)$, $(3, 6, 9)$ and $(3, \begin{smallmatrix}710\\811\end{smallmatrix})$. Since all of these markings have empty race summary intersections, as described in Algorithm 2, there are no executable critical races in this program. Thus, this sequence of reduction steps has yielded a more efficient analysis with the same precision as an unreduced analysis.

6.5 Discussion

A number of interesting observations can be made about our adaptation of Petri net reductions so as to make them property-independent.

In earlier work [DCN95] we describe a complex deadlock-freedom preserving TPN reduction called *forced communication pairs*. Using a series of serial place and parallel place reductions we can produce a property-independent version of the forced communication pair reduction.

The net reductions are intended to produce a net that generates a smaller reachability graph. This has the potential to reduce analysis cost since fewer states will need to be considered. For the TPN reductions shown here the cost of checking a deadlock-freedom or critical-race-freedom property predicate for markings

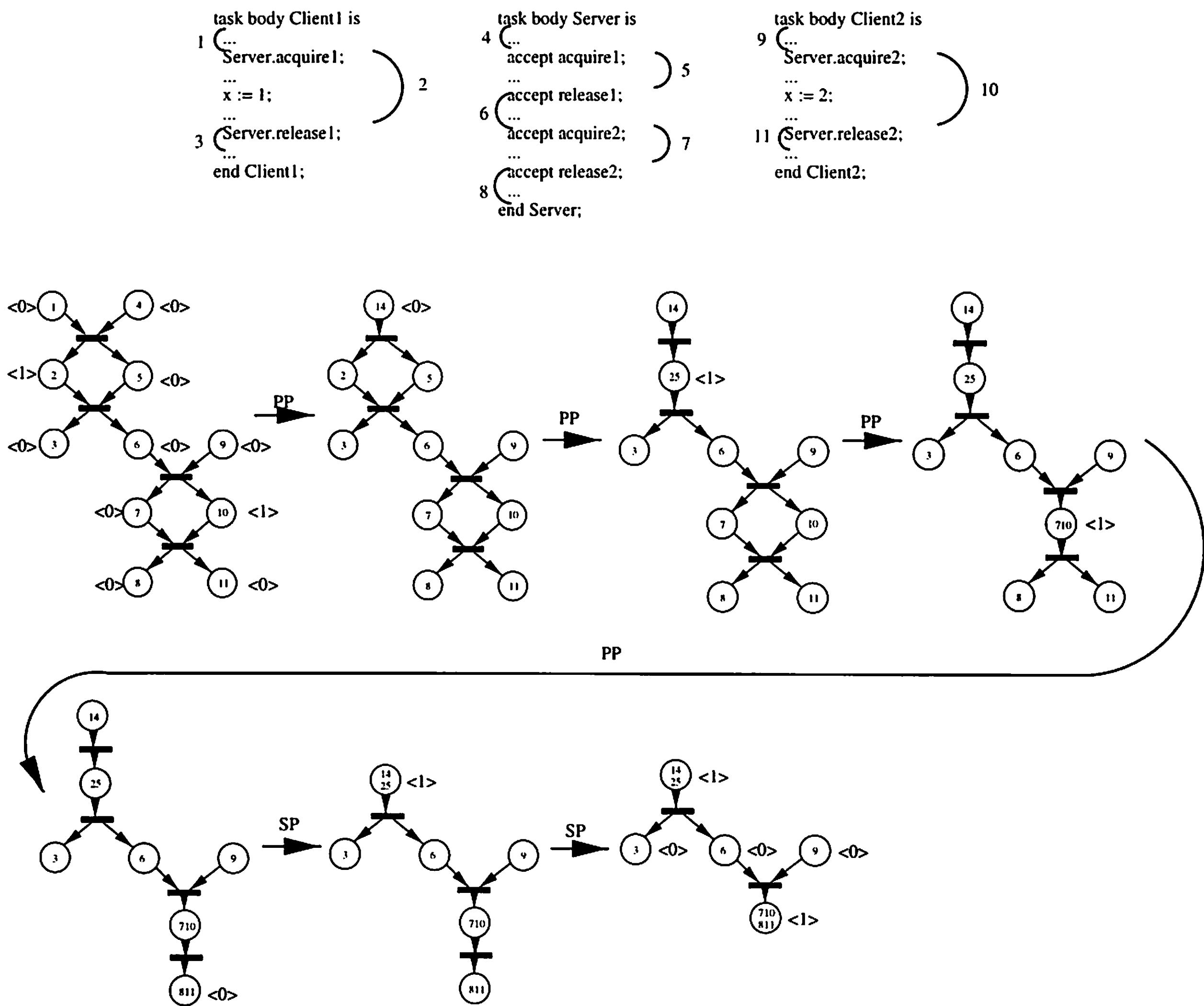

Figure 13: Example of a Sequence of Reduction Steps

of the reduced net will be no greater than for markings of the unreduced net. In fact, as illustrated by the deadlock summaries for the parallel place reduction, the property predicate check time can be reduced since combined summaries represent fewer blocking edges.

