Improving the Accuracy of Petri Net-based Analysis
of Concurrent Programs*

A. T. Chamillard
Lori A. Clarke

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

Abstract

Spurious results are an inherent problem of most static analysis methods. These methods, in an effort to produce conservative results, overestimate the executable behavior of a program. Infeasible paths and imprecise alias resolution are the two causes of such inaccuracies. In this paper we present an approach for improving the accuracy of Petri net-based analysis of concurrent programs by including additional program state information in the Petri net. We present empirical results that demonstrate the improvements in accuracy and, in some cases, the reduction in the search space that result from applying this approach to concurrent Ada programs.

1   Introduction

Developers of concurrent software need cost-effective analysis methods 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 complicated and the number of possible communications is large. One class of methods 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, we would like any static analysis method to be *conservative*; for a given property, the analysis must not overlook cases where the property fails to hold. To ensure conservativeness, methods typically use program representations that overestimate the behavior of the program being analyzed. As a result, these methods may produce *spurious results* -- that is, report that a property fails when in fact the cases in which it fails do not correspond to actual program behaviors. Usually, an analysis method produces a spurious result as a consequence of considering paths that can never be executed in the program (commonly called *infeasible paths*) or of considering aliasing that can never occur in the program. For an example of an infeasible path, consider the program in Figure 1. In the caller2 task, the path through the true branch of the first conditional and the false branch of the second conditional is infeasible, assuming the value of BranchCond does not change between the two conditionals. Infeasible paths are natural phenomena of the internal representations we use for analysis and are usually not indicative of a fault in the code.

This paper presents an approach for improving the accuracy of Petri net-based static analysis methods by eliminating some infeasible paths from consideration. We conjecture a scenario in which an analyst submits a program and property to a static analysis tool and then examines the anomaly report that results from the analysis. Since some of the reported anomalies might be spurious, due to consideration of infeasible paths or imprecise alias resolution, the analyst must examine each anomaly to determine if it is a spurious result or not. If a large number of the results are spurious, weeding these out might overwhelm the analyst, causing results that actually do correspond to erroneous program behavior to be discarded. If the number of spurious results is extremely large, the analyst may lose confidence in the analysis tool altogether and forego using it.

It has been our experience that, after looking at an anomaly report, an analyst easily recognizes certain infeasible paths that are the cause of at least some of the

```
task body caller1 is          task body accepter is         task body caller2 is
begin                         begin                             BranchCond : boolean;
   accepter.entry2;              accept entry1;              begin
end caller1;                     accept entry2;                    ...
                              end accepter;                    if BranchCond then
                                                                  accepter.entry1;
                                                               else
                                                                  null;
                                                               end if;
                                                                  ...
                                                               if BranchCond then
                                                                  null;
                                                               else
                                                                  accepter.entry2;
                                                               end if;
                                                            end caller2;
```

Figure 1. Example Program

spurious results. Early experience with static analysis tools indicated that analysts identified *impossible pairs* of statements after examining anomaly reports. Using information about these impossible pairs to recognize spurious results was shown to be intractable for analyses based on control flow graph representations of a program [GMO76]. The approach presented in this paper for improving accuracy is based on a Petri net model of a concurrent program. We describe how certain kinds of infeasible path information can be effectively captured in this model, improving the accuracy of the analysis results without degrading the performance of the analysis.

Thus, the basic idea is that an analyst would apply the static analysis method to the Petri net model of the program. Through examination of the anomaly report, certain infeasible paths that are causing spurious results to be reported become apparent. The analyst, using our approach, refines the Petri net model of the program with this information and reapplies the analysis. Of course, if the analyst knew of infeasible paths before running the initial analysis, that information could be incorporated immediately. In our experience, however, analysts do not tend to think about infeasible paths until after examining an anomaly report with some obvious spurious results. The new anomaly report typically contains fewer spurious results than the previous report, since the additional information should have eliminated the cause of some inaccuracies. Frequently, the new report is significantly smaller since additional, as yet undetected, spurious results are eliminated as well. This smaller report may not be so overwhelming to evaluate, perhaps allowing the analyst to recognize additional spurious results more easily. The effect is an iterative process in which the analyst examines an anomaly report, adds additional information to the analysis, and reapplies the analysis repeatedly until the desired accuracy is achieved.

