Improving Static Analysis Accuracy
on Concurrent Ada Programs:
Complexity Results and Empirical Findings

A. T. Chamillard

CMPSCI Technical Report 95-49
June 1, 1995

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

Abstract

This paper presents several techniques for improving the accuracy of static analysis of concurrent Ada programs. We determine the cost of building the program representations used to perform the analysis and examine the complexity of performing analysis on those representations. Inaccuracies in the static analysis are reflected in spurious results, which can be generated if the analysis considers paths through the program that are infeasible, or if the effects of aliasing lead to consideration of task communications that can not actually occur. We present three techniques to counter the effects of infeasible path consideration and aliasing and determine the cost of using these techniques. We present empirical results that demonstrate the improvements in accuracy and, in some cases, the reduction in the search space that result from application of our techniques.

1 Introduction

Developers creating concurrent software need cost-effective analysis techniques they can use to acquire confidence in the reliability of that software. Analysis of concurrent programs is difficult because in many cases the patterns of communication among the various parts of the program are nondeterministic and the number of possible communications is large. One class of techniques that can be used for analysis of concurrent programs is static analysis, which uses compile-time information to prove properties about a program. In general, though, many static analysis techniques produce inaccurate results. This paper describes one approach for static analysis of concurrent programs and three techniques that can be used to improve the accuracy of analysis using that approach. It also examines the costs of the various aspects of the analysis.

Any static analysis technique we use should be *conservative*; for a given property, the analysis must not overlook cases where the property holds, but may indicate that the property holds in more cases than are actually possible. To ensure conservativeness, many techniques use program representations that overestimate the behavior of the program being analyzed; as a result, these techniques may produce *spurious results*. A spurious result is an analysis result (i.e., a result about a property of a program) that does not correspond to a possible program behavior. Because the static analysis is performed on a representation that overestimates the program and can not use information available at run-time, the analysis may answer that a property holds, even if that property only holds on paths that can never be executed in the program (commonly called *infeasible paths*). Inaccuracies can be caused by aliasing when the analysis is unable to statically determine the memory location being used in an expression. For instance, static analysis may be unable to statically determine which elements of an array are affected by an operation. The standard approach to this problem is to assume all elements are affected by an operation on any array element; if the operation is a communication in a concurrent program, this approach can cause consideration of communications that are not actually possible in the program. Using the standard approach can thus lead to inaccuracies, since the analysis may answer that a property holds based on certain communications in the program, even if those communications can never occur.

This paper presents several techniques that may improve the accuracy of certain static analysis approaches; the techniques are presented in the context of reachability analysis based on a Petri net program representation. In an attempt to reduce the number of infeasible paths considered in the analysis, we include selected control and/or data information in the Petri net representation. The basic idea is to retain certain information about the states that the program being analyzed can enter during execution; this information may be in the form of specific program statements that have been executed to reach the current program state, or in the form of variable values. To address the aliasing problem, we can use information about the interactions between elements of arrays in some programs to reduce the size of our Petri net representation of those programs.

To ensure our proposed techniques are cost-effective, we must carefully analyze the costs associated with them. These costs can be divided into three kinds: the cost of generating the program representations for the analysis, the cost of performing the analysis on those representations, and the cost of refining the representations using our proposed techniques. The first two costs are analyzed in Section 4, while the cost of refinement is determined as each technique is introduced.

The following section describes some of the approaches that have been used to perform static analysis of concurrent programs. Section 3 describes the program representations we use to examine concurrent programs with our approach, and Section 4 examines the complexities of building and analyzing those representations. Section 5 explains how we represent certain state information to improve accuracy, discusses reductions based on array element interactions, and examines the cost of including the additional information or applying the reductions. Section 6 provides an analysis of our empirical results. Section 7 offers some conclusions based on those results and some pointers to future work.

2 Related Work

Numerous techniques for static analysis of concurrent programs have been proposed, with various time and space requirements. For many of the techniques, methods have been proposed for improving the accuracy of the analysis.

One analysis approach is to check the property of interest by considering all reachable states of the program being analyzed; this approach is commonly called reachability analysis. The set of reachable program states can be estimated using a variety of program representations, including flow graphs [Tay83a, YTL+92] and Petri nets [Pet77, SC88, DCN94]. Theoretical results [Tay83b] imply that, in general, the time and space requirements for this technique are exponential. Methods proposed for improving accuracy of reachability analysis include combining reachability analysis with symbolic execution [YT88] and using program variable value information in the analysis [BDF92]. The accuracy-improving techniques in the following sections are presented in the context of reachability analysis based on a Petri net representation of the programs to be analyzed.

Symbolic model checking techniques [BCM+90] represent the program state space symbolically rather than explicitly. A formula for the property of interest is specified, the program to be analyzed is modeled using Binary Decision Diagrams (BDDs), and a fixed point algorithm is used to determine whether the property formula is valid in the program model. Because checking Boolean satisfiability is NP-complete, determining the validity of the formula in the program model can require exponential time in the worst case; the BDD representations can require exponential space in the worst case. Because the BDDs developed in [BCM+90] contain a large amount of information about the program state, including variable values, techniques for improving the accuracy of this approach are not yet prevalent in the literature.

The Constrained Expression approach [ABC+91] avoids representing the state space of the program altogether. The program and a set of necessary conditions for the property of interest are expressed as a system of inequalities, and integer linear programming techniques are used to determine whether the necessary conditions can be satisfied by the program. In the worst case, solving the system of inequalities can require exponential time, and the system of inequalities can require exponential space. Including information about certain program variable values in the set of inequalities has been proposed [Cor93] as one way to improve the accuracy of this technique.

Another approach is to use data flow techniques for analysis of concurrent programs. Exploration of this approach [TO80, RS90, MR91, CK93, DC94] shows that polynomial-time algorithms can be used in data flow analysis to prove a wide range of program properties. Methods proposed for improving accuracy include identifying program statements that can not execute concurrently [MR93] and including selected information about program paths and program variable values in the analysis [DC94].

3 Program Representations

Because Ada is one of the few commonly used languages supporting concurrency, and because our current tools are written for analysis of Ada programs, we use Ada examples to explain our analysis techniques. We briefly describe here the principal concurrency constructs in Ada and several sources of nondeterminism in concurrent Ada programs. In Ada programs, potentially concurrent activities occur in *tasks* or procedures; for simplicity, we call them tasks in this paper. Ada tasks typically communicate with each other using a *rendezvous*. In a rendezvous, the calling task makes an *entry call* on a specific *entry* in the called task; the calling task then suspends execution until the called task terminates the rendezvous. The called task executes any statements contained in the *accept body* for the entry (the entry in conjunction with the accept body is called an *accept*), then terminates the rendezvous and continues execution. We note that the accept body may not have any statements, in which case the rendezvous simply acts as a synchronization point between the two tasks; data can also be passed between the two tasks at this point through parameters. Nondeterminism is introduced into an Ada program's execution in several ways. If several tasks make an entry call on the same entry, the called task could rendezvous with any of the calling tasks; determining which tasks will participate in the rendezvous is not possible. Another source of nondeterminism is the *select* statement. Select statements have one or more alternatives; when program execution reaches a select statement, the run-time system nondeterministically selects a task for the rendezvous from the alternatives available in a task queue.

In our approach, we use a variety of representations of a concurrent program to capture information about the program. We use a Control Flow Graph (CFG) for each task to represent the flow of control in the task, and a Task Interaction Graph (TIG) for each task to abstract sequential regions of control flow into single nodes. The nodes in the TIG for a task are connected by edges representing possible interactions (entry calls/accepts) between that task and other tasks in the program. We combine the set of TIGs for all the tasks in a program into a TIG-based Petri Net (TPN) to model the system as a whole. Finally, we use the TPN to generate a reachability graph to represent an estimate of all states the program can enter when started in the initial program state. Each of these representations is described more fully below.

Control Flow Graphs

One way to represent the behavior of a program is with a control flow graph [Hec77]. A control flow graph (CFG) is similar to a flow chart, in that it represents all paths through a procedure or task. A control flow graph consists of a finite set of nodes, N = $\{n_i\}$, and a finite set of directed edges, E = $\{e_i\}$. The set of nodes includes a single start node and a single end node for the CFG. In general, there is a single node in the CFG for each of the following: the declaration of the task and any local variables in the task (this node is called a Decl_Region node), the task begin statement, the task end statement, and each executable task statement. We call the task begin statement, task end statement, and set of executable task statements the set of task statements. The start node in a CFG is always the Decl_Region node. There is an edge from n_i to n_j if the task statement corresponding to n_j is potentially executable immediately after execution of the task statement corresponding to n_i. There is also an edge from the start node to the node generated for the task begin statement, an edge from the node generated for the task begin statement to the node generated for the first executable statement in the task, and an edge from each of the nodes representing a possible last executable statement in the task to the node generated for the task end statement. Because our transformation of a program into a CFG is mainly syntactic[1], some of the paths through the CFG may be infeasible. The CFG is a conservative program representation, since it overestimates the set of possible paths through the task.

The start node of the CFG (the Decl_Region node) is generated from the declaration of the task and any local variables in the task. The remaining nodes are generated from the executable task statements, task begin statement, and task end statement, with edges included as described above. Each entry call in the task is represented by a single node, just as a procedure call would be represented. Each accept statement in the task is represented by an Accept_Begin_Stmt node, zero or more nodes representing the executable task statements in the accept body, and an Accept_End_Stmt node. Accept statements with no executable task statements in the body are the only instance in which we add two CFG nodes for a single task statement; therefore, the number of nodes in a CFG is never greater than twice the number of task statements in the corresponding task. For an example of a simple Ada task and the associated CFG, see Figure 1. In the figure, we annotate each node in the CFG with the kind of the statement from which it was created.

```
task body t1 is
   Flag : Boolean;
begin
   select
      accept entry1;
      Flag := true;
   or
      accept entry2 is
         Flag := false;
      end accept;
   end select;
   t2.entry3;
end t1;
```

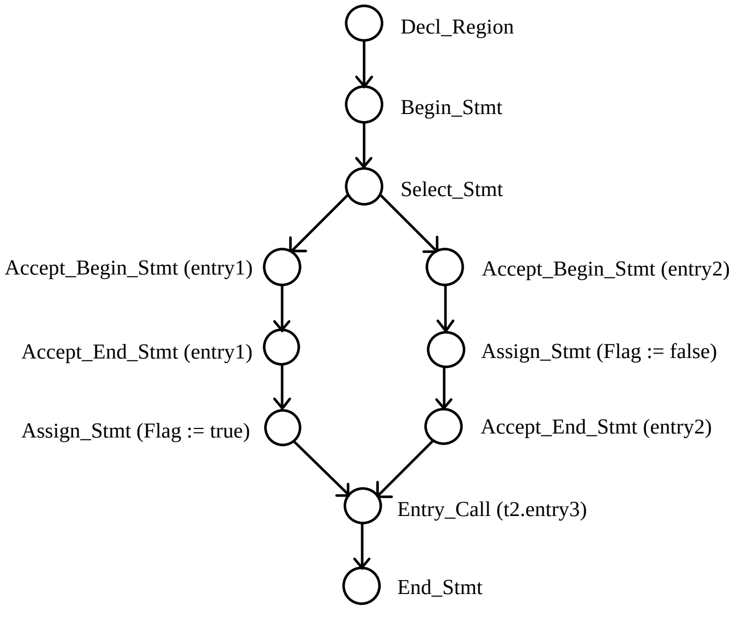

Figure 1. Example Control Flow Graph

[1] The only semantics considered during the transformation of an Ada program into a CFG are the semantics of the programming language; semantics of the specific program being transformed are not taken into account.

Task Interaction Graphs

Because each executable task statement is represented by at least one CFG node, the CFG for a task may include (as distinct nodes) a potentially large number of sequential statements that do not directly affect the task interactions in the program. Long and Clarke [LC89] suggest using Task Interaction Graphs (TIGs) to reduce the size of the program representation while retaining interaction information.

The TIG, like the CFG, consists of a finite set of nodes, N = $\{n_i\}$, and a finite set of directed edges, E = $\{e_i\}$. The difference is that each node n_i represents a region of sequential code, rather than a single task statement, and each directed edge represents a task interaction (either the start or end of an entry call or an accept). The set of nodes includes a single start node and a set of terminal nodes for the TIG. There is an edge from n_i to n_j if and only if the task can potentially participate in the task interaction represented by the edge, causing the task to exit the sequential region represented by n_i and enter the sequential region represented by n_j. As with CFGs, some of the paths in a TIG may be infeasible. The TIG for the task in Figure 1 is shown in Figure 2 below.

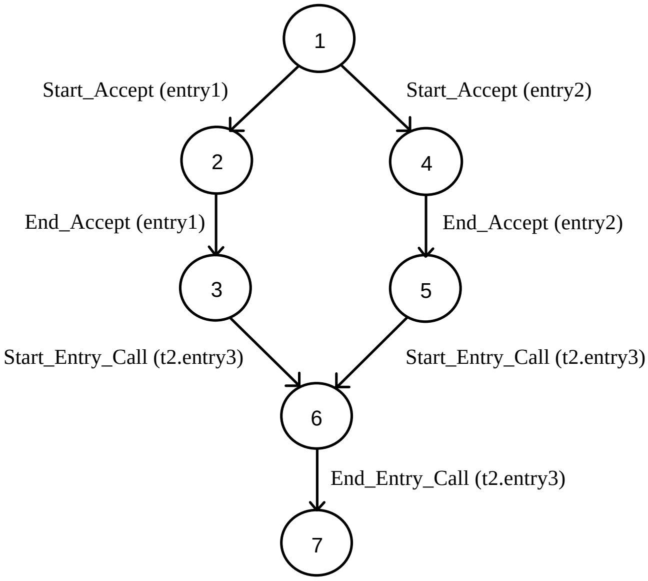

Figure 2. Example TIG

Note that we create two TIG edges from a single CFG node for entry calls; we do so to support our analysis of the set of TIGs for a program. To consider possible interactions in the set of TIGs, we match entry calls and accepts; if the accept has statements in the accept body, we need to be able to match edges to both the Start_Accept and End_Accept edges of the called task.

We observe that TIGs can be considered a simple coarsening of our control flow graph, in which we change our definition of a node to include all task statements except those representing possible interactions between tasks. The set of task statements forming the sequential region of a TIG node are stored as part of the node to support later analysis activities.

Petri Nets

Petri nets have been proposed as a natural and powerful model of information flow in a system [Pet77]. A Petri net can be represented as a 5-tuple (P, T, I, O, M0). P is the set of places in the Petri net; a place can hold zero or more tokens. If a place holds one or more tokens, the place is said to be *marked*. T is the set of transitions in the Petri net; tokens are moved between places in the net by the *firing* of transitions. A transition can only be fired if it is *enabled*; for a transition to be enabled, each of the input places for the transition must contain at least one token. I is a function mapping places in P to inputs of transitions in T. When a transition fires, a token is removed from each of the places that are inputs to the transition, and a token is deposited in each of the output places of the transition; O is a function mapping places in P to outputs of transitions in T. M0 is a list of all the places in the net that are initially marked.

