TIG-based Petri Nets for Modeling Ada Tasking

Matthew Dwyer*
Kari Forester†
Lori A. Clarke*

CMPSCI Technical Report 92–84
December 1992

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

† Department of Computer and Information Science
University of California
Irvine, California 92717

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grants # MDA972-91-J-1009 and #MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

TIG-based Petri Nets for Modeling Ada Tasking

Matthew Dwyer ⋆
Kari Forester†
Lori Clarke ⋆

⋆Department of Computer Science
University of Massachusetts, Amherst

†Dept. of Information & Computer Science
University of California, Irvine [1]

Abstract

This paper presents a Petri net model for concurrent programs that is based on a representation of concurrent programs called *task interaction graphs* (TIGs). The size of these TIG-based Petri nets and the applicability of existing Petri net analysis techniques to them is discussed. A comparison of TIG-based Petri nets to an existing Petri net model of concurrent programs is also included.

1 Introduction

Analyzing concurrent software is a complex and difficult task. Asynchronously running processes produce an exponential number of potential interleavings that are hard to analyze with static methods. Erroneous patterns of communication can produce anomalies, such as deadlock or race conditions, that can be difficult to discover or reproduce. Analysis and verification of concurrent programs is an active area of research. Many different approaches to static analysis of concurrent programs exist, including Petri nets [MR87, MSS89, SMBT90], state based models [Tay83b, LC89, YTFB89, McD89, YY90], data flow [TO80, DS91, LC91], and constrained expressions [ADWR86, ABC+91]. There has also been research on dynamic analysis of concurrent programs [HL85, TK86, Tai85] and on the formal specification and verification of concurrent programs [OG76, Lam83, Dil90].

In this paper, we look at two models proposed as the basis for static analysis of concurrent programs that use rendezvous style communication: a well developed Petri net model and a recently proposed model called TIGs. We propose a Petri net model based on the TIG representation that exhibits the same reachability graph compaction as the TIG representation. We evaluate the size of this model and its potential as a basis for Petri net based static analysis.

Petri nets [Pet81] are a well studied and widely applied formalism for modeling systems. Murata classifies analysis methods for Petri nets into three groups: reachability tree method, matrix-equation approach, and reduction or decomposition techniques [Mur89]. In [MR87, SC88, MZGT85]

[1]This work was supported by the Defense Advanced Research Projects Agency under Grant MDA972-91-J-1009.

Petri nets have been used to model and analyze concurrent programs that use rendezvous style communication. The reachability graph of such a Petri net has been used to perform analysis of Ada tasking programs [SC88, MR87]. More recently, the matrix-equation approach has been used to detect a class of deadlocks in Ada tasking programs [MSS89] and in [TST] a set of reduction rules for Petri nets are given that preserve the properties necessary for detection of deadlock in Ada tasking programs.

Another approach to modeling concurrent programs that use rendezvous style communication was introduced by Taylor [Tay83b]. In this and subsequent work [LC89, YTFB89], tasks are modeled as individual flow graphs. Task flowgraphs can be combined to generate control flow concurrency graphs. These concurrency graphs contain all of the possible concurrency states of the system and are analogous to the reachability graph derived from Petri nets. Long and Clarke have proposed a model for representing tasking programs that hides control flow information called TIGs[LC89]. TIGs can be used to generate more compact reachability graphs.

In the following section we give a brief overview of Petri nets and TIGs. Section 3 shows how a TIG-based Petri net is constructed and discusses the applicability of existing Petri net analysis techniques to TIG-based Petri nets. Section 4 analyzes the size of TIG-based Petri nets. It compares the results of this analysis to an existing Petri net model of tasking programs. Finally it discusses the implications of the results with respect to the effectiveness of existing Petri net analysis techniques. Section 5 mentions directions for future work.

2 Overview

This section defines general Petri net and TIG terminology. Reachability analysis is described in general terms and applied to both Petri net and TIG representations. Finally we introduce a simple example to illustrate the concepts presented in the paper.