Our approach allows the analyst to include selected control and/or data information in the Petri net model of the program. The basic idea is to introduce information about the states that the program being analyzed can enter during execution; this information may be in the form of sequences of program statements or in the form of variable values. Petri nets are used because including additional program state information in the net and using that information to control the transitions in the net is relatively straightforward. We hypothesize that, by including additional program state information in the Petri net, we can generate a more accurate estimate of the program state space. Analysis of this more accurate state space considers fewer infeasible paths, potentially reducing the number of spurious results reported by the analysis and increasing the value of the analysis results.

The following section describes some of the major methods that have been used to perform static analysis of concurrent programs, Section 3 describes the program representations we use to analyze concurrent programs with our approach, and Section 4 explains how we represent certain state information to improve the accuracy of those representations. Section 5 presents our empirical results, and Section 6 offers some conclusions based on those results and some pointers to future work.

2   Related Work

Numerous methods for static analysis of concurrent programs have been proposed. In this section we survey the major methods and describe accuracy-improving approaches that have been suggested for reducing the number of infeasible paths considered by these methods.

Reachability analysis checks whether a selected property, often called the *property of interest*, can occur in a concurrent program by considering all reachable states of the program being analyzed. The set of reachable program states can be generated using a variety of program representations, including flow graphs [Tay83a, YTF+89] and Petri nets [Pet77, SC88, DCN95]. Theoretical results [Tay83b] imply that, in general, the time and space requirements for this method are exponential. Several approaches have been proposed to reduce the number of infeasible paths considered by

reachability analysis. One proposed approach is to combine reachability analysis with symbolic execution to prune infeasible paths from the estimated reachable state space [YT88]. Symbolic execution, however, is an expensive method that can not be guaranteed to determine feasibility. Our approach entails straightforward extensions of the Petri net representation.

Other proposed approaches use program variable value information to exclude some infeasible paths from consideration [BDF92, DBD+94]. These approaches assume that, if a variable value is to be modeled, that value is always statically determinable. This assumption seems overly restrictive in general. In contrast, our variable value technique accounts for regions in which the value is not statically determinable, but can only improve accuracy in regions in which the value is determinable. Additionally, the effect of modeling selected variable values is not quantified in [BDF92] or [DBD+94], while Section 5 below compares the sizes of reachability graphs generated with and without modeling of selected variable values.

Symbolic model checking methods [BCM+90] represent the program state space symbolically rather than explicitly. With this method, the program to be analyzed is modeled using Binary Decision Diagrams (BDDs), and the property of interest is specified by a formula. 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. In addition, the BDD representations can require exponential space in the worst case. Note that these representations are structured to symbolically capture the entire program state, so this method is already as accurate as possible given only compile-time information. In contrast, our approach only adds information as it is needed, thereby limiting the size of the program representation.

The Constrained Expression method [ABC+91] avoids representing the state space of the program altogether. Selected program behavior 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. Including information about certain program variable values [Cor93] in the set of inequalities has been proposed as one way to efficiently provide accurate results. This approach is more limited than the approach we propose here.

Data flow analysis is another method that has been applied to concurrent programs. This method employs polynomial-time algorithms to prove a range of program properties [TO80, RS90, MR91, CK93, DC94]. Infeasible synchronization events can be excluded from consideration by identifying program statements that can not execute concurrently [MR93]. An approach, similar to the approach described here, is being explored where the number of infeasible paths is reduced by including selected information about program paths and program variable values [DC94]. This approach encodes the information with the property, whereas our approach encodes the information in the program representation.

An advantage of our approach and that described in [DC94] is that they provide a flexible means for incrementally including additional program state information to improve the accuracy of the analysis. After examining the anomaly report from an analysis run, an analyst can specify additional information to be included to improve the accuracy of the results as needed. In addition, the analyst can choose whether to represent this additional information in terms of control or data information, depending on which representation is best suited to the situation at hand.