Petri net modeling appears to be a valuable tool for modeling concurrent software [SC88]. A Petri net can be generated directly from the control flow graphs of a given program. We can also build a Petri net from the TIGs of a program [DCN94], with the resulting Petri net called a TIG-based Petri Net (TPN). To generate the TPN, we build a place for each node in the set of TIGs of a program. We build a transition for each possible interaction between tasks in the program, where the input places correspond to the sources of the matching entry call and accept TIG edges, and the output places correspond to the targets of the matching entry call and accept TIG edges. To form the initial marking M0 of the TPN, for each task in the program we put one token in the place that corresponds to the start node of the TIG for that task. For an example TPN, based on the program *data* in Appendix 1, see Figure 3. The places in the figure are labeled with the task name and TIG node number from which they were created.

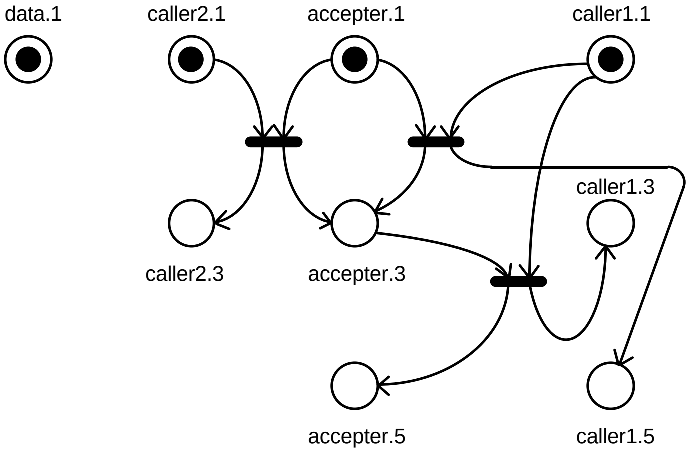

Figure 3. TIG-based Petri Net

Before examining TPNs in the context of certain standard Petri net properties, we make several observations about general characteristics of TPNs. We note that each task in the program can have only one of its places marked at any time; this corresponds to the fact that each task can only be in one place in its execution at a time. Each transition in the TPN has an input place from each of two distinct tasks and an output place for each of those same tasks; because each task can only have one place marked at any time, a marked place in a task can only be the output of an enabled transition if it is also an input to that transition. This characteristic will be of use to us in some of our proofs of TPN properties. In the initial marking M0 of a TPN, each place contains at most one token, since a place corresponds to the initial state of the corresponding task (so it has one token) or it does not (so it has zero tokens). With these TPN features in mind, we can now consider some important Petri net properties.

A Petri net is called *safe* if each place in the Petri net can contain at most one token.

Theorem 1. TPNs are safe.

Proof. To see that all TPNs are safe, we observe that places can only be made *unsafe* (containing more than one token) when a transition, which represents an interaction between two tasks, is fired. We note that only the output places of a transition can be made unsafe when that transition is fired. We first consider the safety of the places in the task making the entry call in such an interaction. For each transition in the net, we must consider two cases:

Case I: A single place in the calling task is both an input and an output of the transition; this can occur if the sequential region represented by the place is contained in a loop. In this case, the token is removed from the place by the transition, then deposited back into the place. Clearly, the place can contain at most one token.

Case II: The input and output places are distinct. In this case, the only way the transition can lead to an unsafe place is if the output place already contains a token. By the semantics of Petri nets, the input place must be marked for the transition to fire; because the input and output places are distinct and a task can only have one place marked at a time, the output place can not be marked if the transition is enabled. Thus, the output place in this case is safe.

We can apply the same arguments to the input and output places of the accepting task for the interaction. Thus, none of the transitions in the TPN can lead to an unsafe Petri net, so all TPNs are safe. Q.E.D.

A Petri net is called *conservative* if the number of tokens in the net is constant, which implies that for each transition the number of inputs is equal to the number of outputs.

Theorem 2. TPNs are conservative.

Proof. Because all of the transitions in a TPN have exactly two input places and two output places, there are no transitions in the TPN that *generate* (add to the system) or *consume* (remove from the system) tokens. Thus, TPNs are conservative. Q.E.D.

Reachability Graphs

In many cases, we would like our analysis to determine whether or not the concurrent program being analyzed could potentially enter a state in which a specified property holds (deadlock, for instance). One approach to answering such questions is to enumerate all possible program states and check the property at each state; a reachability graph can be used to represent the program state space.

A reachability graph for a Petri net consists of a set of nodes, N = {n_i}, and a set of arcs, A = {a_i}. If the Petri net is safe, N and A are finite. Nodes in the reachability graph correspond to markings of the Petri net; the root node of the reachability graph corresponds to the initial marking (M0) of the Petri net. An arc goes from n_i to n_j if and only if the marking of the Petri net can change from n_i to n_j with the firing of a single transition. Although in actuality several interactions, represented by fired transitions, can take place concurrently, we can capture all possible execution sequences by firing a single transition at a time; we use this approach, because the resulting graph is greatly simplified. We note that only markings reachable from the initial marking by some sequential combination of transition firings are included in the reachability graph. It is helpful to observe that a marking of a TPN simply represents the states of all the tasks being modeled by the TPN; we therefore consider nodes in the reachability graph as states the program can reach when started from the initial program state. Figure 4 provides the reachability graph for the TPN in Figure 3. The nodes in the figure are labeled with the TPN places that are marked in the corresponding program state.

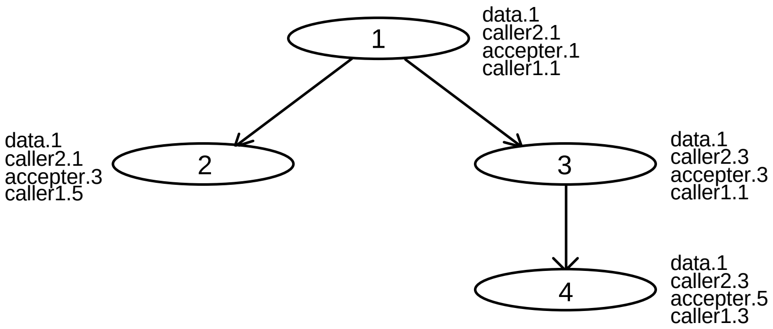

Figure 4. Reachability Graph

4 Complexity Results

There are three types of complexities considered in this paper: the cost of generating the representations used in the analysis, the cost of performing the analysis on those representations, and the cost of refining those representations using our proposed techniques. The first two costs must be considered for any representations used for static analysis, so we address them here. The complexity analysis for the accuracy-improving techniques will be more understandable after the techniques are introduced, so we defer these analyses to the next section.

For each of the algorithms in this paper, we provide an asymptotic upper bound on the cost of the algorithm using O-notation. We use O-notation because it provides an upper bound on the worst case running time of our algorithms, while also giving a general complexity measure that is valid for the algorithm on all inputs. For a given function g(n), O(g(n)) can be defined as {f(n) : there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$} [CLR92]. O(g(n)) represents a set of functions; when we show an algorithm has cost O(g(n)), we demonstrate membership in that set of functions. In general, O-notation does not specify how tight the upper bound is on the algorithm in question. For clarity, we only include factors that dominate the cost in the O-notation for our algorithms; for example, the n^3 in an $O(n^3 + n)$ algorithm dominates the cost, so we express the algorithm cost as $O(n^3)$. Including only dominant factors slightly loosens the bound provided, but makes the worst case cost of the algorithm more readily apparent.

Some of the results below raise interesting questions about the value of worst case complexities versus average case complexities for our algorithms. Worst case complexity is a useful measure, since it provides a solid upper bound on the cost of a given algorithm. However, if the worst case is rare, then the average complexity may be more representative of the algorithm's performance in actual practice. If in these cases we achieve an average complexity that is much better than the worst case complexity, then average complexity may be a more useful measure for determining the applicability of that algorithm. Since determining average case algorithmic complexity is much more difficult than worst case analysis, especially when the input distribution is unknown, we provide worst case complexities for our algorithms. We note, however, that promising algorithms should not be rejected simply because they have poor worst case complexities; through careful experimentation, we may discover that the worst case is rare and that the performance of these algorithms in the average case is acceptable.

Generating Representations

This subsection examines the costs of the algorithms we use to generate our program representations. Where applicable, we provide the algorithmic cost in terms of number of CFG nodes in a task or total number of CFG nodes in the program. We provide the cost in these terms because the rest of our representations are based directly (TIGs) or indirectly (TPNs, reachability graphs) on the CFGs for the program being analyzed.

To construct a TIG from a CFG, we need to traverse the CFG, creating a TIG node for each sequential region, then connecting the TIG nodes with edges representing the task interactions that lead between these regions. Some algorithms for performing the TIG construction, and their complexities, are discussed below.

An algorithm to create the TIG nodes and edges is as follows:

Algorithm 1

```
1  for each node in the CFG
2      if the node represents a task interaction then
3          if the CFG node is an entry call node
4              create 2 new TIG edges, one for start_entry_call and one for
                   end_entry_call
5              connect the edges with a new, empty TIG node between them
6              create a new TIG node as the destination of the end_entry_call edge
7          else
8              create a new TIG edge representing the interaction
9              create a new TIG node as the destination of the new TIG edge
```

Steps 2-9 take constant time. The loop in step 1 executes (number of CFG nodes) times. Thus, the algorithm above takes O(number of CFG nodes) time.

After creating the TIG nodes and edges, we need to connect them appropriately. An algorithm to do so is:

Algorithm 2

```
1  for each node in the TIG
2      retrieve the CFG node corresponding to the entering edge for the TIG node
3      for each successor of the CFG node
4          if the successor is a select statement
5              if there is a select alternative that does not represent a task interaction
6                  label the edge as "non-blocking" (this label is used in later
                       analysis)
7              else
8                  label the edge as "blocking" (this label is used in later analysis)
9          if the successor is a task interaction
10             connect the TIG edge corresponding to the successor node to the TIG
                   node
```

Within the loop in Step 3, Step 5 dominates the complexity, since it involves looking at all the successors of the current CFG node; in the worst case this takes O(number of CFG nodes) time. The loops in Steps 1 and 3 each iterate (number of CFG nodes) times in the worst case, so the total execution time for this algorithm is O(number of CFG nodes^3).

Combining the two algorithms above, we find that it takes O(number of CFG nodes^3) time to build a TIG from a CFG. Note that, technically, the cost of the algorithm is O(number of CFG nodes + number of CFG nodes^3). We observe that the (number of CFG nodes$^{3)}$ term dominates the cost, and demonstrate this with the following logic. O-notation tells us that the cost of the first algorithm is ≤ c*(number of CFG nodes) for some constant c. Similarly, the cost of the second algorithm is ≤ c'*(number of CFG nodes^3) for some constant c'. The total cost of the combined algorithms can then be expressed as c"*(number of CFG nodes + number of CFG nodes^3), where c" = max (c, c'). Since (number of CFG nodes + number of CFG nodes^3) ≤ (2*number of CFG nodes^3), we can express the total cost of the algorithms as ≤ 2c"*(number of CFG nodes^3), which is O(number of CFG nodes^3).

We note that, in some cases, we can perform a simple optimization on the set of TIGs for a program. If all possible accepts for a given entry call do not have accept bodies, we can replace the start_entry_call edge, the end_entry_call edge, and the empty node between them with a single synch_entry_call edge; we also replace the start_accept edge, end_accept edge, and the empty node between them with a single synch_accept edge. We perform this optimization on all the sets of TIGs in our examples. To perform this optimization we must compare each edge with all other edges, looking for possible entry call/accept matches; this optimization thus costs O(number of TIG edges^2) time. Because we only generate TIG edges from CFG nodes, this optimization costs O(total number of CFG nodes for program^2).

As discussed above, we can generate a Petri net to model a concurrent program by using the set of CFGs for a given program directly; an algorithm to do so is as follows:

Algorithm 3

```
1  create a new Petri net place for each CFG node in the program that does not correspond to a
       task interaction
2  for each edge in the CFG that does not have a task interaction node as source or destination,
3       create a transition with the source node as input and the target node as output
4   for each node in the CFG corresponding to a task interaction
5       for all nodes in other tasks/procedures of the program representing task interactions
6           if the nodes match (entry call/accept or accept/entry call)
7               create new transitions with the places corresponding to the CFG nodes
                that are predecessors of the matching nodes as inputs, and the places
                corresponding to the CFG nodes that are successors of the matching nodes
                as outputs
```

Step 7 requires checking each node in the set of CFGs for the program to see if it is a predecessor or successor of one of the matching nodes, so this step takes O(total number of CFG nodes for program) time. The loops in steps 4 and 5 each iterate (total number of CFG nodes for program) times in the worst case. Step 2 requires examining each edge in the CFG, which in the worst case takes O(total number of CFG nodes for program^2) time. Step 1 takes O(number of CFG nodes for program) time, since we check each node to see if it corresponds to a task interaction or not. The total time for the algorithm is thus O(number of CFG nodes for program^3).

Similarly, an algorithm for generating a TPN from a set of TIGs for a program is as follows:

Algorithm 4

```
1   create a new TPN place for each TIG node
2   for each edge in the TIG
3       for all TIG edges in other tasks/procedures of the program
4           if the TIG edges match (entry call/accept or accept/entry call)
5               create a new transition with the places corresponding to the TIG nodes
                that are sources of the matching edges as inputs, and the places
                corresponding to the TIG nodes that are targets of the matching edges as
                outputs
```

Step 4 takes constant time. Step 5 also takes constant time, because TIG edges contain explicit source and target information. Since in the worst case we can have a TIG edge for each CFG node, the loops in Steps 2 and 3 each execute (total number of CFG nodes for program) times in the worst case. Step 1 takes O(total number of CFG nodes for program) time, so the total time for the algorithm is O(total number of CFG nodes for program^2). Recalling that it took O(number of CFG nodes^3) to generate the TIG from the CFG for a task, we observe that going from CFGs to TPNs costs O(total number of CFG nodes for program^3) time in the worst case. We thus have the same algorithmic complexity whether we build the net directly from CFGs or generate TIGs first, though we note that the net built from TIGs is generally smaller than that built directly from the CFGs [DCN94].

To build the reachability graph for a TPN, we can use the algorithm below:

Algorithm 5

```
1   create a reachability graph node for the initial marking of the TPN
2   add the new node to the front of search list
```

```
3  while the search list is not empty
4      remove the current search node from the front of the search list
5      add the current search node to the set of visited nodes
6      identify enabled transitions for the marking of the current search node
7      for each enabled transition
8          generate a new marking by firing the transition
9          if the new marking has not been visited
10             create a reachability graph node for the new marking
11             add the new node to the front of the search list
12         connect an arc from the current search node to the node for the new marking
```

Steps 10-12 take constant time. The check in Step 9 takes O(number of nodes in reachability graph) time in the worst case. Firing a transition in Step 8 can be accomplished in constant time. The loop in Step 7 executes (number of enabled transitions in TPN) times; in the worst case, this step takes O(number of transitions in TPN) time. Step 6 takes O(number of transitions in TPN) time, since we must check each transition in the TPN to see if it is enabled. Steps 4 and 5 take constant time. The loop in Step 3 executes (number of nodes in reachability graph) times, and Steps 1 and 2 take constant time. The total cost of the algorithm is then O(number of nodes in reachability graph^2 * number of transitions in TPN).