Petri Nets

Petri nets are a graphical formalism used for specifying concurrent systems. A Petri net is a directed bipartite graph with node called *places* and *transitions*. Typically, places are drawn as circles and transitions as bars. The edges of the graph are called *arcs*. Arcs are labeled with a positive integer representing their *weight*. A *marking* is an assignment of an integer to each place in the net and represents the number of *tokens* at that place. Tokens are drawn as black dots inside of places. A marking is given by a k-vector, M, where k is the number of places in the net and $M(i)$ denotes the number of tokens at place i. Formally, a Petri net is a tuple (P,T,F,W,M_0), where P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, W is a function assigning weights to arcs, and M_0 is the initial net marking. Associated with each transition is a set of *input places*, places at the head of incoming arcs, and *output places*, places at the tail of outgoing arcs. A transition is *enabled* if each input place of the transition is marked with at least as many tokens as the weight given on the associated input arc. A transition *fires* by removing W(p_i,t) token from each input place p_i and adding W(t,p_o) tokens to each output place p_o. A transition may not fire unless it is enabled.

In this paper all of the Petri nets discussed are *ordinary*, having arc weights of 1, and *safe*, having a maximum of 1 token per place.

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

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

Figure 1: Ada tasking example

Task Interaction Graphs

TIGs have been proposed by Long and Clarke [LC89] as a compact representation for tasks that is amenable to analysis. The TIG abstraction hides control flow information internal to a task thereby facilitating concurrency analysis. TIGs divide tasks into maximal sequential regions, where such task regions define all of the possible behaviors between two consecutive task interactions. A task interaction is any point where the behavior of one task can be influenced by the behavior of another task. Formally, a TIG is a tuple (N, E, S, T, L, C), where N is the set of nodes representing task regions, E is the set of edges representing task interactions, S is the start node, T is the set of terminal nodes, L is a function assigning labels to edges, and C is a function assigning *pseudocode* to nodes. The start node represents the region where task execution begins and the terminal nodes represent regions where task execution potentially ends. Each node has a fragment of *pseudocode* associated with it that represents the code in that task region, which in our examples are Ada statements, plus two non-executable statements, ENTER and EXIT, that mark region entry and exit points. The edges of a TIG are labeled with the tasking interactions that cause transitions from one region to another. The tasking interactions we consider are Ada entry calls and accept statements. There are four distinct kinds of tasking interactions: starting an entry call, ending an entry call, starting an accept statement, and ending an accept statement. It is necessary to model both the start and end of a rendezvous explicitly because accept bodies are themselves task regions that perform sequential computation that must be captured in the representation.

Figure 1 presents a simple Ada program that will be used as an example throughout the rest of the paper. Task T1 consists of 5 sequential regions. To illustrate the idea of maximal sequential regions consider the initial region of T1. Region 1 enters at the beginning of the task and exits at the select statement. There are two exits out of this region. The first exit is on the start of the accept for E1 and the second is on the start of the accept for E2. The pseudocode for task T1 is given in figure 2. The TIGs for tasks T1 and T2 are given in figure 3. Given that regions represent all sequential execution paths between pairs of consecutive tasking interactions, it is possible for distinct TIG nodes to contain pseudocode for the same program statements. When EXIT pseudocode statements are duplicated in this way, multiple TIG edges may be used to represent a single Ada communication statement in the source program. This is illustrated by the duplication of the statement `EXIT(ACCEPT_START(E1),2)` in regions 1, 4 and 5 of task T1 in

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

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

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

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

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

Figure 2: Pseudocode for task T1 of example

figure 3. Note that a TIG represents a single task instance. The potential behaviors of a collection of tasks can be modeled by matching edges from different TIGs, whose labels represent call and accept requests for the same task entry.