3 Program Representations

Because Ada is one of the few commonly used languages supporting concurrency, we use Ada examples to explain our static analysis method and our accuracy-improving approach. The approach, however, is applicable to any language using rendezvous-style communication, and could be extended to most other communication styles as well. In Ada programs, potentially concurrent activities occur in *tasks*[1]. 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, then terminates the rendezvous and continues execution.

Our static analysis method builds upon a variety of internal representations of a concurrent Ada program to capture information about the program. First, we represent each task with a Task Interaction Graph (TIG) [LC89], which abstracts 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 then combine the set of TIGs for all the tasks in a program into a Petri net [DCN95] to model the system as a whole. Finally, we use the Petri net to generate a reachability graph to represent an estimate of all states the program can enter when started in the initial

---

[1]Concurrent activities in Ada programs can also occur in procedures; for simplicity, we call them tasks in this paper.

program state. Petri nets and reachability graphs are central to the techniques we use for improving accuracy, so these representations are described more fully below.

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, where 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

termination points for a task are represented with double circles. For example, the caller2 task could potentially terminate at place 6 (by taking the false branch of the first conditional and the true branch of the second), place 7 (by taking the true branch of both conditionals), or place 8 (by taking the true branch of the first conditional and the false branch of the second). We use TIG-based Petri Nets (TPNs) because it has been shown that TPNs substantially reduce the size of the Petri net, thereby increasing the size of the programs that can be successfully analyzed [DCN95]. Although this example is small, in general Petri nets can be extremely complex and are not usually visualized.
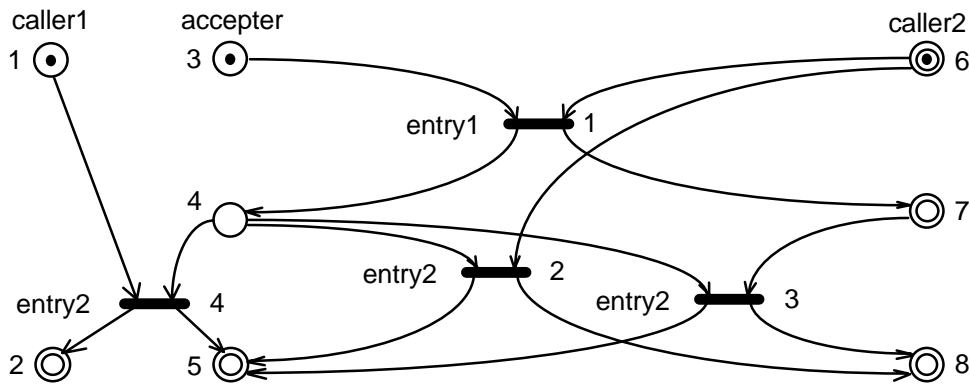

Figure 2. Petri Net

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 nets appear to be a valuable representation for modeling concurrent software [SC88]. In our analysis method, we use a Petri net representation generated from the set of TIGs for the concurrent program. Each place in the Petri net corresponds to a sequential region of code in one of the tasks in the program, and each transition represents a possible interaction (entry call/accept) between two tasks in the program. For an example Petri net, based on the TIGs generated for the program in Figure 1, see Figure 2. In Figure 2, the places representing a task's states are displayed in a column under the task name and each transition, which represents an inter-task communication, is displayed between the two interacting tasks[2]. Places that represent potential

A Petri net is called *safe* if each place in the Petri net can contain at most one token. Safety is a desirable property, because safe Petri nets are guaranteed to have a finite number of reachable states. It has been shown that TPNs are safe [Cha95].

Reachability Graphs

Often, developers want to determine whether or not the concurrent program being analyzed could potentially enter a state in which a specified property is violated; for instance, is it possible for the program to enter a state in which it deadlocks. One method for 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}. 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

accepter and caller2 tasks for the entry2 entry. Transition 2 represents the interaction occurring after caller2 takes the false branch in the first conditional and transition 3 represents the interaction occurring after caller2 takes the true branch in the first conditional.