We should make several observations about the algorithm above. Placing the reachability graph nodes on the front of the search list in Steps 2 and 11 yield a depth-first generation of the reachability graph; for the same complexity, we could add the nodes to the end of the list for a breadth-first generation. The space requirements for the search list for a depth-first traversal are generally much less than those for a breadth-first traversal, so we chose to implement depth-first graph generation.

Step 9 contributes a multiplicative factor of (number of nodes in reachability graph) to the cost of the algorithm; this cost could be obtained by maintaining a list of the visited nodes, and searching the list from the beginning each time we check to see if a specified node has been visited. Holzmann [Hol88] suggests hashing the marking as an index into a hash table of the visited nodes; using this approach yields a constant time check for a visited node on average. In the worst case, if all the nodes hash to the same hash table index the checks can still take O(number of nodes in reachability graph) time. We have implemented this approach for its average performance.

Because it is difficult to predict the distribution of numbers that will represent markings, we have also implemented a universal hashing scheme, which uses randomly selected multipliers in the hash function to avoid consistent worst case performance. Our universal hashing algorithm is from [CLR92]; Theorem 12.3 in [CLR92] proves that, if we use a hash function selected from the universal hashing functions and we are hashing fewer keys than slots in the hash table, then the expected number of collisions for a given key is less than 1. Thus, the expected time to determine whether a node has been visited is constant, since the chains we must search in the hash table are very small. On average, then, generating a reachability graph from a TPN takes c*(number of nodes in reachability graph * number of transitions in TPN) time, where c is some constant. Of course, the worst case is still as described above.

One of the drawbacks of reachability graphs is that they tend to grow large very quickly as the program size increases. It would thus be helpful to be able to check properties of reachable program states without actually generating the reachability graph. The *reachability problem* tries to determine whether a Petri net started with a given marking can reach a specified marking. Mayr [May84] presents an algorithm for solving the reachability problem using an iterative approach rather than generating the entire reachability graph and inspecting it for the marking of interest. While the idea of avoiding generation of the entire reachable state space is intriguing, there are several problems with this approach. The complexity of the algorithm presented is unknown, so it is unclear what the cost of this approach would be in practice. More importantly, we are usually interested in checking whether markings with a certain property are reachable, rather than whether a specific marking is reachable. It is therefore not clear that the approach described in [May84] is applicable to our analysis.

Analyzing Representations

We now move from examining the cost of generating our program representations to determining the cost of performing analyses on those representations. The complexity of answering various questions about Petri nets has been widely studied in the literature [Gra79, Jan87, JLL77, May84, MM81]; it turns out that the complexities of answering these questions extend over a wide range. The first two results below do not directly relate to our analyses, since it is doubtful we will need to check whether two TPNs have the same set of reachable markings; however, we include them here to more completely demonstrate the range of Petri net problem complexities.

Both Grabowski [Gra79] and Jantzen [Jan87] have shown that determining whether the reachable markings for two arbitrary Petri nets are identical, called the Petri net equality problem, is undecidable. This is proved using a reduction of Hilbert's Tenth Problem to the Petri net equality problem; Hilbert's Tenth Problem has been shown to be unsolvable [Dav73]. Grabowski goes on to prove that the problem becomes decidable if the Petri nets in question have no unbounded places; we note that safe Petri nets have no unbounded places.

Mayr and Meyer prove in [MM81] that, if the reachability sets of two Petri nets are finite (which is true of safe Petri nets), determining equality of the sets (called the Finite Equality Problem, or FEP) is decidable; the problem is not, however, primitive recursive. First, they provide a bounded version of Hilbert's Tenth Problem to find solutions in a finite initial segment of N, which is decidable by exhaustion. They then introduce a nonprimitive recursive problem called the Bounded Polynomial Inequality Problem (BPI), which determines if, for a certain range of inputs bounded by a nonprimitive recursive function similar to Ackermann's function, p(y) <= q(y) for two arbitrary functions p and q and input y. Next, they note that Petri nets can be used to compute polynomial functions with nonnegative integer coefficients; the result is called a Weak Petri Net Computer (WPNC). Jantzen [Jan87] shows that any function computed by a WPNC is primitive recursive. Using slightly modified WPNCs for functions p and q, Mayr and Meyer perform a poly-time reduction of BPI to the Finite Equality Problem. Finally, they conclude that FEP is decidable, but not primitive recursive. This result is of theoretical interest, since FEP seems to be one of a small set of uncontrived decidable problems that are not primitive recursive.

Jones et al. [JLL77] examine the complexities of a variety of problems on Petri nets. While many of their results are for arbitrary Petri nets, there is one result for safe Petri nets that is of interest to us. Given a Petri net and a constant k, determining whether the net is k-bounded (i.e., no place ever contains more than k tokens) is PSPACE-complete. Since a safe Petri net is simply a 1-bounded net, this result shows the difficulty of determining if a Petri net is safe. It is therefore critical that we carefully construct our TPNs as safe Petri nets rather than building them and then checking for safety; we have already shown that our TPNs are safe.

We next consider some general complexity results for reachability analysis of concurrent programs. Taylor examines several important problems in reachability analysis [Tay83b] and we review some of his results below. In many cases, he demonstrates NP-completeness by a reduction of 3SAT to the problem in question. We note that, although Taylor proves these results using a flow graph representation of the program, the complexity arises not from the representation itself, but from the many different task interaction possibilities in the program. Taylor's results thus generally describe the complexity of the reachability analysis, rather than the specific complexity of analysis on his representations, so they are applicable to analysis of our reachability graphs as well. At the moment, we restrict our attention to properties that can be checked solely by looking at program states; later, we discuss other types of properties that may be of interest.

Which rendezvous can occur in the program directly affects whether certain statements can execute in parallel, whether the program terminates, and so on. Taylor shows that determining whether a specific rendezvous from the set of all possible rendezvous can occur is NP-complete. He assumes that the rendezvous can occur if there is a path to that rendezvous in his flow graph representation of the program; as with CFGs and TIGs, some of the paths in his flow graph may be infeasible, and thus not actually executable. Determining whether a specific rendezvous can occur is even NP-complete under severe restrictions on the program. These restrictions include prohibiting branches, loops, and select statements in all tasks, or prohibiting branches and loops in the tasks and only allowing one task to have entry calls on a given entry.

Another question of interest is "Is there an execution sequence in which at least one task enters an infinite wait?" In other words, is it possible for at least one task to be unable to terminate in some execution of the program; this scenario is commonly called deadlock. Taylor shows this problem to be NP-complete, even under the severe restrictions cited above. It is clear that a variety of important questions in static analysis of concurrent programs are intractable; indeed, Taylor points out that "Only when enough restrictions are applied to make a system fully deterministic do the problems become tractable" [Tay83b].

Taylor's results indicate the complexity of using reachability analysis to answer certain questions in general; we now examine the cost of answering questions about specific properties using our representations (a reachability graph generated from a TPN). In the worst case, checking for deadlock in a reachability graph node consists of examining all possible communication choices within the TIG regions represented by the program state of the node. As pointed out in [DCN94], this can be exponential in the number of tasks in the program, though in practice it appears to be much less. Checking for critical data races is also examined in [DCN94]; the cost for each node in the reachability graph is found to be O(number of program tasks * number of shared variables) for the algorithm given. Thus, we see that the cost of checking properties in a reachability graph generated from a TPN is property-specific and, for the examples given, ranges from linear to exponential in the number of program tasks.

Thus far, we have discussed questions about properties that can be checked solely by looking at single program states; the cost of answering the question is simply the cost of examining the nodes in the reachability graph, and the time required to do so is O(number of nodes in reachability graph * time to check property at each node). The deadlock and critical race properties described above are of this type; we can check these properties by looking at the task states in the program at each reachability graph node. For many other questions about properties, the path to a node is important in addition to the node itself; we call these *path properties*. Path properties can be separated into universal questions and existential questions. An example of a universal question is "Does Sender 1 always stop sending a message after starting to send a message?"; an example of an existential question is "Does there exist a path in which Sender 1 starts sending a message and then Sender 2 starts sending a message before Sender 1 stops sending the message?" We differentiate between universal and existential questions because our interpretation of the analysis results will differ based on the type of question. If static analysis answers affirmatively to a universal question, we can accept that answer, because the analysis is conservative as described in Section 1. If the analysis answers affirmatively to an existential question, the result could be spurious, so we should investigate further to ensure the corresponding path is feasible. For both types of path properties, we must actually traverse paths in the graph rather than simply check nodes. In many cases our reachability graphs are cyclic; because the number of paths in a cyclic graph is infinite, these questions are difficult to answer using reachability graphs. General model checking can be used to check path properties on a reachability graph [YTL+92]; we could also use the reachability graph in a data flow problem to answer questions about path properties.

5 Improving Accuracy

Because we perform static analysis on models that overestimate the behavior of the system being analyzed, we may often answer questions about properties of the system with spurious results. The analysis may consider infeasible paths through the program, or aliasing can cause the analysis to consider communications in the program that are not actually possible[2]. Because a large number of spurious results will likely dissuade an analyst from trusting the analysis results, it is important to reduce the number of spurious results. On the other hand, we must be careful that the techniques we use to improve the accuracy of the analysis do not add significantly to the complexity of that analysis.

Here we examine several encouraging approaches for improving the accuracy of the static analysis without adding significantly to the cost of such analysis. The first two techniques involve modeling additional program state information to eliminate some infeasible paths from consideration by the analysis. The first technique explicitly retains information about past program states to eliminate some of these paths, while the second technique implicitly eliminates some infeasible paths by modeling variable values. The third technique excludes some impossible communications from consideration by accurately accounting for aliasing in an array of tasks.

Enforcing Impossible Pair Constraints

There are a variety of complex problems involving control flow through a program; one of these problems that affects static analysis concerns impossible pairs. Impossible pairs are pairs of program statements that can not both execute in the same execution of the program; in other words, both statements can not be on the same execution path. As noted in [GMO76], determining whether or not there exists an executable path in the program, constrained by

[2] Aliasing can also cause a variety of other problems for static analysis techniques.

impossible pairs of statements, is NP-complete (by reduction of 3SAT). In fact, as also shown in [GMO76], the problem remains NP-complete even if the control flow graph is acyclic, and the in-degree and out-degree of all nodes is limited to two. We describe below how to incorporate information about impossible pairs into our representation of concurrent programs, without having to actually solve the impossible pairs constrained program path problem.

In this paper, we use a different definition of impossible pairs from the one given above; we continue to use the term impossible pairs because of their close relationship to the impossible pairs above. The semantics of our impossible pair is such that executing the first member of the impossible pair should inhibit execution of the second member, but executing the second member of the impossible pair should have no impact on the executability of the first member. The key differences between our meaning and that in [GMO76] is that our pair is only impossible in one direction (first member to second member, not second member to first member) and we will let both statements execute in an execution of the program (given certain conditions described below). Our approach involves representing additional program state information to eliminate infeasible paths that contain both members of an impossible pair; for simplicity, we restrict our attention here to cases in which the impossible pair consists of two interaction (entry call or accept) statements. We observe that statements in an impossible pair are conceptually different from statements that Can't Happen Together (CHT) [MR93]; impossible pairs identification is concerned with identifying invalid sequences of statements, and CHT analysis is concerned with identifying statements that can not execute concurrently.

There are three distinct activities associated with our approach: recognizing the impossible pairs in a program, recognizing which regions in the program re-enable second members of the impossible pairs (discussed further below), and including information about the impossible pairs in the TPN. For simple programs, the impossible pairs and regions re-enabling them are easily recognized; in more complicated programs, automated techniques such as symbolic evaluation [CR81] can be used. We assume that some technique has been used to recognize the impossible pairs and the regions re-enabling them, so our discussion below focuses on including information about these impossible pairs in our TPN.

For an example of when this technique is useful, consider the program called *impos* in Appendix 1. If the ComplicatedCond in the caller1 task evaluates to true, leading to the entry call on entry1 in the first conditional, the call on entry2 in the second conditional is impossible because the truth value of ComplicatedCond doesn't change. We can use information about this impossible pair to improve the accuracy of the reachability graph.

For clarity in our initial explanations below, we assume a single impossible pair in the program. We recognize that multiple impossible pairs are possible in a program, and thus extend our explanations to account for these as well.

In general, to include impossible pair information in our TPN we add two new places to the TPN for the task statement that is the second member of the impossible pair in the program. The first new place, called the Enabled place for the statement, is used to enable execution of the statement. The second new place, called the Disabled place for the statement, reflects when the statement is not allowed to execute. Because we restrict our attention here to impossible pairs of interaction statements, the first member and second member of the impossible pair are each represented by one or more transitions in the TPN. We connect the Enabled place as an input of all transitions that correspond to the task statement for the second member, which ensures the statement can only execute when the Enabled place contains a token. We also connect the Enabled place as an output of these transitions, which lets the task statement execute multiple times. Following the semantics of impossible pairs, executing the first member of the impossible pair should prohibit the second member from executing; to enforce these semantics, we ensure that firing the transition corresponding to the first member of the impossible pair results in an unmarked Enabled place and a marked Disabled place for the second member of the impossible pair. Because the second member may be enabled or disabled before executing the first member, we copy the transition corresponding to the first member, including all inputs and outputs of the transition. We then use the original transition to change the second member from enabled to disabled when the first member is executed and the duplicate transition to keep the second member disabled if it is already disabled when the first member is executed; we call these *disabling transitions*. For a simple example of these connections, consider Figure 5 below, where transition 1 corresponds to the first member and transition 2 corresponds to the second member of the impossible pair. Note that no tokens are provided in the figure; based on the execution of the TPN to this point, either the Enabled place or the Disabled place will contain a token.

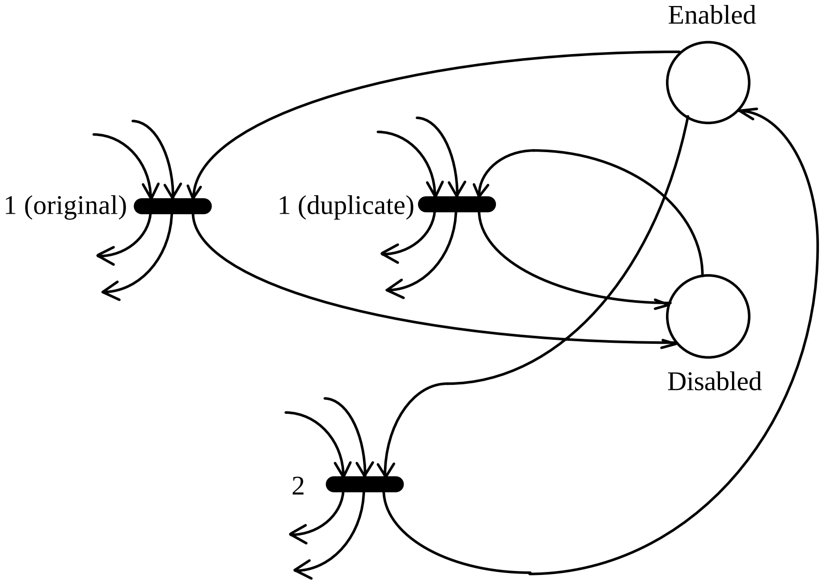

Figure 5. Disabling the Second Member of an Impossible Pair

We must also consider the possibility that the second member of an impossible pair should only be disabled temporarily. This can occur if the condition that caused the second member to be inhibited can change on some path from the first member to the second member of the impossible pair. For instance, if the two selects in the *impos* program were contained within a loop and ComplicatedCondition was changed at the end of the loop, the transition corresponding to the call on entry2 should be re-enabled at the end of the loop. Because the statement changing such a condition will typically not be an interaction statement, this statement is contained within the TIG region corresponding to a place in the TPN; we call this region a *re-enabling region*, since it re-enables execution of a statement. To re-enable the second member, we modify transitions into the places corresponding to the re-enabling region. Because the statement to be re-enabled may be enabled or disabled before we reach the transition to be modified, we copy the transition, including all inputs and outputs of the transition. We then use the original transition to change the statement from disabled to enabled and the duplicate transition to keep the second member enabled if it is already enabled; we call these *re-enabling transitions*. For a simple example of these connections, consider Figure 6 below, where place A corresponds to a region that re-enables the statement for which the Enabled and Disabled places have been generated and transition 3 is the transition to be modified.