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

Reachability

Many static analysis techniques rely on searching a program's state *reachability graph* for properties of interest. Any state based representation of concurrent programs is amenable to this analysis. To accomplish this we require a definition of the set of potential system states, the state successor function, and the initial system state. The reachability graph is the transitive closure of the successor function applied to the initial system state.

For Petri nets the set of states is the set of net markings reachable from the initial marking M_0. The state successor function is defined for each marking by the set of enabled transitions and the markings produced by firing each individually. A Petri net transition is *dead* if there is no reachable marking for which it can be enabled.

For a concurrent program modeled as a set of m TIGs where $(N_i, E_i, S_i, T_i, L_i, C_i)$ is the ith TIG, the set of potential states is a set of m-tuples $(c_1, c_2, \ldots, c_m)$, where $c_i \in N_i$. The initial state is the m-tuple $(S_1, S_2, \ldots, S_m)$, of start nodes for each TIG. The state successor function is defined such

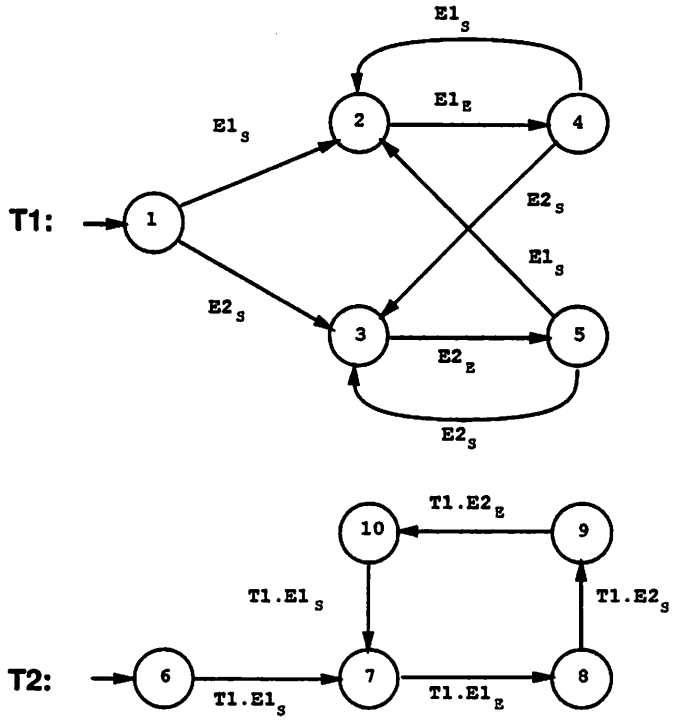

Figure 3: TIGs for example

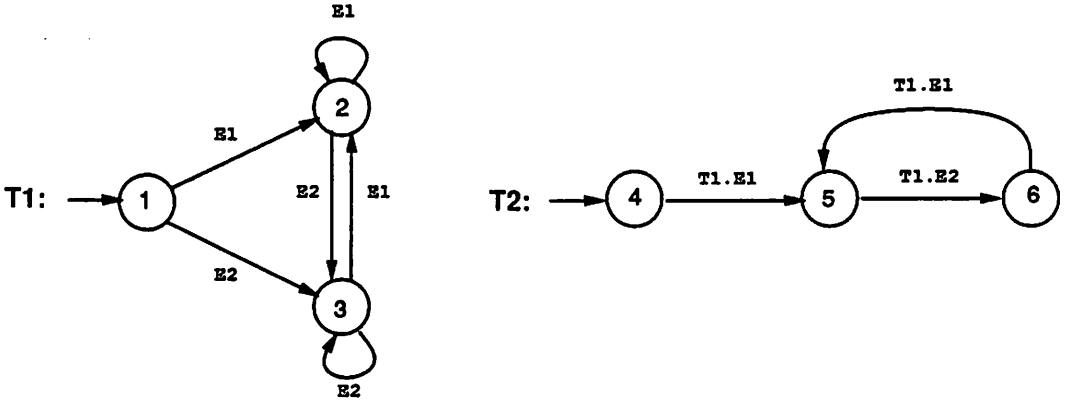

Figure 4: Reduced TIGs for example

that $(next_1, next_2, ...next_m)$ is a successor of a state $(c_1, c_2, ...c_m)$ if and only if $\exists i$ and j such that $((c_i, next_i) \in E_i$ and $(c_j, next_j) \in E_j$ and $L(c_i, next_i)$ and $L(c_j, next_j)$ represent a potential task interaction) and $\forall k \neq i$ or j, $(c_k = next_k)$. As an example, the reachability graph constructed from the set of TIGs in figure 4 is illustrated in figure 5. Although often manageable in size in practice, Taylor [Tay83a] demonstrated that, in general, the size of the reachability graph is exponential in the number of tasks in the program.

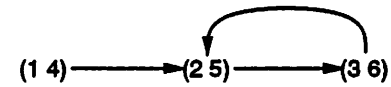

Figure 5: Reachability graph from Reduced TIGs and TPN for example

3 TIG-based Petri net model