---

[2]Because of the optimized representation used in a TIG, two transitions are used to represent the interaction between the

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 Petri net simply represents the states of all the tasks being modeled by the Petri net; we therefore consider nodes in the reachability graph as states the program can reach when started from the initial program state. Figure 3 provides the reachability graph for the Petri net in Figure 2. Each node in the figure is annotated with the Petri net places that are marked in the corresponding program state.
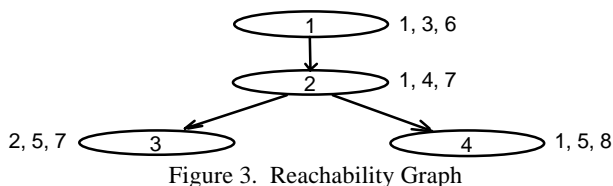

Figure 3. Reachability Graph

4 Improving Accuracy

In this section we examine an approach for improving the accuracy of static analysis without adding significantly to the cost of such analysis. To improve accuracy, we include additional program state information in the Petri net. Although we describe the approach in terms of TPNs, the approach is also applicable to other Petri net representations, such as those from [SC88]. The reachability graph generated from this enhanced Petri net representation provides a more accurate estimate of the program state space than the original reachability graph. Analysis of the revised reachability graph is thus more accurate, and the number of spurious results reported by the analysis should be less than or, in the worst case, the same as the number of spurious results reported for the original reachability graph. Since we propose a scenario where an analyst introduces additional information in response to discovering spurious results in the anomaly report, we would expect the number of such results to decrease. The increase in cost to gain this accuracy improvement includes the cost of incorporating the additional program state information in the Petri net and the cost of analyzing the resulting reachability graph.

Our approach can incorporate additional control flow or data flow information in the Petri net. The first technique, enforcing impossible pairs, retains information about past program states to eliminate some infeasible paths from consideration by the analysis; this technique may be suitable when conditionals are controlled by complicated conditions or when interactions between certain program statements are easily recognized by the analyst. The second technique, representing variable values, eliminates some infeasible paths by modeling variable values. This technique is suitable when conditionals are controlled by a small number of boolean or enumerated variables. We would expect an analyst to select the technique that seems most appropriate or natural for the problem at hand.

For either technique, it is important that the enhanced Petri net continue to be an accurate representation of the program under analysis; in other words, adding the additional control or data information must not hide errors that would have been exposed through analysis of the original Petri net. Although not presented here, to ensure our techniques are error-preserving we have verified that the new Petri net is still an accurate representation of the program. Since the new Petri net is actually a more accurate representation than the original Petri net, it can be shown that the only program states removed from the reachability graph are those that are reached through infeasible paths.

Enforcing Impossible Pairs

Impossible pairs [GMO76] are pairs of program statements that can not both execute in the same execution of the program. In the mid-seventies, impossible pairs were recognized as an intuitive concept that developers could potentially exploit to improve the accuracy of their results. It was demonstrated in [GMO76], however, that deciding whether or not a path exists that does not include any impossible pairs is an NP-complete problem. Rather than explicitly solving the above problem to improve accuracy, we implicitly remove some infeasible paths from consideration by adding information about impossible pairs to the Petri net.

In this paper, we use a less restrictive definition of impossible pairs than the one given in [GMO76], since we believe our definition more accurately captures the restriction that an analyst would want to include. In our definition, executing the first member of the impossible pair inhibits execution of the second member, but executing the second member of the impossible pair has no impact on the executability of the first member[3]. In an extension of our technique, we also account for cases in which the second member of an impossible pair should only be disabled temporarily; this can occur if the condition that causes the second member to be disabled

---

[3]Of course, using our definition an analyst could represent two statements a and b as an impossible pair as described in [GMO76] by specifying two impossible pairs, [a,b] and [b,a].

can subsequently change. Finally, we restrict our attention here to cases in which the impossible pair consists of two interaction (entry call or accept) statements, since the majority of concurrency analysis is concerned with communication events.

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, whereas CHT analysis is concerned with identifying statements that can not execute concurrently.