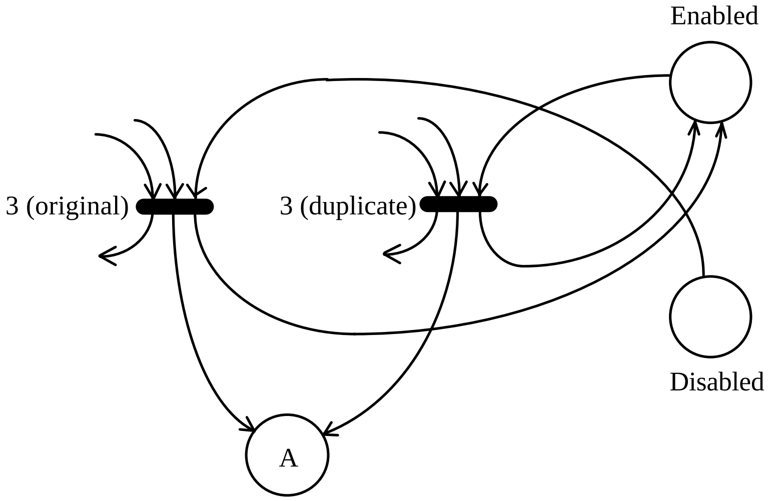

Figure 6. Enabling the Second Member of an Impossible Pair

To ensure that the second member is enabled or disabled (but not both), we connect the new places to the net such that exactly one of the Enabled place/Disabled place pair for the second member is marked at any given time. The Enabled place is initially marked, and the Disabled place is initially unmarked. Firing an original disabling transition removes a token from the Enabled place and places the token in the Disabled place; exactly one of the two places is still marked. Firing a duplicate disabling transition removes a token from the Disabled place and places the token back into the Disabled place. Firing an original re-enabling transition removes a token from the Disabled place and places the token in the Enabled place; exactly one of the two places is still marked. Firing a duplicate re-enabling transition removes a token from the Enabled place and places the token back into the Enabled place. Thus, exactly one of the Enabled place/Disabled place pair for a given statement is marked at any given time.

We now consider the cost of including information about impossible pairs in the TPN. Continuing with our assumption that there is a single impossible pair in the program, an algorithm to create the new places and connect them to the net is as follows:

Algorithm 6

```
1  create an Enabled place and a Disabled place
2  mark the Enabled place
3  for each transition in the net
4      if the transition corresponds to the first member of the impossible pair
5          add the Disabled place as an output of the transition
```

```
6               copy the transition
7               add the Enabled place as an input of the original transition
8               add the Disabled place as an input of the duplicated transition
9         if the transition corresponds to the second member of the impossible pair
10              add the Enabled place as an input of the transition
11              add the Enabled place as an output of the transition
12        if the transition leads to a place corresponding to a re-enabling region
13              add the Enabled place as an output of the transition
14              copy the transition
15              add the Disabled place as an input of the original transition
16              add the Enabled place as an input of the duplicate transition
```

Steps 1-2, 4-5, 7-13, and 15-16 take constant time. Because duplicating a transition involves creating a new transition and copying all the inputs and outputs from the old transition to the new one, Steps 6 and 14 each take O(number of input places per transition + number of output places per transition) time. Since the loop from Step 3 executes (number of transitions in the original TPN) times, the complexity of including impossible pair information in the net is O(number of transitions in original TPN + average number of input places per transition + average number of output places per transition). Because the number of transitions in the net is generally much larger than the number of inputs and outputs per transition (for our basic TPNs, the sum of inputs and outputs is 4), we can simplify this to O(number of transitions in original TPN).

Thus far, we have assumed a single impossible pair in the program. We are now ready to relax that assumption and consider more complicated representation of impossible pairs information. If there are multiple impossible pairs in the program, Steps 1 and 2 in the algorithm above must be executed for each impossible pair in the program; this contributes an additive factor of (number of impossible pairs in program) to the cost of the (slightly revised) algorithm. We note that, given a single impossible pair, we could assume that each statement in the program disables no more than one other statement, and each region re-enables no more than one statement. If one or both of these conditions does not hold, we iterate over the transitions in the TPN multiple times; on each iteration, we treat each transition as a disabling or re-enabling transition, as appropriate, for a single statement. Since the number of duplications performed on a given transition is (number of second members disabled + number of second members re-enabled), the number of iterations we perform over the transitions in the TPN is given by max(number of second members disabled + number of second members re-enabled). Thus, to get the total cost of including impossible pairs information in the TPN, we multiply the constant for the (slightly revised) algorithm by max(number of second members disabled + number of second members re-enabled). We include this cost in the constant rather than including it explicitly because the number of transitions in the original TPN and the number of impossible pairs will still typically be the dominant factor.

By iterating over the transitions in the TPN max(number of second members disabled + number of second members re-enabled) times and executing Steps 5-18 of the algorithm above on each iteration, we ensure we capture every combination of markings for Enabled and Disabled places affected by each transition. We note that if a transition in the original net affects (disables, re-enables, or some combination of the two) several second members of impossible pairs, the number of transitions in the revised net resulting from this transition is exponential in the number of second members affected. Thus, a transition in the original net that should disable 2 second members of impossible pairs will result in 3 new transitions (4 total) in the new net. Consider the simple example below, where transition 4 should disable both the statement corresponding to transition 5 and the statement corresponding to transition 6.

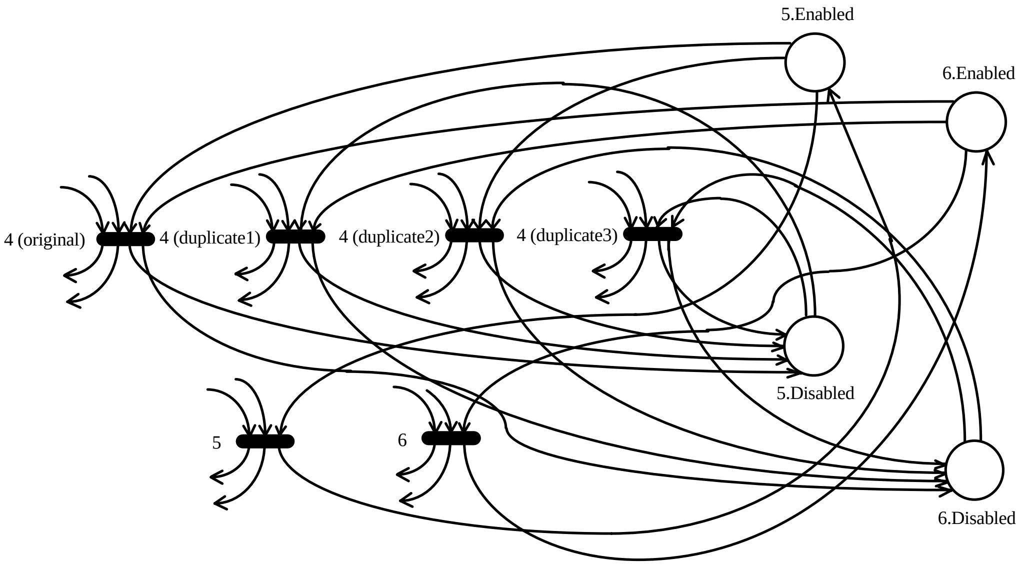

Figure 7. Disabling Multiple Second Members of Impossible Pairs

In addition to the costs discussed above, we must also consider potential additional costs incurred generating the reachability graph. Recall that the average reachability graph generation takes C*(number of nodes in reachability graph * number of transitions in TPN) time. The technique described above adds transitions to the TPN in several cases, and an increase in the number of transitions in the net increases the time it takes to generate the corresponding reachability graph if the number of nodes in the reachability graph stays constant (or increases); if the resulting graph has fewer nodes, the time to generate the graph may still increase. In either case, if the nodes in the graph will be traversed several times to check various properties, the additional initial time to generate the graph may be worthwhile. Of course, the resulting graph is more accurate as well, so the additional generation time could simply be considered the cost of adding accuracy.

Including information about impossible pairs not only affects the time to generate the reachability graph, it affects the size of the graph as well, both in terms of nodes and arcs. In some cases, including this information decreases the number of nodes in the reachability graph; in others, including the information increases the number of nodes in the graph. Adding transitions to the TPN can increase the number of arcs in the reachability graph, since each enabled transition from a given reachable state creates a new arc in the reachability graph. We include both node and arc counts in our experimental results in Section 6 to indicate how this technique affects the reachability graph for the programs shown.

It is important to note that the new Petri net continues to be safe.

Theorem 3. TPNs including impossible pairs information are safe.

Proof. For each transition, we must consider each case below; note that the cases are not disjoint, since a transition could correspond to the first member of an impossible pair, correspond to a second member of a different impossible pair, and lead to a place corresponding to a re-enabling region.

Case I: The transition corresponds to the first member of an impossible pair. After the modifications above, the original transition from Step 6 is only enabled if the Enabled place is already marked, while the duplicate transition from Step 8 is only enabled if the Disabled place is marked (and hence the Enabled place is unmarked). Since only one of these transitions can be enabled at any given time, only one of these transitions can place a token in the Disabled place. If the original transition from Step 6 fires, there was no token in the Disabled place, so adding a token to the place yields one token in the place. If the duplicate transition from Step 8 fires, the token is removed from the Disabled place before being placed back into the Disabled place, so the Disabled place still contains only one token.

Case II: The transition corresponds to the second member of an impossible pair. In this case, the token is removed from the Enabled place for the second member of the impossible pair by the transition, then deposited back into the Enabled place. Clearly, the place can contain at most one token.

Case III: The transition leads to a place corresponding to a re-enabling region. After the modifications above, the original transition from Step 14 is only enabled if the Disabled place is marked, while the duplicate transition from Step 16 is only enabled if the Enabled place is already marked (and hence the Disabled place is unmarked). Since only one of these transitions can be enabled at any given time, only one of these transitions can place a token in the Enabled place. If the original transition from Step 14 fires, there was no token in the Enabled place, so adding a token to the place yields one token in the place. If the duplicate transition from Step 16 fires, the token is removed from the Enabled place before being placed back into the Enabled place, so the Enabled place still contains only one token.

Because the original TPN was safe and no new or modified transitions can make the TPN unsafe, the resulting TPN is safe as well. Q.E.D.

We should also note that the resulting TPN continues to be a conservative Petri net, since the number of tokens in the TPN is constant.

Theorem 4. TPNs including impossible pairs information are conservative.

Proof. For each transition, we must consider each case below; note that the cases are not disjoint, since a transition could correspond to the first member of an impossible pair, correspond to a second member of a different impossible pair, and lead to a place corresponding to a re-enabling region.

Case I: The transition corresponds to the first member of an impossible pair. Firing the transition from Step 6 removes the token from the Enabled place and deposits the token into the Disabled place. Firing the transition from Step 8 removes the token from the Disabled place and deposits the token back in the Disabled place. Only one of the two transitions can be enabled at any given time, and each of them preserves the number of tokens in the TPN.

Case II: The transition corresponds to the second member of an impossible pair. In this case, the transition removes the token from the Enabled place, then deposits the token back into the Enabled place. Tokens are neither generated nor consumed by the transition, so this transition preserves the number of tokens in the TPN.

Case III: The transition lead to a place corresponding to a re-enabling region. Firing the transition from Step 14 removes the token from the Disabled place and deposits the token into the Enabled place. Firing the transition from Step 16 removes the token from the Enabled place and deposits the token back in the Enabled place. Only one of the two transitions can be enabled at any given time, and each of them preserves the number of tokens in the TPN.

Since the original TPN was a conservative Petri net and all new and modified transitions preserve the number of tokens in the TPN, the resulting TPN is conservative as well. Q.E.D.

We now return to the *impos* program in Appendix 1. The TPN without impossible pairs included for this example is shown in Figure 8, and the corresponding reachability graph is shown in Figure 9. Node 3 in the reachability graph represents a deadlock of the caller2 task; however, the transition fired to enter this node represents an interaction that is not possible based on the fact that ComplicatedCond evaluated to true in the first conditional in the caller1 task, and was not changed before the second conditional. Therefore, we have a reachability graph node in which deadlock occurs, but this is a spurious result, since the program can not actually execute the path required to reach the node. Using the technique for impossible pairs described above, we add impossible pairs information to the TPN. The resulting TPN is shown in Figure 10, where Steps 6-10 in the algorithm above have been performed on transition 1 and Steps 11-13 have been performed on transitions 2 and 3 of the original TPN. The corresponding reachability graph is shown in Figure 11. Note that we have retained the reachability graph node numbering from Figure 9 in Figure 11 to facilitate comparison. We see that the spurious result has been removed by the additional information included, and thus analysis of the resulting graph can yield more accurate results.