We propose a Petri net model for Ada tasking programs that hides the details of task control flow. These *TIG-based Petri nets* (TPN) are constructed from a set of TIGs. A TPN maintains a strong relationship with the set of TIGs; each place in the Petri net has a one-to-one correspondence with a tasking region and each transition represents a potential task interaction. TPNs can be constructed quite simply by creating a transition for each pair of TIG edges whose labels represent a call and accept on the same task entry and by making the source and destination regions of these edges the input and output places of the new transition. More precisely:
Input: $Set = (Tig_1, Tig_2, \ldots, Tig_k)$ a set of TIGs
Output: (P, T, F, W, M_0) is a Petri-net
Algorithm:

$\forall Tig_i \in Set$
  $\forall n \in N_i$
    $P = P \cup$ `create-place`(n)

$\forall e_i = (n_i^{head}, n_i^{tail}), e_j = (n_j^{head}, n_j^{tail})$ where $e_i \in E_n \wedge e_j \in E_m \wedge n \neq m \wedge$ `match`$(L_n(e_i), L_m(e_j))$
  $T_{new} =$ `create-transition`
  $T = T \cup T_{new}$
  $F = F \cup ($`place`$(n_i^{head}), T_{new})$
  $F = F \cup ($`place`$(n_j^{head}), T_{new})$
  $F = F \cup (T_{new},$`place`$(n_i^{tail}))$
  $F = F \cup (T_{new},$`place`$(n_j^{tail}))$

$W = 1$

$M_0 =$ `create-marking(place(`S^1`),place(`S^2`),...,place(`S^k`))`

where `match(label,label)` is true if label represents a call and an accept of the same task entry, `create-place(node)` creates a unique place for a given TIG node value, `place(node)` returns place for a given TIG node value, `create-transition` creates a unique transition, and `create-marking(place-list)` creates a marking with 1 token at each place appearing in the argument list and 0 tokens at all other places. We note that Pezzi, Taylor and Young independently developed [PTY] a similar algorithm for constructing Petri nets from labeled flow graphs.

This algorithm constructs a Petri net that overestimates the possible task interactions of the program. All potential task interactions are included as a result of the simple and efficient matching of TIG edge labels, but some of these interactions can never be executed. In section 5 we discuss an approach for eliminating the corresponding dead transitions. It should be noted that building a Petri net with transitions for only the executable task interactions is equivalent to construction