The technique described below involves representing additional program state information to eliminate infeasible paths that contain both members of an impossible pair. For an example of when this technique is useful, consider the program in Figure 1, and assume for the moment that the conditions in the if statements are much more complicated than the value of a boolean variable. If the condition in the first conditional in the caller2 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 the condition does not change. Note that, similar to symbolic model checking, we could try to encode the possible values of the complicated condition in the Petri

we assume that these are relatively easy for an analyst to manually identify after examining the anomaly report. We would expect that after discovering several spurious results in the report, the analyst would introduce specific impossible pair information to improve the accuracy of the results. In any case, for this presentation we assume that some method 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 Petri net.

To simplify our explanation, we assume a single impossible pair in the program but note that the technique can be extended to multiple impossible pairs [Cha95]. Also note that, using the same basic technique, more complicated flow constraints than impossible pairs could be incorporated given Petri net representations of those constraints.

To illustrate the ideas presented here, we modify the Petri net given in Figure 2. Transition 1, which corresponds to the accepter.entry1 statement in the caller2 task, is the first member of the impossible pair. Transitions 2 and 3, which correspond to the accepter.entry2 statement in the caller2 task, represent the second member of the impossible pair. The enhanced Petri net is shown in Figure 4.
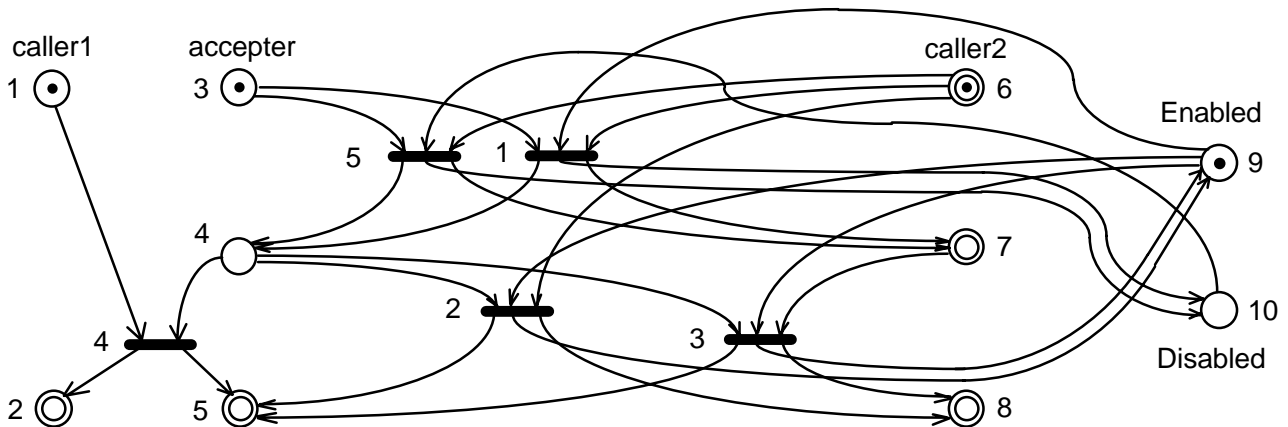

Figure 4. Petri Net With Impossible Pairs Represented

net. For general boolean expressions, however, the encoding of the condition in the Petri net could be quite large. Instead, we use information about this impossible pair to improve the accuracy of the Petri net and the corresponding reachability graph.

There are three distinct activities associated with enforcing impossible pairs: recognizing the impossible pairs in a program, recognizing which regions in the program re-enable second members of the impossible pairs, and including information about the impossible pairs in the Petri net. Although sophisticated methods, such as symbolic evaluation [CR81], could be used to recognize impossible pairs and regions re-enabling them,

In general, to include impossible pair information in our Petri net we add two new places that control firing of the transitions corresponding to the second member of the impossible pair in the program, and also add duplicates of the transitions corresponding to the first member of the impossible pair. The first new place, called the Enabled place for the second member, is used to enable execution of the second member; the second new place, called the Disabled place for the second member, is used to inhibit execution of the second member. 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 Petri net. We connect the Enabled place as an input to 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 (transitions 2 and 3 in Figure 4) . We also connect the Enabled place as an output of these transitions, which lets the task statement execute multiple times. Since executing the first member of the impossible pair prohibits the second member from executing, we must 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 (transition 1 in Figure 4) to change the second member from enabled to disabled when the first member is executed and the duplicate transition (transition 5 in Figure 4) to keep the second member disabled if it is already disabled when the first member is executed; we call these *disabling transitions*.