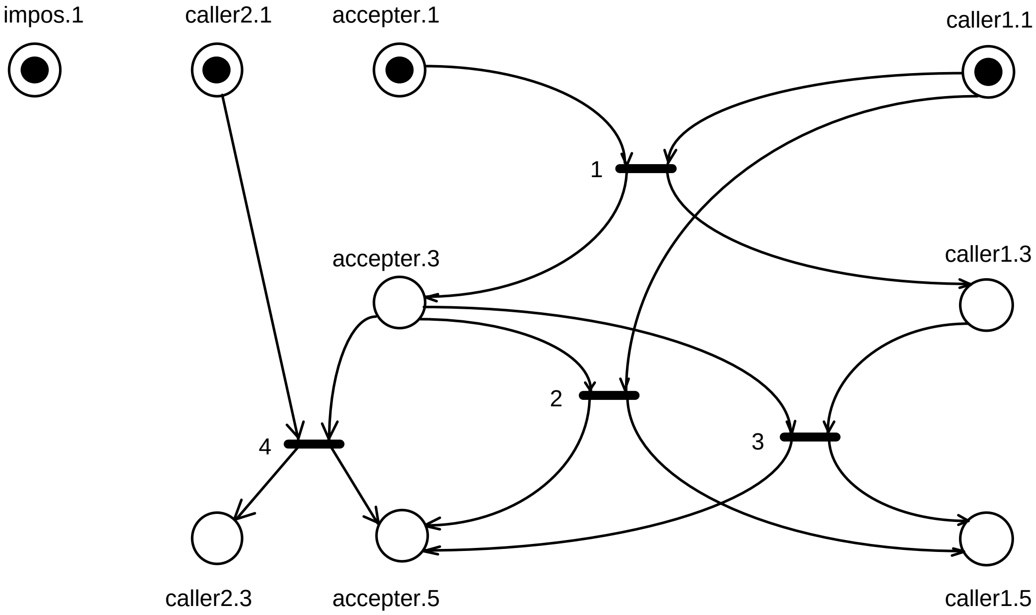

Figure 8. TPN Without Impossible Pairs Represented

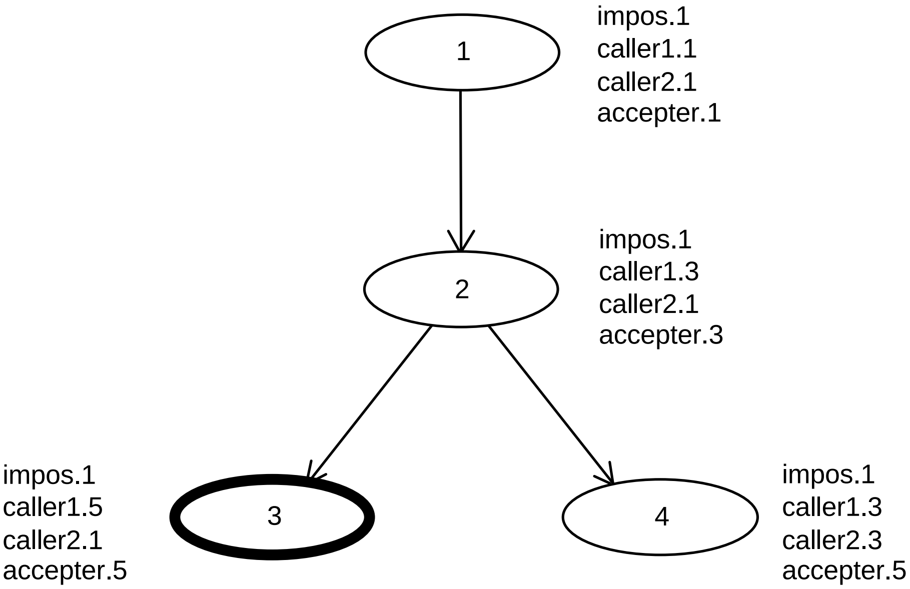

Figure 9. Reachability Graph Without Impossible Pairs Represented

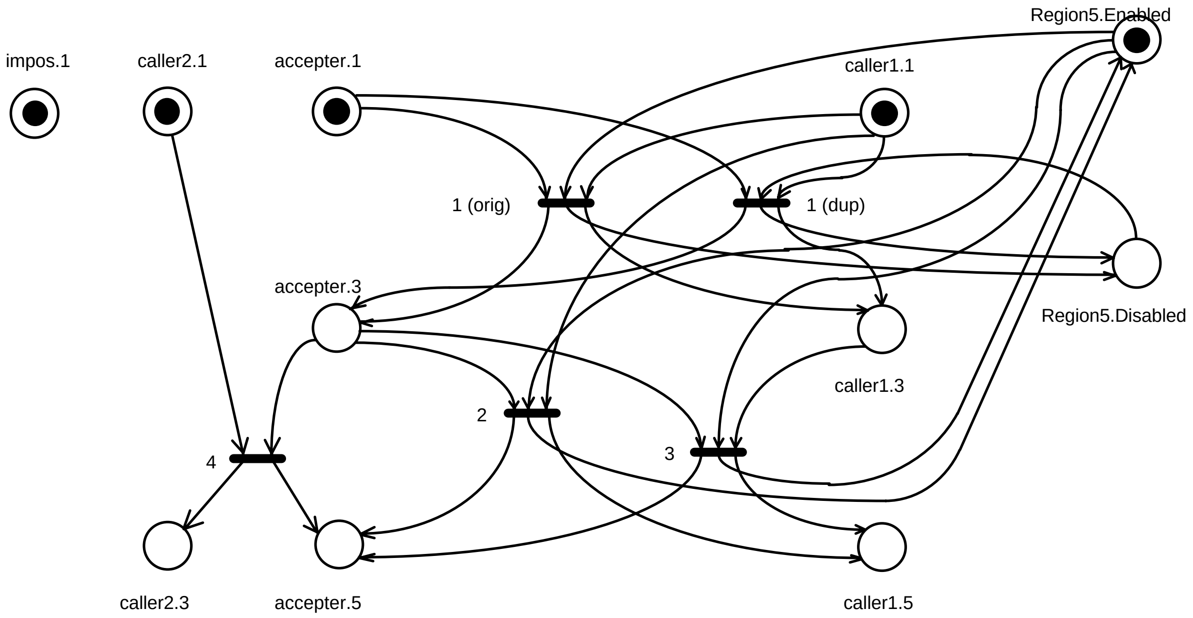

Figure 10. TPN With Impossible Pairs Represented

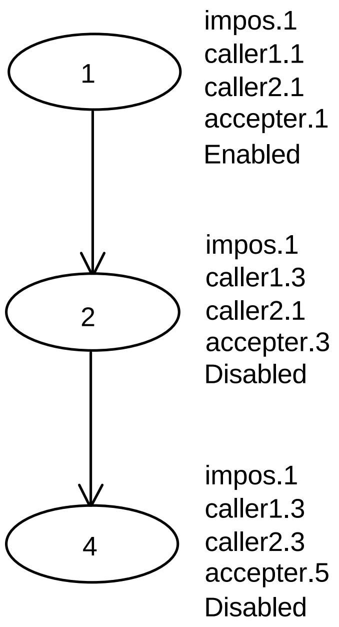

Figure 11. Reachability Graph With Impossible Pairs Represented

We therefore see that representing impossible pairs information in the TPN can lead to better accuracy in our static analysis. This approach is particularly attractive when static information about the impossible pairs in the program is readily available and transitions correspond to members of a single impossible pair. This approach will tend to be expensive for programs for which the TPN contains transitions that affect multiple members of impossible pairs, since the number of these transitions grows exponentially in the number of impossible pairs affected.

Representing Variable Values

When we include representation of impossible pairs information in our TPN, we eliminate some infeasible paths from consideration by explicitly representing information about paths in the program execution. We can also implicitly eliminate some infeasible paths by representing the values of selected variables in the program. As with the impossible pairs technique, we modify the TPN to capture additional information about the program states. In this case, however, the state information is in the form of variable values; we can use the additional information to exclude interactions that are infeasible based on those values, thereby excluding some infeasible paths from our analysis.

There are four activities to be considered when we represent variable values in our TPN: recognizing the interactions that are controlled by specific variable values, recognizing the regions that change the variable's value (and how they change it), building the representation for the variable, and connecting it to the existing TPN. In this paper, we assume the first two actions have been accomplished using symbolic evaluation, dataflow analysis, or some other technique; our focus is on the actual representation and inclusion of the variable value information.

For an example of when this technique is useful, consider the program called *data* in Appendix 1. BranchCond is set to true in caller1, so caller1 makes the entry call on entry1; the entry call on entry2 is impossible, based on the value of BranchCond. Thus, if we modify our TPN to include information about values of the variable BranchCond, we can improve the accuracy of our analysis by eliminating consideration of the entry call on entry2.

We represent a variable in the program with a *variable subnet*, which contains two kinds of places: value places and operation places. The variable subnet includes a value place for each possible value of the variable, plus an "Unknown" place to account for those occasions on which we can not statically determine the variable value. The variable subnet for a Boolean variable would thus have a "True" place, a "False" place, and an "Unknown" place. Balbo et al. [BDF92] suggest an alternate approach in which each variable has a single place, with the variable's value represented by the number of tokens in the place; we choose not to use this approach because it yields an unsafe net. The variable subnet that we use also includes operation places for the valid operations on a variable of the given type; for example, the valid operations on a boolean variable are "Assign True", "Assign False", and "Not". For each operation, we connect the corresponding operation place to transitions between the appropriate value places. For example, the Boolean variable subnet contains a transition with "Assign True" and "False" as inputs and "True" as an output. The variable subnet is effectively a finite state machine for the variable, with transitions between the states (values) of the variable controlled by operations on the variable.

To make the resulting subnet safe, we modify the TPN to ensure the operation places can never contain more than one token; the modifications described below are based on a transformation described by Peterson [Pet81]. For every operation place for the variable, we add an operation prime place, yielding two places for each possible operation on the variable. For each transition with an operation place as an output, we add the corresponding operation prime place as an input. For each transition with an operation place as an input, we add the corresponding operation prime place as an output. This transformation yields a safe subnet, with the additional property that only one of the operation place/operation prime place pair for a given operation can be marked at any given time. We also note that, since it is possible for the program to exit a TIG region in which the value of a variable is statically determinable into a TIG region in which the value is not statically determinable, we need to provide an "Assign Unknown" operation as well. The resulting variable subnet for a Boolean variable is as shown in Figure 12 below; note that the subnet shown is not yet connected to a TPN.

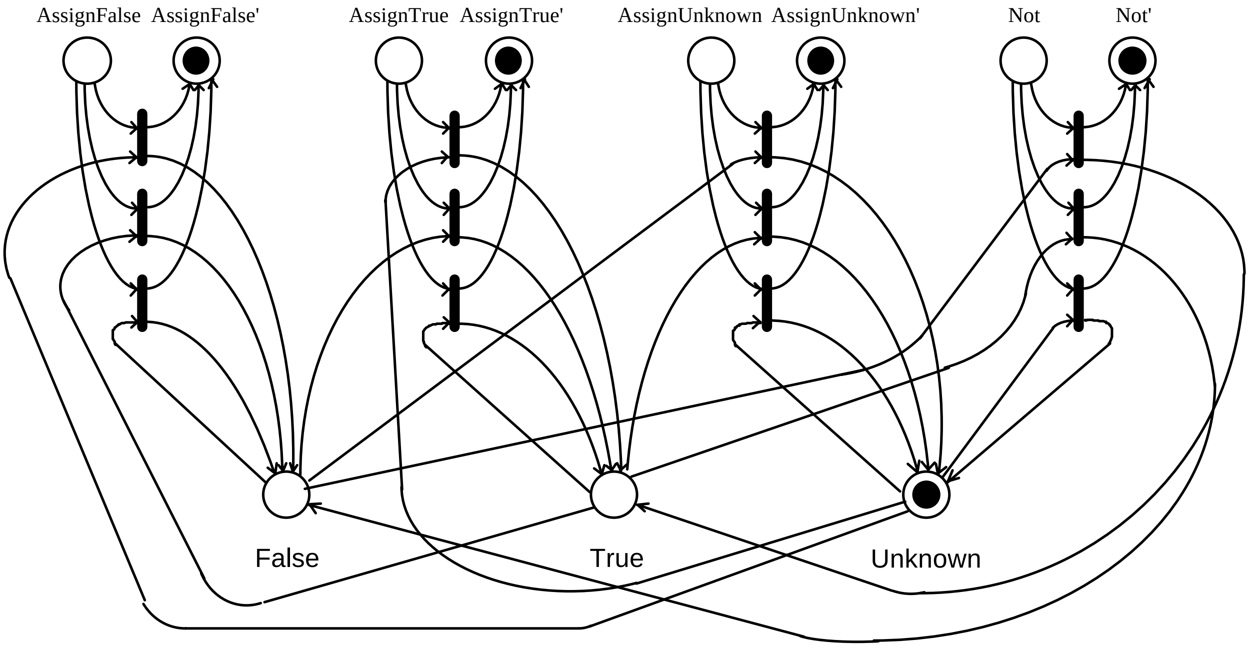

Figure 12. Boolean Variable Subnet

We now consider the cost of building variable subnets; an algorithm for building a Boolean variable subnet is as follows:

Algorithm 7

1 create three value places for three possible values: True, False, and Unknown
2 create four operation places for possible operations on the variable: Assign True, Assign False, Assign Unknown, Not
3 create four operation prime places for the possible operations on the variable
4 create 12 transitions to effect changes in the variable values based on operations to be performed on the variable
5 connect value, operation, and operation prime places to the transitions as shown in figure 12

Each of the steps above takes constant time, so building the variable subnet for a Boolean variable takes constant time. To build a variable subnet for an enumerated type, for example, we first build value places for each possible variable value; this takes O(number of variable values) time. We then add operation and operation prime places for each possible operation on the variable. These operations include assigning each value and taking the successor or predecessor of the current value, so this takes O(number of variable values) time. Finally, we build all the transitions to apply the operations to the variable values. Since each operation needs a transition from each variable value to the target value, this takes O(number of valid operations * number of variable values) time. The total time to build a variable subnet for an enumerated type is thus O(number of valid operations * number of variable values).

To use the additional information provided by the variable subnet to improve the accuracy of our static analysis, we need to connect the variable subnet to the existing TPN. These connections occur in three places: at transitions controlled by a variable, at transitions leading into places corresponding to TIG regions that perform an operation on a variable, and at transitions leading out of places corresponding to TIG regions that perform an operation on a variable. A transition is controlled by a variable if the transition can only occur if the variable has a certain value; in this case, we copy the transition. The appropriate value place for the variable is connected as an input to the original transition, and the same value place is connected as an output of the transition to preserve the value of the variable. We add the Unknown value place as an input and output for the duplicated transition to represent the fact that the interaction may be possible in the case where the variable's value is currently undetermined. In addition, we add all operation prime places for the variable as inputs and outputs for the original and duplicate transitions to ensure the variable's value has stabilized before we use it. In this manner, we exclude all markings from the reachability graph that include firing this transition when the variable does not have the required value, thereby improving the accuracy of the analysis. To effect changes to the variable values, we need to account for TIG regions from the program (places in the TPN) in which the variable is changed (by assignment, for instance); we call these regions *modifying regions*. For each of these regions, we add the appropriate operation place as an output and the corresponding operation prime place as an input of all transitions leading into the modifying region; this initiates modification of the variable on entry into the modifying region. We also add the operation prime place as an input and output of all transitions exiting the modifying region; because the operation prime place will not be marked until the operation on the variable is completed, this ensures the modification is complete before the program exits the modifying region. We note here that a single TIG region can potentially modify a given variable in several different ways; our technique does not currently handle this special case, and we assume that each TIG region performs no more than one operation on each variable represented by a variable subnet.

We have already considered the cost of building a variable subnet; we now consider the cost of connecting a variable subnet to the original TPN as described above. An algorithm to do so is as follows:

Algorithm 8

```
for each transition in the original TPN
    if the transition is controlled by the variable
        add the operation prime places for the variable as inputs and outputs of the transition
        copy the transition
        add the appropriate value place as an input and output of the original transition
        add the Unknown place as an input and output of the duplicate transition
```

```
7       if the transition has a place corresponding to a modifying region as an output
8               add the operation place for the operation performed by the region as an output of
                        the transition
9               add the corresponding operation prime place as an input of the transition
10      if the transition has a place corresponding to a modifying region as an input
11              add the operation prime place for the operation performed by the region as an
                        input and output of the transition
```

Steps 2-3 and 5-11 take constant time. Because duplicating a transition involves creating a new transition and copying all the inputs and outputs from the old transition to the new one, Step 4 takes O(number of input places per transition + number of output places per transition) time. The loop from Step 1 executes (number of transitions in original TPN) times, so the complexity of this algorithm is O(number of transitions in original TPN). Thus, the total cost of building a variable subnet and connecting it to the original TPN is O(number of valid operations * number of variable values + number of transitions in original TPN).