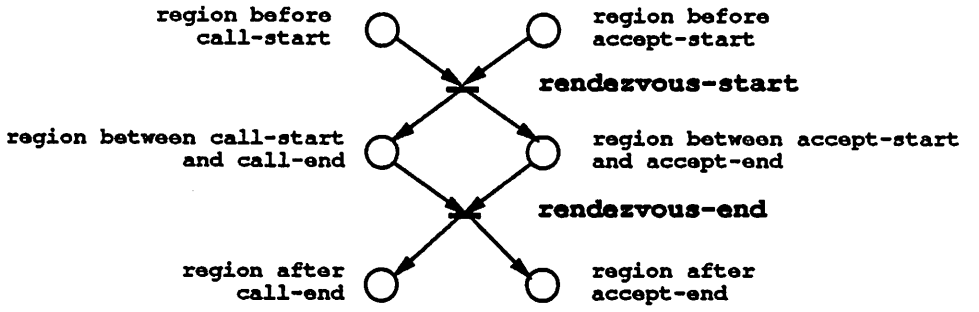

Figure 6: TIG-based Petri net representation for Ada rendezvous

of the reachability graph and therefor is intractable in general.

Figure 6 illustrates the Petri net fragment that represents a single Ada rendezvous between a calling and accepting task. In the case of multiple callers this fragment is replicated with the accepting task participating in all potential rendezvous. The resulting TPNs are ordinary, as $W = 1$, and safe, as M_0 has only values of 0 or 1, and for all transitions t, |input places of t| = |output places of t| = 2, so the number of tokens is preserved across transitions. As such many existing Petri net analysis techniques apply to TPNs, for example reachability, invariant methods, and structural reductions [Mur89]. Reachability graphs for TPNs are isomorphic to the reachability graph generated from the set of TIGs to which the TPN corresponds. We note that the reduction applied to TIGs, in the case of a bodyless accept, is also applicable to TPNs. In fact this reduction is equivalent to applying the *FPP* and *FST* structural reductions discussed in [Mur89].

Continuing with our example, figure 7 illustrates the TPN constructed from the reduced TIGs in figure 4 where the executable transitions and arcs are in bold. The reachability graph for this TPN is given in figure 5.

4 Analysis

For a Petri net we are interested in the number of places and transitions as a measure of the size of the representation. We consider two different analytical approaches in this section. First, we argue that TPNs will have few places for all Ada tasking programs by showing an upper bound on the number of places in a TPN. Then, we reason about the number of transition in a TPN by considering skeletal Ada tasking programs that represent common communication patterns. We discuss the implications of these results for applying existing Petri net analysis techniques to TPNs. Finally, we compare our results to an existing Petri net model of Ada tasking programs.

Places

As discussed in section 3 each place in a TPN corresponds to a single node of one of the TIGs that represents a task in the program. The number of places in a TPN is independent of the number of potential task interactions. The total number of places in a TPN is the sum of the number of nodes of the TIGs representing the program. A single Ada task may have a number of entry calls and accept statements, these may be either a synchronizing rendezvous, where an accept has no body, or a remote procedure call (RPC), where an accept has a body. Consider a task with c_{synch}