Many of the Petri net reductions defined by Shatz et. al. [ST+94] are specific to the structure of Ada-nets; they serve primarily to coarsen the standard Ada-net representation. Since the TPN is already coarsened by its construction from TIGs, those reductions will be neither applicable nor needed.

The cost of performing reductions is based on the the cost of detecting reduction opportunities, on the cost of applying each reduction, and on the number of reductions applied. For the TPN reductions described above, the cost of applying a reduction is a small constant factor more than applying an Ada-net reduction. Although both techniques perform similar net transformations, TPN reductions also compute the combined summary information, which for critical race and deadlock freedom analyses involved bit-vector operations.

Detection of reduction opportunities is based on the structure of the nets and the structure of the reduction patterns. The reductions described above are also considered in Ada-net reductions and, while there are few details available about the costs of detecting opportunities for individual reductions, we expect the costs for the TPN reductions to be comparable. The number of reductions applied promises to be significantly less for TPN reductions, however, since the Ada-specific reductions are not used. Overall, the cost of performing a series of TPN reductions appears to comparable, and less, than the cost of applying Ada-net reductions.

We have only described a small number of reductions and summary information combining functions. In all of these instances not only does the reduced net remain conservative, but there is no loss of precision in the analysis results. This may not always be the case. There may be more complex reductions and summary combining functions that yield conservative but imprecise information in the reduced net. These might still be desirable because of their impact on the size of the reachability graph, which is the primary limiting factor in reachability analysis.

Thus, with appropriate modifications many existing Petri net reductions can be adapted to TPNs. We have demonstrated how these reductions can be engineered so as to preserve different kinds of information in support of different analyses. Initial experimentation with reduction techniques on small academic examples indicates that the large number of dead transitions in TPNs, as illustrated in Figure 6, is a barrier to applying the simple reductions described in this section. We are planning to broaden our investigation to include additional reductions and analyses. In addition, we believe there is potential for applying property-independent reductions to alternative Petri net models, such as coarsened versions of Ada-nets.

7 Conclusion

In this paper we have presented a compact, flexible Petri net representation for Ada tasking programs. The TPN is an property-independent, coarsened, multi-level representation that allows different sources of property-specific information to be bound to reachability graph nodes. This allows the significant cost of constructing the reachability graph to be amortized over a number of different analyses. We introduced the concept of a property predicate and provide evidence that checking such predicates over the set of reachable TPN markings is practical for a collection of concurrent programs. Empirical evidence shows that significant reduction in the cost of reachability analyses can be obtained by using TPNs rather than unreduced Petri nets that explicitly represent program control flow.

TPNs have three advantages over flow graph models. First, a mature body of Petri net theory and analysis techniques can be applied to the analysis of TPN models. For example, in this paper we have described how Petri net reductions can be applied to TPNs. The reductions in this paper are described in terms of TPNs, but the concept of property-independent reductions based on combining sub-net summary information is more broadly applicable. Second, the existence of a common underlying model allows detailed comparison of TPN-based analyses to other Petri net analyses. Furthermore, this comparison can focus on the essential work required for analysis rather than the time to run an implementation of analysis tools. Third, recent work [CC96] has described techniques for encoding additional information in Petri net models so as to increase the precision of analysis results. This work can also be applied to TPN-based analysis.

It has been suggested that no single technique is suitable for analysis of all properties of all concurrent programs. TPNs combine the advantages of Petri net and TIG-based reachability analysis. Our hope is that this will enable TPN-based reachability analyses to capitalize on the benefits of and leverage off of new advances in concurrency analysis based on Petri nets and TIGs.

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, March 1991.

[ASU85] A.V. Aho, R. Sethi, and J.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. IEEE Computer Society Press, June 1990.

[Ber87] G. Berthelot. Checking properties of nets using transformations. In *Advances in Petri nets*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1987.

[CC96] A.T. Chamillard and L.A. Clarke. Improving the accuracy of petri net-based analysis of concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 24–38, January 1996.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the European Software Engineering Conference*, volume 717 of *Lecture Notes in Computer Science*, pages 283–300. Springer-Verlag, 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.

[DCN95] M.B. Dwyer, L.A. Clarke, and K.L. Nies. A compact petri net representation for concurrent programs. In *Proceedings of the 17th International Conference on Software Engineering*, pages 147–158, April 1995.

[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*, pages 36–48, October 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, 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*, pages 97–107, 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.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[PTY95] M. Pezzè, R.N. Taylor, and M. Young. Graph models for reachability analysis of concurrent programs. *ACM Transactions on Software Engineering and Methodology*, 4(2):171–213, April 1995.

[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.

[ST+94] S.M. Shatz, S. Tu, , T.Murata, and S. Duri. Theory and application of Petri net reduction for Ada tasking deadlock analysis. Technical report, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, 1994.

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

[Tay83b] 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.

[Val91] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, 1991.

[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, October 1991.