As for impossible pairs, we must also consider potential additional costs incurred generating the reachability graph. We add transitions in this technique when a transition is controlled by a variable; an increase in the number of transitions in the net may increase the time it takes to generate the corresponding reachability graph. If the resulting graph is smaller, the time to generate the graph may still increase, but the additional initial generation time may be worthwhile if several properties are checked on the graph. In any case, the resulting graph is more accurate, so the additional generation time can be considered to be the cost of adding accuracy.

Including variable subnets not only affects the time to generate the reachability graph, it affects the size of the graph as well, both in terms of nodes and arcs. The number of nodes can increase or decrease, based on the program being analyzed, and the additional transitions in the TPN can lead to an increase in the number of arcs in the reachability graph. We include both node and arc counts in our experimental results below to indicate how this technique affects the reachability graph for the programs shown.

Because adding transitions to the TPN can increase both the size of the reachability graph and the time it takes to generate the graph, we must carefully examine our technique in the context of the number of transitions added. Our variable subnets use operation places to ensure the additional representation in the TPN does not increase the number of transitions in the TPN significantly. Because a variable can only have a single value at any given time, when we assign a new value to the variable we need a transition from each of the old values to the new value. If we provided these transitions for each of the transitions into a modifying region, in the worst case the number of transitions added would be (number of places corresponding to TIG regions * number of variable values). Using operation places, we only add (number of valid operations on variable * number of variable values) transitions to the TPN. Since the number of possible operations on a represented variable is generally much less than the number of TIG regions represented by places in the TPN, this is a more efficient means of effecting changes to the variable values.

It is important to note that the new Petri net continues to be safe.

Theorem 5. TPNs including variable subnets are safe.

Proof. For each transition, we must consider each case below. Note that if the transition is within the variable subnet, only Case I applies; if the transition is not within the variable subnet, Case II, III, and/or IV may apply. Cases II, III, and IV are not disjoint, since a transition could be controlled by the variable, lead into places corresponding to modifying regions, and lead out of places corresponding to modifying regions.

Case I: The transition is within the variable subnet. Each transition within the variable subnet has an operation place and a value place as input, and an operation prime place and a value place as output. Since only one of the operation place/operation prime place pair can be marked at any given time, the transition can only fire if the operation prime place does not contain a token (and the operation place does); the operation prime place thus can not contain more than one token. We build the variable subnet so that only one value place can be marked at a time. If firing the transition changes the variable's value, the new value must be unmarked since the old value is marked. If firing the transition gives the variable the same value as it had before firing the transition, the value place is both an input and an output of the transition; the token is removed from the value place, then placed in the same value place by the transition. Each value place can thus contain no more than one token, so the transitions within the variable subnet can not make the TPN unsafe.

Case II: The transition is controlled by the variable. For each transition controlled by a variable value, we add the appropriate value place as both an input and output for the transition. Since firing the transition removes the token from the value place before placing the token back into the value place, firing this transition can not cause the TPN to be unsafe.

Case III: The transition leads into places corresponding to modifying regions. For each transition leading into a place corresponding to a modifying region, we connect the appropriate operation place as an output of the transition and the corresponding operation prime place as an input to the transition. Because the transition is only enabled if the operation prime place is marked, which means the operation place is not marked, placing a token in the operation place by firing this transition can not make the TPN unsafe.

Case IV: The transition leads out of places corresponding to modifying regions. For each transition leading out of a place corresponding to a modifying region, we connect the appropriate operation prime place as an input and output of the transition. Since firing the transition removes the token from the operation prime place before placing the token back into the operation prime place, firing this transition can not cause the TPN to be unsafe.

Since the original TPN was safe and the new and modified transitions can not make the TPN unsafe, the resulting TPN is safe. Q.E.D.

We also note that the resulting TPN is still a conservative Petri net.

Theorem 6. TPNs including variable subnets are conservative Petri nets.

Proof. To clarify the proof, we first consider conservativeness of operation/operation prime place pairs, then we consider the conservativeness of the set of value places for a variable.

Conservativeness of Operation/Operation Prime Place Pairs: At any given time, either an operation place or its associated operation prime place is marked. All transitions with an operation prime place as an output have either the same operation prime place as input or the corresponding operation place as input. If the operation prime place is the input, when the transition is fired the token is removed from the operation prime place and then placed back in the operation prime place. If the operation place is the input, when the transition is fired the token is removed from the operation place and placed in the operation prime place. All transitions with an operation place as an output have the corresponding operation prime place as an input; when the transition is fired, the token is removed from the operation prime place and placed in the operation place. Thus, tokens are never consumed nor generated between each operation place/operation prime place pair, so all transitions with connections to operation places conserve the number of tokens in the pair.

Conservativeness of Value Places: Since we model changes in a variable's values with a transition that has the old value as an input and the new value as an output, value place tokens are never consumed nor generated by the variable subnet transitions; these transitions simply move the token from one value place to another. If a value place is connected as an input to a transition that is not in the variable subnet, that value place is also connected as an output of the transition; these transitions neither consume nor generate value place tokens. Thus, all transitions with connections to variable value places conserve the number of tokens in the value places.

Because the original TPN was a conservative Petri net, and the number of tokens in operation/operation prime pairs and value places remains constant, the resulting TPN is a conservative Petri net as well. Q.E.D.

For an example of the application of this technique, we return to the program named *data* in Appendix 1. The resulting TPN was shown in Figure 3 and the reachability graph was shown in Figure 4. Note that the reachability graph includes a marking corresponding to a deadlock (node 2); this result is spurious, since the interaction represented by Transition 1 can only occur if BranchCond is false, and we can see from the program that BranchCond is true. Using a variable subnet as described above yields the TPN as shown in Figure 13 and the reachability graph shown in Figure 14. Although the reachability graph is somewhat larger, the accuracy is better, since the spurious result is no longer reported. In this case, the two possible interleavings of firing the transition that changes the variable value and firing the transition that is independent of the variable value (between caller2 and accepter) cause an increase in reachability graph size.

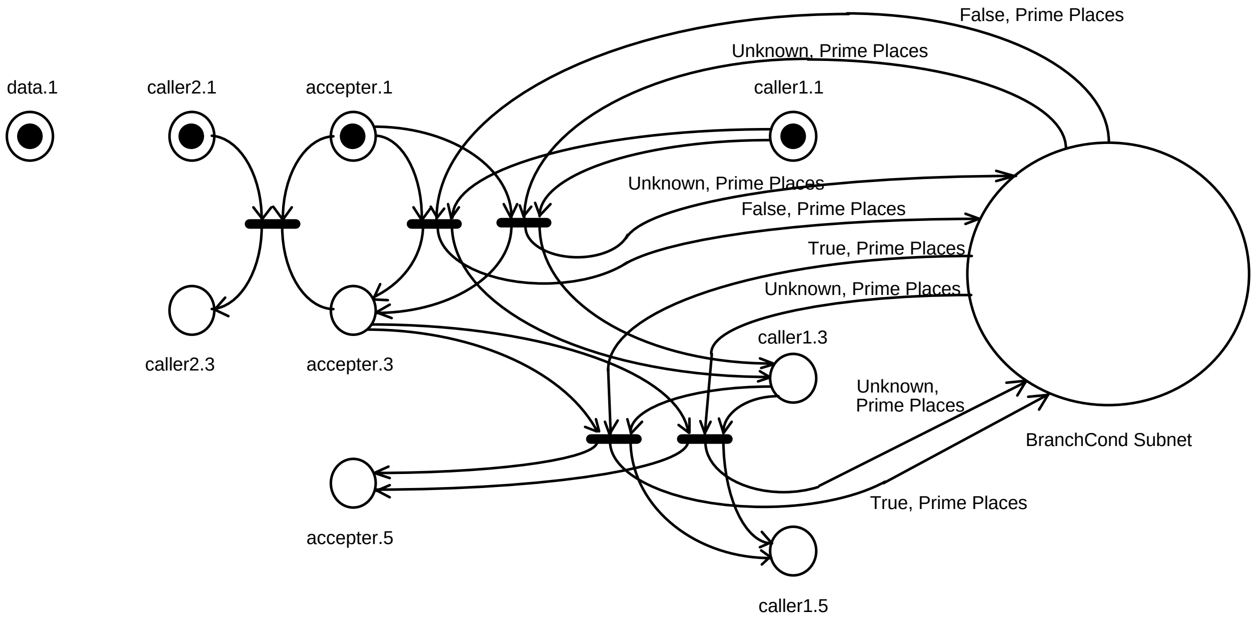

Figure 13. TPN With Variable Subnet Added

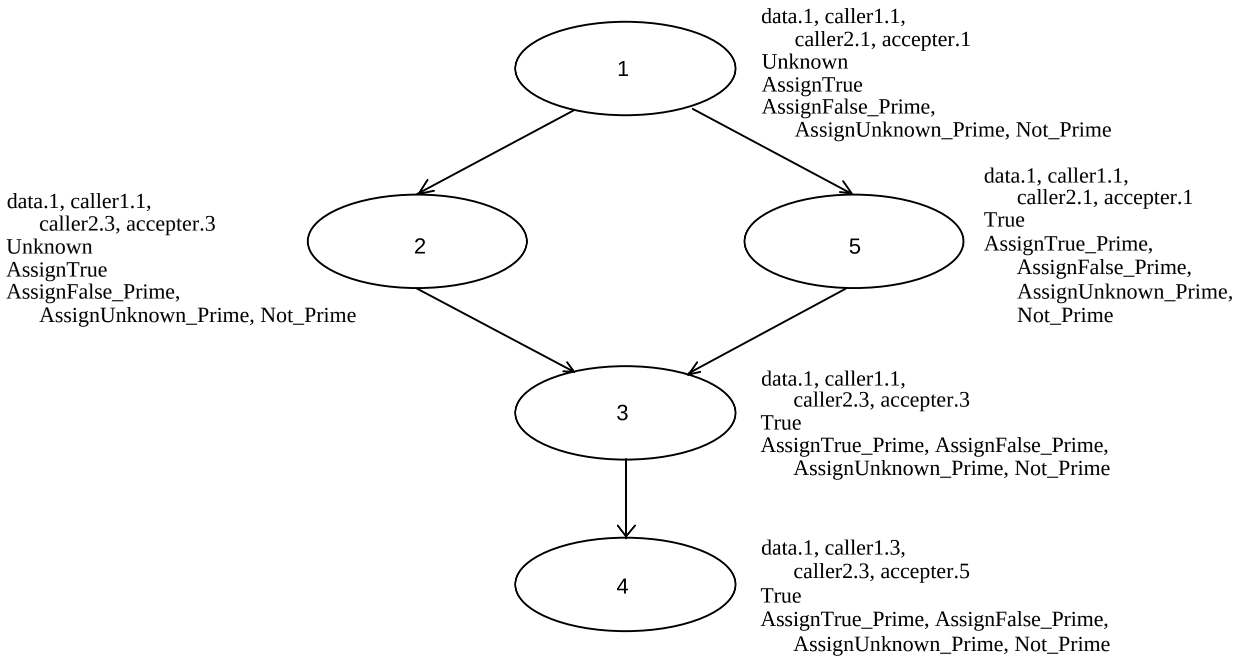

Figure 14. Reachability Graph Using Variable Subnet

While representing variable values in the TPN can lead to improved accuracy, and a smaller reachability graph in some cases, the approach described does have some limitations.

One such limitation is the requirement to be able to statically determine variable values to gain accuracy improvement. By duplicating transitions that are controlled by a variable value as described above, we ensure we don't lose accuracy, because the duplicate transitions are enabled if the value of the variable is not statically determinable. However, firing these transitions is equivalent to firing the corresponding transitions in the original TPN, so if the variable never has a value other than Unknown, the reachability graph using the variable subnet is equivalent to the reachability graph generated from the original TPN. Thus, we only gain accuracy in the reachability graph if, at some point, the value of the variable is statically determinable. It is unclear how often variable values are statically determinable in typical programs, and the accuracy improvements provided by this approach clearly depend on this characteristic of the program being analyzed.

Another limitation of the approach is that, for control flow choices based on complicated conditions, potentially including many variables, it is difficult to determine how to connect the value places to the transitions to accurately reflect the fact that the interaction occurs if and only if the complicated condition holds. This problem is partially addressed in the impossible pairs approach described above, but does not seem to be generally solvable using variable subnets.

Finally, the size of the variable subnet grows as a function of the number of values the represented variable can have. Thus, for Boolean variables, or enumerated types with a small number of possible values, the variable subnets are of reasonable size. However, since the size of the subnet grows rapidly as the number of variable values increases, it may be too costly to represent variables with a large number of possible values. In fact, we can not represent some common types (like Natural or Integer), unless we consider the target machine of the program (32 bit machines might only have Naturals up to 2,147,483,647, for instance), and representing larger types (Float, for instance) is even more problematic. It does seem, however, that we may be able to represent ranges of values of interest for certain large variables. For instance, we may only be interested in whether a Natural variable is less than or equal to 10 or greater than 10; in this case, it may be possible to represent it as two "values", <= 10 and > 10. It's not clear whether this approach will be of much benefit, but it provides one possible way to represent larger ranges of values with variable subnets.

A benefit of this approach is that efficient algorithms for recognizing the regions that affect a variable's value are available. If the control flow decisions on that variable are not extremely complicated, recognizing the transitions controlled by the variable value and making the appropriate connections is also feasible. The additional information added to the TPN is based on the variable type rather than the original TPN, so the variable subnet for a variable with relatively few values, used in relatively few locations, does not increase the TPN size significantly.

Considering Interaction Structure of the Program

As noted in Section 1, aliasing can contribute to static analysis inaccuracies. We recognize that aliasing can cause a variety of problems for static analysis techniques; however, the specific problem we address in this section is trying to determine the array elements potentially affected by an operation, even when the array index is not statically determinable. In some cases, we may be able to reduce spurious results caused by array indexing by more accurately representing the interaction structure of the program in question. As an example of a potential application of this technique, consider a program consisting of an array of tasks; each time an interaction is possible with an element of the array, the standard approach to aliasing would include interactions with all elements of the array in the program representation. It may be the case, however, that there is some statically determinable information about the interactions between array elements that can be used to reduce the size of the representation.

There are two activities to be considered when we apply this technique: determining the valid interactions between array elements and using this information to reduce the TPN. In this paper, we assume the first action has been accomplished using symbolic evaluation, dataflow analysis, or some other technique; our focus is on the actual reduction of the TPN using this information.

A program amenable to this approach (called *archproj*) is included in Appendix 1. The program is a reduced version of a simulation of a systolic array that determines pixel visibility based on light direction for each pixel in a screen. In the example analyzed here, each task (element of the array) is provided with an ID, the upper left element is sent the light direction, which then propagates through the array, and then each task is sent a stop signal. The critical point is that each array element calculates its neighbors on the left, right, top, and bottom; these are the only array elements with which this element can communicate. In addition, this information can be determined at compile time, so we can use it to reduce our program representation.

In general, to use this approach we first build a TPN assuming each element can communicate with all other elements; this treats each operation on (interaction with) an array element as an operation on (interaction with) all elements in the array, a standard approach to account for aliasing. A general algorithm to reduce the TPN based on element interaction information is as follows:

Algorithm 9

```
1  for each transition in the TPN
2      if the transition represents an impossible interaction between array elements
3          delete the transition
```

Since the transitions in our TPNs carry sufficient information to do the check in Step 2 in constant time and Step 3 can also be accomplished in constant time, the algorithm executes in O(number of transitions in TPN) time. Note that the reduction could instead be performed during TPN construction, with a new transition created only if the transition represents a possible interaction between array elements.