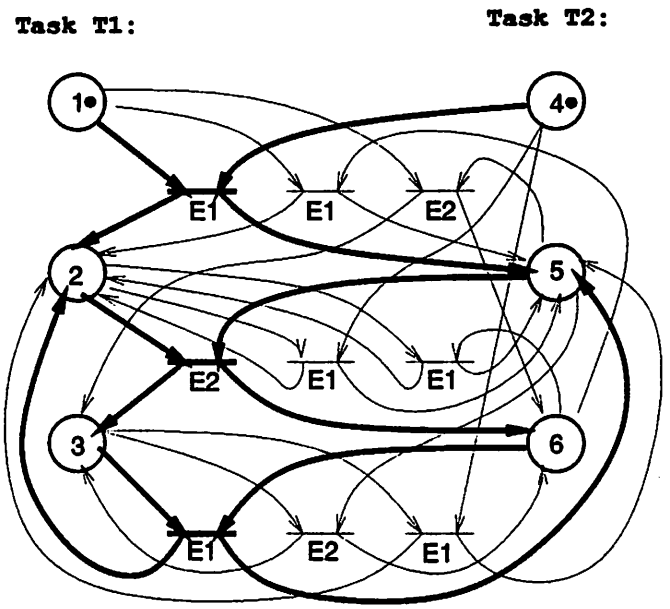

Figure 7: TIG-based Petri net for example

synchronous entry calls, c_{rpc} RPC entry calls, a_{synch} synchronous accept statements and a_{rpc} RPC accept statements. The TIG representing such a task has $\leq 2(c_{rpc}) + 2(a_{rpc}) + c_{synch} + a_{synch} + 1$ nodes. For each RPC rendezvous the start and end interactions are modeled separately, while for each synchronizing rendezvous the start and end interactions are collapsed to a single interaction in the TIG model. Each modeled interaction produces at most 1 TIG region, the region for which it is the entry interaction. The additional node is to model the initial region of the task. We note that this is a strong upper bound. Furthermore this bound demonstrates that the number of TPN places is linear in the number of communication statements in the program.

We mentioned that TPNs are ordinary and safe. In fact they have a well defined structure that allows us to reason about the size of the reachability graphs they generate. The reachable markings of a TPN are restricted such that the sum of the tokens in the places that correspond to regions of a single task, which we call a *task sub-net* (TSN), is 1. This single token per TSN indicates the current execution state of the task, i.e., the current task region. A TSN place hides task state information that might otherwise be represented explicitly in the net. Not surprisingly, TPNs generate smaller reachability graphs than Petri nets with similar structure that explicitly represent more task details.

Petri net reduction techniques attempt to recognize patterns in a Petri net and replace the pattern with a Petri net fragment that has fewer places [TST]. These reductions are not information preserving; they are designed to retain certain liveness properties in the reduced net that are sufficient to detect deadlock. If we wish to only check for deadlock in a program modeled as a TPN we can apply these reduction techniques as well.

Transitions

In TPNs transitions are used to model potential task interactions. In general the number of potential interactions in an Ada tasking program is a function of the number of entry calls in a calling task and of accept statements in the accepting task for each task entry. Such a general characterization of the number of potential interactions does not provide any information on the number of potential interactions in realistic Ada tasking programs. We would like to analyze TPNs with respect to a set of skeletal Ada tasking programs that reflect common communications patterns found in realistic programs. As a first attempt we consider programs that have appeared in the concurrency analysis literature [SC88, SMBT90, YTFB89, HL85]. We identify three common patterns of communication: sequence, iteration and choice, and consider some compositions of these patterns. Figure 8 gives skeletal Ada programs for the patterns considered in this section. Note that each pattern can appear in a calling or accepting task, but the syntax may be different, e.g., calling choice is an if-elsif-...-else while accepting choice is a select.

```
task body Choice is
begin
    select
        accept E1;
    or
        accept E2;
    .
    .
    .
    or
        accept En;
    end select;
end Choice;
```

```
task body Sequence is
begin
    T1.E1;
    T1.E2;
    .
    .
    .
    T1.En;
end Sequence;
```

```
task body Choice-Iter is
begin
    loop
        select
            accept E1;
        or
            accept E2;
        .
        .
        .
        or
            accept En;
        end select;
    end loop;
end Choice-Iter;
```

```
task body Seq-Iter is
begin
    loop
        T1.E1;
        T1.E2;
        .
        .
        .
        T1.En;
    end loop;
end Seq-iter;
```

Figure 8: Ada tasking programs for communication patterns

	seq	choice	seq-iter	choice-iter
seq	$2n$			
choice	$2n$	$2n$		
seq-iter	$2n+1$	$2n+1$	$2n+3$	
choice-iter	n^2+2n	n^2+2n	n^2+3n+1	n^3+2n^2+2n

Table 1: Number of TPN transitions for communication patterns

Our initial analysis considers pairs of tasks, one calling and one accepting task. The calling task has a single call to each of the n entries in the accepting task. The accepting task has a single accept statement for each of its entries. The organization of these calls and accepts varies according to the communication pattern being considered. Table 1 gives the number of TPN transitions created from the various combinations of communication patterns, where rows are the calling task, columns are the accepting task, and the table is symmetric along the diagonal. Here we see that the number of transitions becomes non-linear in n when at least one the of the tasks has a choice within a loop. Most of these transitions, except in the case of 2 choice-iteration tasks, are dead. They are created by the duplication of a task interaction on the exit edge of the initial task region in the TIG representation, and carried over into the TPN. Choice-iteration tasks with n branches are modeled by a TIG where each of the n interactions on the branches is the entry to a region and each of these regions has n exits which enter into these same regions; this models the end of an iteration and the choice made on the next iteration. This duplication of entry and accept requests as TIG edges is responsible for the large number of TPN transitions.

We realize the limitations of this analysis, e.g., skeletons eliminate variables that control program flow, chosen patterns may not fully represent those found in real programs, lack of multiple callers, lack of multiple call/accepts for a single task entry, real programs may not scale in the number of select branches. In spite of these limitations we feel the analysis produces a useful characterization of the number of TPN transitions for a set of communication patterns that do appear in practice. The impact of the number of TPN transitions on the cost of analysis varies with the Petri net analysis technique.

The size of a resulting reachability graph is unaffected by the presence of dead TPN transitions. The existence of a large number of live transitions in a TPN does not increase the number of nodes in the reachability graph. It will increase the number of edges, which is bounded by the square of the number of nodes. In practice the large number of live transitions tends to increase the cost of generating the reachability graph.

As mentioned above, the reductions of [TST] are designed to reduce the number of places rather than transitions. In fact there is one suggested reduction rule that increases the number of live transitions in the reduced net. Note that for structural reductions Petri net patterns may match live or dead transitions. It is not clear at present how the presence of additional transitions affects the applicability of these structural reduction rules.

By definition [Mur89] dead transitions cannot contribute to the number of transition invariants of a Petri net, whereas the presence of large numbers of live transitions will increase the number of transition invariants. Transition invariants have only been applied to the analysis of non-cyclic tasks [MSS89], however, it is an open question as to how this analysis can be applied to tasks with

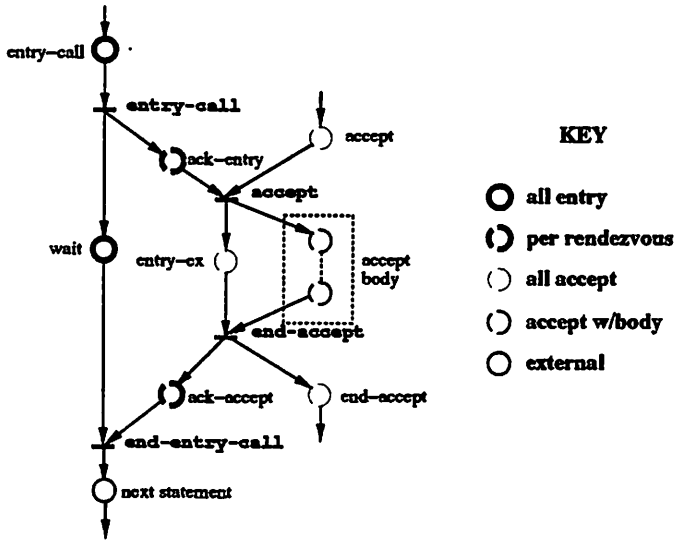

Figure 9: Ada-net representation of rendezvous

the communication patterns we consider above.