We have noted above that including information about impossible pairs or variable values can increase the size of the reachability graph (in terms of nodes and arcs) or the generation time for the graph. Application of this reduction can not result in an increase in size or generation time, since the number of transitions in the system stays the same or decreases. Since each enabled transition from a given reachable state creates a new arc in the reachability graph, reducing the number of transitions can only reduce the number of arcs in the reachability graph. If all the transitions removed by this optimization are unreachable, the reachability graph stays the same as before, though it takes less time to generate because of the reduced number of transitions.

The TPN resulting from this reduction is both safe and conservative. Because the original TPN was safe and removing transitions can not add tokens to places in the net, the resulting TPN is safe. Because the original TPN was conservative and removing transitions can not cause tokens to be generated or consumed, the resulting TPN is conservative.

We now return to our example. To apply the technique to the *archproj* program, in Step 2 of the algorithm above we use the fact that each array element can only communicate with its left, right, top, or bottom neighbors in the array. We should note that our tool set does not currently build a TPN for an array of task types, so we use a program that exhibits comparable behavior; empirical results for several array sizes are included in Section 6.

With this technique we are attempting to reduce the impact of aliasing by using additional statically-determinable information to improve accuracy; we do this by reducing the size of our program representation by removing impossible array element interactions. The approach is generally applicable whenever valid array element interactions can be statically determined. This includes arrays of tasks representing processors connected using any static interconnection network (a network of non-reconfigurable links between processors), including rings, trees, meshes, and hypercubes [Hwa93].

6 Empirical Results

We have run experiments on a small set of programs for two purposes. In Section 4, we commented that in some cases average case algorithm complexity may be a more useful measure of algorithm applicability than worst case complexity; one goal of these experiments was to gather some empirical data about the actual algorithm costs for these examples. In Section 5, we introduced three techniques for improving the accuracy of our static analysis; the other goal of these experiments was to gather information about how the application of these techniques affects the TPNs and reachability graphs for these examples.

To execute the experiments below we use an existing tool set, which we have modified to provide some additional capabilities. Tools to convert an Ada program to a CFG, a CFG to a TIG, and a set of TIGs to a TPN were already available. We have developed a general tool to generate the reachability graph from a TPN. We have also built several specialized tools to include impossible pair information, variable subnets, and interaction structure consideration for the specific problems below.

In the next subsection we briefly discuss the three types of programs included in the experiments. In the second subsection we examine actual algorithm costs for generating our program representations without including any of the techniques introduced in Section 5. In the final subsection, we discuss our approach for implementing those techniques in the experiments and discuss the effect of these techniques on the TPNs and reachability graphs of the programs represented.

Programs Included in the Experiments

For the experiments described here, we use various sizes of the readers/writers problem, the gas station problem, and the *archproj* program in Appendix 1.

The notation rwXY indicates an instance of the readers/writers problem with X readers and Y writers; the code for readers/writers programs is fairly standard, with a Boolean variable WriterPresent used to track the presence of a writer. We have included a copy of the code for rw21 in Appendix 1.

The notation gasXY indicates an instance of the standard gas station problem with X customers and Y pump(s). We have included a copy of the code for gas31 in Appendix 1.

The notation archproj2x2 indicates an instance of the archproj program shown in Appendix 1, with a 2x2 array of processor tasks.

Average Case vs Worst Case Complexity

In this subsection, we consider the actual costs of the algorithms used to generate our program representations and contrast those costs with the worst case costs for the algorithms. We could use the actual costs below to estimate average costs for the algorithms, though formulating a function for the average cost in terms of CFG nodes could be difficult. We therefore simply use the actual costs as an indicator of the relationship between average and worst case costs, without actually generating the average cost functions.

To generate a TIG from a CFG, we use Algorithms 1 and 2 from Section 4. The actual and worst case costs for Algorithm 1, which creates the TIG nodes and edges from the CFG, are the same, since the algorithm always iterates over all CFG nodes. In contrast, the actual and worst case costs for Algorithm 2, which connects the TIG nodes and edges into a graph, tended to differ on our examples, in most cases significantly. We provide a comparison of these results in Figure 15 below. The rows in the table correspond to the tasks in each of the example programs. Note that when we have multiple identical tasks in a program (reader_1 and reader_2 in rw21, for example), we only provide numbers for one of these tasks. The columns in the table correspond to the values for the dominant factors in the algorithm; both average case and worst case numbers would be multiplied by some constant c to yield total cost, so this constant is not included in the table.

	Number CFG Nodes	Actual Cost	Worst Case Cost
rwXY tasks			
rw	4	1	1
reader_1	6	5	216
writer_1	6	5	216
read_write_control	22	17	10,648
gasXY tasks			
activate_tasks	4	1	1
pump	12	9	1,728
customer_1	10	9	1,000
operator (gas31)	23	25	12,167
archproj tasks			
archproj (2x2)	13	41	2,197
proc1 (2x2)	28	30	21,952

Figure 15. Actual vs Worst Case Cost for Algorithm 2

The figure above shows a significant difference, in most cases, between the actual and worst case costs of the algorithm used to complete building a TIG from the CFG. At least for these examples the actual cost, which may be indicative of the average cost, is a better measure of the applicability of the algorithm.

To build a TPN for the set of TIGs in a program, we use Algorithm 4 from Section 4. Because we always iterate over all TIG nodes and TIG edges, the actual and worst case costs for this algorithm are the same.

To generate a reachability graph from a TPN, we use Algorithm 5 from Section 4. We include a comparison of actual and worst case costs in Figure 16.

	Reachability Graph Nodes	Actual Cost	Worst Case Cost
rw21	41	119	80,688
rw22	175	691	2,021,250
rw23	609	3,033	31,154,004
rw32	579	2,883	27,154,521
rw25	6,229	43,603	4,656,052,920
rw52	5,811	40,677	3,748,217,031
gas31	493	986	18,228,675
gas51	9,746	26,802	15,482,476,110
archproj2x2	398	605	87,280,604

Figure 16. Actual vs Worst Case Cost for Algorithm 5

The difference in these costs is significant, and there are two key factors contributing to this difference. The worst case complexity assumes that all reachability graph nodes hash to the same index in a hash table; because we use a Universal hashing scheme, this very rarely happens in practice. The worst case complexity also assumes that all TPN transitions are enabled from any given program state. In the examples above, the average number of enabled transitions per state ranges from 2.9 to 7.0, while the total number of transitions in the TPNs ranges from 48 to 551. The large differences between actual and worst case costs for the algorithm above imply that average cost would be much better than worst case cost as an indicator of algorithm applicability.

Effects of Accuracy Improving Techniques on TPNs and Reachability Graphs

In this subsection, we discuss how each of the accuracy-improving techniques was implemented in the experiments. We then show how using these techniques affects the TPNs and reachability graphs for the programs represented.

For the impossible pair technique, identifying the impossible pairs in the program to be analyzed is done manually. Once we have identified which regions are members of impossible pairs, we provide this information to a tool that scans the transitions in the TPN and automatically modifies the transitions as described in the previous section.

When we use the variable subnet technique, we provide the name of the variable to be modeled to the TPN tool. The tool then automatically generates a variable subnet with the appropriate value and operation places. Currently, we only automatically build Boolean variable subnets, though we have the algorithm to generate subnets for enumerated types as well. We then take the resulting variable subnet and manually connect it to the original TPN by recognizing interactions that are controlled by the variable value and also identifying TIG regions in which an operation is performed on the variable. This activity could be automated by scanning for the variable name in branches and select guards and by collecting information about operations on the variable for each TIG region.

		TPN		Reachability Graph	
Program	Refinement	Places	Transitions	Nodes	Arcs
rw21	N/A	17	48	41	119
	Imp	25	183	31	71
	Var	28	105	52	94
rw22	N/A	20	66	175	692
	Imp	28	306	98	276
	Var	31	138	166	348
rw23	N/A	23	84	609	3,031
	Imp	31	429	248	794
	Var	34	171	426	978
rw32	N/A	23	81	579	2,884
	Imp	31	336	308	1,097
	Var	34	168	502	1,295
rw25	N/A	29	120	6,229	43,571
	Imp	37	675	1,320	4,888
	Var	40	237	2,330	5,908
rw52	N/A	29	111	5,811	40,660
	Imp	33	638	2,972	14,955
	Var	40	228	4,678	16,665
gas31	N/A	39	75	493	987
	Imp	45	111	931	1,773
	Var	87	224	559	885
gas51	N/A	59	163	9,746	26,785
	Imp	64	463	22,841	57,655
archproj2x2	N/A	103	551	398	606
	Struct	103	295	105	154
archproj3x3	N/A	408	9,201	***	***
	Struct	408	2,817	***	***

Figure 17. Effects of Techniques on TPNs and Reachability Graphs

For the interaction structure technique, we manually determine the possible array element interactions as the left/right/top/bottom interactions. We then use this information in a tool that automatically scans the transitions in the TPN, removing those that represent impossible array element interactions. Determining the possible array element interactions can be automated with close examination of the calculation of the left/right/top/bottom neighbors of the elements.

The effects of using these techniques for the programs used can be found in Figure 17. In the figure, N/A means that we simply generate the TPN from the set of TIGs for a program. Imp specifies a TPN that includes information about impossible pairs, Var specifies a TPN that includes one or more variable subnets, and Struct indicates instances in which interaction structure has been used to reduce the TPN.

For the Imp version of the TPN for readers/writers problems, we detect the impossible pairs resulting from whether or not a writer is present and include modeling for these impossible pairs. For the Var version of this problem, we model the WriterPresent variable to reflect the control based on this variable in the guards of the main select statement. Since we model the "same" information, it is valid to directly compare the results of these refinements.

We observe that, for instances of readers/writers larger than rw21, both techniques yield two benefits: they improve the accuracy of the analysis by eliminating consideration of some infeasible paths through the program and they reduce the size of the reachability graph. For rw21, impossible pairs yields a smaller reachability graph, but including the variable subnet increases the size of the reachability graph. As noted in the previous section, this occurs because of the possible interleavings of firing transitions that change the variable value and firing transitions that are independent of the variable value. As the problem is scaled, the affect of these interleavings seems to decrease, and we see reduction in the reachability graph size instead of growth.

For the Imp version of the gas station problems, we use impossible pairs to reflect the fact that if customer1 enters an empty pump queue, then customer1 gets their change before any other customer (and similarly for the other customers). For the Var version of gas31, we implement a variable subnet for each element of the customer queue, in addition to the counter for the number of active customers. Because our tools don't currently automatically build subnets for enumerated or subrange types, we manually built the subnets for this version. For both versions, the reachability graph is more accurate than the original graph.

Including information about impossible pairs in gas31 and gas51 yields reachability graphs with 1.9 and 2.3 times more nodes and 1.8 and 2.2 times more arcs than the original reachability graph, respectively . We note that the impossible pairs information included only accounts for the case when a customer enters an empty pump queue, which occurs a small number of times compared to all the execution possibilities. Including all possible interleavings of this relatively uncommon occurrence with other activities of the system leads to a larger reachability graph to analyze.

For gas31, modeling the customer queue and number of active customers yields a slight increase in the number of reachability graph nodes, so simply checking for a property at each node would take somewhat longer. We note, however, that manually building the variable subnets was tedious. Although building the subnet for each queue element is straightforward and has been described above, the difficulty comes in recognizing where the gas31 code moves the queue forward and representing that movement with the subnets. In any case, the analysis is more accurate, since using the variable subnets ensures that change is always given to the correct customer. We note that, unlike the readers/writers problems, our impossible pairs results are not comparable to the Var version, since we are not capturing the same information in our TPN.

To perform the reduction for the archproj problems, we build the TPN, then scan for transitions that represent impossible communications between array elements. For instance, element (1,1) can only communicate with elements (1,2) and (2,1) in the program; this is statically determinable from the left/right/top/bottom neighbor calculations within the task. We therefore eliminate all transitions with elements (1,1) and (2,2) as the two communicating tasks. Applying this reduction yields a 74% reduction in the size of the reachability graph for the 2x2 instance. We have also generated the TPNs for a 3x3 instance of the problem, though limitations in our storage capacity precluded actually saving the TPNs so it was not possible to generate the corresponding reachability graphs. We do note the almost 70% reduction in the number of transitions in the TPN, however, and hypothesize that the reachability graph would be reduced significantly as well. We also observe that the relative benefits of this approach increase as the problem is scaled, since as the array gets larger, more and more transitions representing invalid communications are removed from the TPN.

As noted in Section 5, the impossible pairs and variable subnet techniques increase the number of transitions in the TPN, thus potentially increasing the time it takes to generate the corresponding reachability graph. The average reachability graph generation takes c*(number of reachability graph nodes * number of transitions in TPN) time; ignoring the constant c, for rw23 (for example) this time is 51,156 with no additional modeling, 72,846 with the variable subnet, and 106,392 including representation of impossible pairs. Thus, it could take approximately twice as long to generate the impossible pairs reachability graph rather than the original reachability graph. We note, however, that the resulting graph is 60% smaller, so if the nodes in the graph will be traversed several times to check various properties, the additional initial time to generate the graph may be worthwhile.

7 Conclusions

Static analysis can be used to answer questions about properties of concurrent programs; by generating the reachability graph for the program being analyzed, we conservatively estimate all possible execution sequences of the program. The cost of building the representations described here range from polynomial in the number of task statements to quadratic in the number of possible program states in the worst case, though we have noted in several cases that average case cost may provide a more realistic indication of the applicability of certain algorithms. The complexity of performing analysis on our representations range from decidable (but not primitive recursive) to linear in the number of program tasks; many of the analysis problems, even under severe restrictions, are NP-complete.

We have identified a variety of methods that can be used to improve the accuracy of static analysis of concurrent programs; in several cases, these methods reduce the size of the reachability graph for the system as well. The impossible pairs technique retains additional program state information in the form of the impossible pair places that are currently disabled because of the program path taken to this point, and the variable subnet technique retains additional program state information in the form of the current value of selected variables. Our third technique uses information about possible array element interactions to modify our program representation to counter at least some of the effects of aliasing.

The cost of using the above techniques can vary considerably from program to program. To effectively use variable subnets, we must first recognize which variables affect the control flow of the program and identify the regions in which those variables are modified. We must also determine how the represented values should be connected to the transitions of the TPN to accurately reflect how the values influence the interactions of the program; the difficulty of doing so ranges from very easy (for control flow decisions based on a Boolean variable's value only, for example) to very difficult (for control flow decisions containing complicated conditions). We can sidestep the difficulty with complicated conditions by including impossible pairs information instead. The complexity of adding the information for the impossible pairs is linear in the number of original transitions in the TPN; the difficulty comes in recognizing the regions of the program that represent impossible pairs. To apply our structural refinement, the array elements with which each array element can interact must be statically determinable; the cost of this approach comes in determining those possible array element interactions by analyzing the appropriate areas of the program. In our example program, the neighbor calculation is simple, so this is not difficult. It is easy, however, to conceive of other programs in which the neighbor calculation is much more troublesome, making the possible element interaction determination more difficult. In any case, performing the refinement once the possible interactions have been determined is linear in the number of original transitions in the TPN.