Comparison

As a point of comparison we consider an existing Petri net model for Ada tasking , called *Ada-nets*, used in Shatz's toolkit for static analysis of tasking behavior [SC88]. The representation for a single Ada rendezvous between a calling and accepting task in this Petri net model is illustrated in figure 9. The key describes 5 different classes of places: **all entry** places are used for modeling entry calls, **all accept** places are used for modeling accept statements, if the accept statement has a body the **accept w/body** places are also required, **per rendezvous** places are used for modeling all potential rendezvous, the **external** place represents the representation of the program statement following the entry call. The fundamental difference between Ada-nets and TPNs is that Ada-nets model concurrent programs by explicitly representing potential tasking interactions, control flow information, and detailed Ada tasking semantics, whereas TPNs hide control flow information and do not explicitly model detailed Ada tasking semantics.

Ada-nets are constructed by connecting Petri sub-nets that represent each individual task in the program with Petri net fragments that represent potential task interactions [SC88]. The sub-nets that represent individual tasks are called *process sub-nets* (PSN) and they include places and transitions that explicitly model program control flow constructs that contain task interactions.

For comparison with TPNs we develop a lower bound on the number of places in an Ada-net. The total number of places in an Ada-net is the sum of the number of places of the PSNs plus the places needed to model all potential task interactions. The PSN representing a task with c_{synch} synchronous entry calls, c_{rpc} RPC entry calls, a_{synch} synchronous accept statements, and a_{rpc} RPC accept statements has $\geq 2(c_{rpc}) + 4(a_{rpc}) + 2(c_{synch}) + 2(a_{synch}) + 2$ places. As described in the appendix of [SC88], accept statement with bodies are modeled with 4 places and all other accept statements and entry calls are modeled with 2 places. The additional 2 nodes model the beginning and end of the task. This lower bound on the number of PSN places is weak. Additional places

	seq	choice	seq-iter	choice-iter
seq	$2+8n$			
choice	$3+8n$	$4+8n$		
seq-iter	$3+8n$	$4+8n$	$4+8n$	
choice-iter	$4+8n$	$5+8n$	$5+8n$	$6+8n$

Table 2: Number of Ada-net transitions for communication patterns

are included to model all control flow statements that contain entry calls or accept statements.

It is clear that the upper bound on TIG nodes, discussed above, is less than the lower bound on PSN places. Furthermore TPNs require no additional places to represent potential task interactions, while Ada-nets require 2 places for each potential task interaction. This can result in a large number of additional Ada-net places since the number of potential interactions grows as the product of the number of entry calls to a single entry in a calling task and the number of accept statements for the entry in the accepting task.

Table 2 gives the number of Ada-net transitions created from the various combinations of communication patterns. This confirms that the number of TPN transitions is large.

Preliminary results for some small programs indicate that the effect of additional TPN transitions on the size of the reachability graph is offset by the reduction in the number of TPN places. More work needs to be done to understand this tradeoff.

5 Conclusion

In this paper, we have presented a Petri net model for tasking programs based on *task interaction graphs* that is efficient to construct. We have developed a strong upper bound on the number of places of these TIG-based Petri nets. Comparison with an existing Petri net model for Ada tasking programs provides evidence that the number of places in TPNs is relatively small. The analysis of the number of transitions demonstrates that for some patterns of task communication the number of transitions in TPNs is large. Although the number of transitions is of concern, it appears that the number of places is more important in terms of its impact on certain analysis techniques.

We intend to continue this work along a number of lines. We would like to extend the analysis of section 4 to include other Petri net models of Ada tasking programs. In addition we would like to evaluate TPN representation with respect to real programs and thus intend to build a TIG to TPN translator. This has the additional benefit of enabling us to experiment with existing Petri net analysis tools [MR87]. We are currently looking into methods for pruning dead transitions TPNs. This involves using conservative dataflow analysis to compute an approximate ordering relation for TIG edges. For any pair of TIG edges that are guaranteed to be ordered with respect to each other, i.e., they cannot execute in parallel, we need not construct a TPN transition. A number of dataflow problems are candidates for this application including B4 [DS91] and CHT [MR91]. Implementation of this technique would allow us to measure the amount of pruning of dead TPN transitions that is possible on real programs.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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