In several of the programs examined, the reachability graph size or complexity was reduced as a side effect of the improved accuracy. Static analysis models include infeasible as well as feasible paths through the program and must conservatively account for aliasing; the state space which needs to be searched for the property is therefore larger than the actual possible state space of the program. Because our goal was to improve accuracy by eliminating impossible program states from the reachability graph, it is reasonable to expect a smaller reachability graph to result. On the other hand, in some cases our modeling of the additional state information leads to larger graphs, because we add possible interleavings between activities on our variable subnets or Enabled/Disabled places and the original TPN. In all cases, the generated reachability graph represents more accurately the possible states of the program because of the additional information modeled.

Though we have chosen to apply these techniques to generating reachability graphs from TPNs, the ideas can be generalized for use in other static analysis techniques. For instance, an idea similar to variable subnets, called variable automata, is discussed in [DC94]. In general, additional information about a program's execution state could be included in the flow graph used in a data flow problem; this should yield accuracy improvements similar to those observed here. Reducing the effects of aliasing on static analysis accuracy, based on possible interactions of array elements, is a general technique that can be applied before or after construction of the representations of the system in question. While the analysis steps here are specific to our approach, the ideas should be applicable to a variety of static analysis techniques.

Because of various limitations, we have only examined the viability of these techniques on a small sample of programs. To more accurately quantify how well these techniques work in general, more experiments need to be run on a larger sample of programs. Our future plans include performing a series of experiments using these techniques on a wider range of program sizes and complexities.

It would also be interesting to make the tool interactive to determine the effects on analysis accuracy of representing user-supplied information. If the analysis yields spurious results that are not easily eliminated using the above techniques, it may be possible to include additional information from the user to refine the TPN to improve accuracy.

The results above are encouraging, in that they all lead to improvements in the accuracy of our reachability graph, and for some programs reduce the size of the reachability state space as well. Further work needs to be done to more accurately quantify the benefits of these techniques, and the tools should be made more robust to allow additional investigation of these and other techniques for improving static analysis accuracy.

References

[ABC$^+$91] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204-1222, November 1991.

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

[BDF92] Gianfranco Balbo, Susanna Donatelli, and Giuliana Franceschinis. Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171-187, July 1992.

[Cor93] James C. Corbett. Identical tasks and counter variables in an integer programming based approach to verification. In Martin Feather and Axel van Lamsweerd, editors, *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 100-109, Los Alamitos, California, December 1993.

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

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1992.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods - implementations and applications. In Chandrasekaran and Radicchi, editors, *Computer Program Testing*, pages 65-102. North-Holland Publishing Company, 1981.

[Dav73] Martin Davis. Hilbert's tenth problem is unsolvable. *The American Mathematical Monthly*, 80(3):233-269, March 1973.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (to appear).

[DCN94] Matthew B. Dwyer, Lori A. Clarke, and Kari A. Nies. A compact petri net representation for concurrent programs. Technical Report TR 94-46, University of Massachusetts, Amherst, 1994.

[GMO76] Harold N. Gabow, Shachindra N. Maheshwari, and Leon J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3):227-231, September 1976.

[Gra79] Jan Grabowski. The unsolvability of some petri net language problems. *Information Processing Letters*, 9(2):60-63, August 1979.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, 1977.

[Hol88] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software - Practice and Experience*, 18(2):137-161, February 1988.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.

[Jan87] Matthias Jantzen. Complexity of place/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets : Central Models and Their Properties*, pages 413-434. Springer-Verlag, 1987.

[JLL77] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in petri nets. *Theoretical Computer Science*, 4(3):277-299, June 1977.

[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 PA, May 1989.

[May84] Ernst W. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13(3):441-460, August 1984.

[MM81] Ernst W. Mayr and Albert R. Meyer. The complexity of the finite containment problem for petri nets. *Journal of the ACM*, 28(3):561-576, July 1981.

[MR91] Stephen P. Masticola and Barbara G. Ryder. A model of ada programs for static deadlock detection in polynomial time. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 97-107, May 1991.

[MR93] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the ACM Symposium on Principles and Practices of Parallel Programming (PPOPP)*, 1993.

[Pet77] James L. Peterson. Petri nets. *Computing Surveys*, 9(3):223-252, September 1977.

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

[RS90] John H. Reif and Scott A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1-30, 1990.

[SC88] S.M. Shatz and W.K. Cheng. A petri net framework for automated static analysis of ada tasking behavior. *The Journal of Systems and Software*, 8(5):343-359, December 1988.

[Tay83a] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362-376, May 1983.

[Tay83b] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57-84, 1983.

[TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transaction on Software Engineering*, SE-6(3):265-277, May 1980.

[YT88] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499-1511, October 1988.

[YTL+92] Michal Young, Richard N. Taylor, David L. Levine, Kari Forester, and Debra Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. SERC Technical Report TR-128-P, Purdue University, West Lafayette, Indiana, October 1992.

Appendix 1

impos

```
procedure impos is

   task accepter is
         entry entry1;
      entry entry2;
   end accepter;
   task caller1;
   task caller2;

   task body accepter is
   begin
      accept entry1;
      accept entry2;
   end accepter;

   task body caller1 is
            . . .
   begin
            . . .
      if ComplicatedCond then
         accepter.entry1;
      end if;

      if ComplicatedCond then
         null;
      else
         accepter.entry2;
      end if;
   end caller1;

   task body caller2 is
   begin
      accepter.entry2;
   end caller2;

begin   -- impos
   null;
end impos;
```

data

```
procedure data is

   task accepter is
      entry entry2;
      entry entry1;
   end accepter;
   task caller1;
   task caller2;

   task body accepter is
   begin
      accept entry2;
      accept entry1;
   end accepter;

   task body caller1 is
      BranchCond  : BOOLEAN;
   begin
      BranchCond := true;
      if BranchCond then
         accepter.entry1;
      else
         accepter.entry2;
      end if;
   end caller1;

   task body caller2 is
   begin
      accepter.entry2;
   end caller2;

begin   -- data
   null;
end data;
```

archproj

```
procedure archproj is

   -- type for processors in the system
   task type ProcessorType is
      entry ID (ProcID : in Natural);
      entry ReceiveMessage (m : in Natural);
      entry Stop;
   end ProcessorType;
```

```
-- type for connections from processor
type connection is record
   Row      : Natural;
   Column   : Natural;
end record;

-- constants for row and column sizes
RowSize : constant Natural := 2;
ColumnSize : constant Natural := 2;

-- variables for processor array
type ProcArrayType is array (1..2, 1..2) of ProcessorType;
procarray : ProcArrayType;

-- type for processors in the system
task body ProcessorType is

   keeplooping : boolean := true;

   -- variables for processor-specific configuration
   ProcessorID     : Natural;
   ProcessorRow    : Natural;
   ProcessorColumn : Natural;

   -- variables for connections
   LeftNeighbor    : connection;
   RightNeighbor   : connection;
   TopNeighbor     : connection;
   BottomNeighbor  : connection;

   -- variables for input/output messages
   Message : Natural;

begin

   -- wait for information about my ID
   accept ID (ProcID : in Natural) do

      ProcessorID := ProcID;

   end ID;

   -- calculate neighbors on top, bottom, left, and right
   ProcessorRow := ((ProcessorID - 1) / ColumnSize) + 1;
   ProcessorColumn := (ProcessorID mod ColumnSize);
```

```
if ProcessorColumn = 0 then
   ProcessorColumn := ColumnSize;
end if;

-- calculate top connection
if (ProcessorRow = 1) then
   TopNeighbor.Row  := 0;
   TopNeighbor.Column := 0;
else
   TopNeighbor.Row := ProcessorRow - 1;
   TopNeighbor.Column := ProcessorColumn;
end if;

-- calculate bottom connection
if (ProcessorRow = RowSize) then
   BottomNeighbor.Row := 0;
   BottomNeighbor.Column := 0;
else
   BottomNeighbor.Row := ProcessorRow + 1;
   BottomNeighbor.Column := ProcessorColumn;
end if;

-- calculate left connection
if (ProcessorColumn = 1) then
   LeftNeighbor.Row := 0;
   LeftNeighbor.Column := 0;
else
   LeftNeighbor.Row := ProcessorRow;
   LeftNeighbor.Column := ProcessorColumn - 1;
end if;

-- calculate right connection
if (ProcessorColumn = ColumnSize) then
   RightNeighbor.Row := 0;
   RightNeighbor.Column := 0;
else
   RightNeighbor.Row := ProcessorRow;
   RightNeighbor.Column := ProcessorColumn + 1;
end if;

-- wait for messages (until signalled to stop)
while (keeplooping) loop

   select

      -- accept a message if one is sent
```

```
        accept ReceiveMessage (m : in Natural) do

          -- store the message for later processing
          Message := m;

        -- end the rendezvous
        end ReceiveMessage;

        -- forward message to the right and down
        if (ProcessorRow = 1) then
          if (RightNeighbor.Row /= 0) then
            procarray (RightNeighbor.Row, RightNeighbor.Column).
                                   ReceiveMessage (Message);
          end if;
        end if;

        if (BottomNeighbor.Row /= 0) then
          procarray (BottomNeighbor.Row, BottomNeighbor.Column).
                                  ReceiveMessage (Message);
        end if;


      or

        -- accept stop message
        accept Stop;
        keeplooping := false;

      end select;

    end loop;

  end ProcessorType;


begin   -- archproj

  procarray(1, 1).ID (1);
  procarray(1, 2).ID (2);
  procarray(2, 1).ID (3);
  procarray(2, 2).ID (4);

  -- give the "master" processor (ID: 1) the direction of light
  ProcArray(1,1).ReceiveMessage (30);

  procarray(1, 1).Stop;
  procarray(1, 2).Stop;
  procarray(2, 1).Stop;
  procarray(2, 2).Stop;

end archproj;
```

rw21

```
procedure rw is

  task read_write_control is
    entry start_read;          -- start a read
    entry stop_read;           -- stop a read
    entry start_write;         -- start a write
    entry stop_write;          -- stop a write
  end read_write_control;

  task body read_write_control is
    type NumberOfReaders is (zero,one,two);
    ActiveReaders  : NumberOfReaders := zero;
    WriterPresent  : BOOLEAN;
  begin
    WriterPresent := FALSE;

    -- force a write first
    accept start_write;
    accept stop_write;

    -- infinite loop
    loop

      select

      -- check for no writers present
      when not WriterPresent =>

        -- no writer present, so okay to start a read
        accept start_read;

        -- increment number of readers active
        ActiveReaders := NumberOfReaders'Succ(ActiveReaders);

      or
      not WriterPresent =>
        -- could also accept a stop read when no writer present
        accept stop_read;

        -- decrement the number of active readers
        ActiveReaders := NumberOfReaders'Pred(ActiveReaders);

      -- check for no readers or writers
      or
      when ActiveReaders = zero and not WriterPresent =>

        -- nothing active, so we can start a write
        accept start_write;

        -- flag that there's a writer present
        WriterPresent := TRUE;

      or

        -- can also accept a stop write
        accept stop_write;

        WriterPresent := FALSE;

      end select;
    end loop;
  end read_write_control;

  task reader_1;

  task body reader_1 is
  begin                 -- Reader_1
    loop
      read_write_control.start_read;          -- try to start reading
      read_write_control.stop_read;           -- try to stop reading
    end loop;
  end reader_1;

  task reader_2;

  task body reader_2 is
  begin                 -- Reader_2
    loop
      read_write_control.start_read;          -- try to start reading
      read_write_control.stop_read;           -- try to stop reading
    end loop;
  end reader_2;
```

```
   task writer_1;

   task body writer_1 is
   begin                        -- Writer_1
      loop
         read_write_control.start_write; -- try to start writing
         read_write_control.stop_write;          -- try to stop writing
      end loop;
   end writer_1;

begin   -- rw

   -- all tasks are activated at the begin statement above
   -- need at least one statement in a procedure
   null;

end rw;
```

gas31

```
procedure gas is

task operator is
   entry prepay_1;
   entry prepay_2;
   entry prepay_3;
   entry charge;
end operator;

task pump is
   entry activate;
   entry start_pumping;
   entry stop_pumping;
end pump;

task customer_1 is
   entry change;
end customer_1;

task customer_2 is
   entry change;
end customer_2;

task customer_3 is
```

```
   entry change;
end customer_3;

task body operator is
   ActiveCustomers : Natural := 0;
   CustomerQueue   : Array (1..3) of Natural;
begin               -- Operator

   -- infinite loop
   loop

      select

         -- can accept a prepayment from customer 1
         accept prepay_1 do

            -- increment the number of active customers
            ActiveCustomers := ActiveCustomers+1;

            -- put this customer in the customerqueue
            CustomerQueue(ActiveCustomers) := 1;

            -- check for this as the first customer
            if ActiveCustomers = 1 then

               -- activate the pump
               pump.activate;

            end if;

         end prepay_1;

      or
         -- can accept a prepayment from customer 2
         accept prepay_2 do

            -- increment the number of active customers
            ActiveCustomers := ActiveCustomers+1;

            -- put this customer in the customerqueue
            CustomerQueue(ActiveCustomers) := 2;

            -- check for this as the first customer
            if ActiveCustomers = 1 then

               -- activate the pump
```

```
        pump.activate;

      end if;

    end prepay_2;

  or
    -- can accept a prepayment from customer 3
    accept prepay_3 do

      -- increment the number of active customers
      ActiveCustomers := ActiveCustomers+1;

      -- put this customer in the customerqueue
      CustomerQueue(ActiveCustomers) := 3;

      -- check for this as the first customer
      if ActiveCustomers = 1 then

        -- activate the pump
        pump.activate;

      end if;

    end prepay_3;

  or

    -- can accept charge from the pump
    accept charge;

    -- check for more customers
    if ActiveCustomers > 1 then

      -- more customers, so activate pump again
      pump.activate;

    end if;

    -- give change to current customer (first in queue)
    if CustomerQueue(1) = 1 then
      customer_1.change;
    elsif CustomerQueue(1) = 2 then
      customer_2.change;
    else
      customer_3.change;
      end if;

      -- decrement number of active customers
      ActiveCustomers := ActiveCustomers-1;

      -- check for more customers
      if ActiveCustomers > 0 then

        -- move the line up
        for i in 1..ActiveCustomers loop
          CustomerQueue(i) := CustomerQueue(i+1);
        end loop;

      end if;

    end select;

  end loop;

end operator;


task body pump is
begin              -- Pump
  loop
    accept activate;
    accept start_pumping;
    accept stop_pumping;
    operator.charge;
  end loop;
end pump;

task body customer_1 is
begin              -- Customer_1
  loop
    operator.prepay_1;
    pump.start_pumping;
    pump.stop_pumping;
    accept change;
  end loop;
end customer_1;

task body customer_2 is
begin              -- Customer_2
  loop
    operator.prepay_2;
```

```
      pump.start_pumping;
      pump.stop_pumping;
      accept change;
   end loop;
end customer_2;

task body customer_3 is
begin               -- Customer_3
   loop
      operator.prepay_3;
      pump.start_pumping;
      pump.stop_pumping;
      accept change;
   end loop;
end customer_3;

begin
  null;
end gas;
```