To ensure that the second member is enabled or disabled (but not both), we have connected 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 (see Figure 4).

In an extension of the technique described above, we also consider the possibility that the second member of an impossible pair should only be disabled temporarily. For example, if the first member of an impossible pair is contained within a loop and the condition is changed at the end of the loop, the second member of the impossible pair 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 Petri net; 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 place 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*. In our example program the second member of the impossible pair is never re-enabled, so these transition modifications are not required for the Petri net in Figure 4.

In our example, the Petri net without impossible pair information is shown in Figure 2, and the corresponding reachability graph is shown in Figure 3. Node 4 in the reachability graph represents a deadlock of the caller1 task. The transition fired to enter this node, however, represents an interaction that is not possible, because the true branch is traversed in the first conditional in the caller2 task to reach node 2, and the condition is not changed before the second conditional. Therefore, an analysis result that reports deadlock for this program is a spurious result, since the program can not actually execute the path required to reach the deadlocked node. Using the technique for impossible pairs described above, we add impossible pairs information to the Petri net as shown in Figure 4; the corresponding reachability graph is shown in Figure 5. Note that in Figure 5 we have retained the reachability graph node numbering from Figure 3 to facilitate comparison. For this example the spurious result has been removed by the additional information included, and thus analysis of the resulting graph can yield more accurate results.
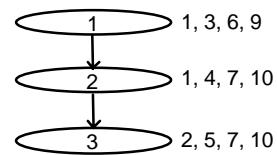

Figure 5. Reachability Graph With Impossible Pairs Represented

Representing Variable Values

When we include representation of impossible pairs information in our Petri net, 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. This technique is applicable when conditions in the program conditionals are relatively simple and include a small number of boolean or enumerated variables whose values can be statically determined in at least some regions of the program. As with the impossible pairs technique, we modify the Petri net 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 this additional information to exclude interactions that are infeasible based on those values, thereby excluding some infeasible paths from our analysis.

For an example of when this technique is useful, consider again the program in Figure 1 and assume that BranchCond is set to true at the beginning of caller2. Thus, caller2 makes the entry call on entry1, but the entry

call on entry2 is impossible, based on the value of BranchCond. If we modify the corresponding Petri net to include information about values of the variable BranchCond, we can improve the accuracy of the reachability analysis by eliminating consideration of the entry call on entry2.

There are four activities to be considered when we represent variable values in a Petri net: 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 Petri net. We believe that this is often straightforward in practice, particularly when a boolean variable is used to control communication in the program. For these cases, an analyst should easily be able to identify such controlling variables and could specify those variables for inclusion in the Petri net. In this paper, we assume the first two actions have been accomplished and focus on the actual representation and inclusion of the variable value information.

We represent a variable in the program for which we want to maintain value information with a *variable subnet*. This subnet contains two kinds of places: value places and operation places. The 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's value. To simplify the presentation, we describe a variable subnet for a boolean variable. The variable subnet for a Boolean variable would have a "True" place, a "False" place, and an "Unknown" place. When the "Unknown" place is marked, the variable could be true or false; based on the connections described below, both possibilities are considered during generation of the reachability graph. The "Unknown" place is marked in the initial marking of the Petri net. The variable subnet 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 Petri net to ensure the operation places can never contain more than one token, using transformations similar to those 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. If none of the regions corresponding to marked places in the initial marking of the original Petri net modify the modeled variable, all operation prime places are marked in the initial marking of the Petri net; otherwise, the appropriate operation places are marked, with the corresponding operation prime places left unmarked. We also note that, since it is possible for the program to exit a region in which the value of a variable is statically determinable into a 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 6, but the subnet shown has not yet been connected to the Petri net for a program.
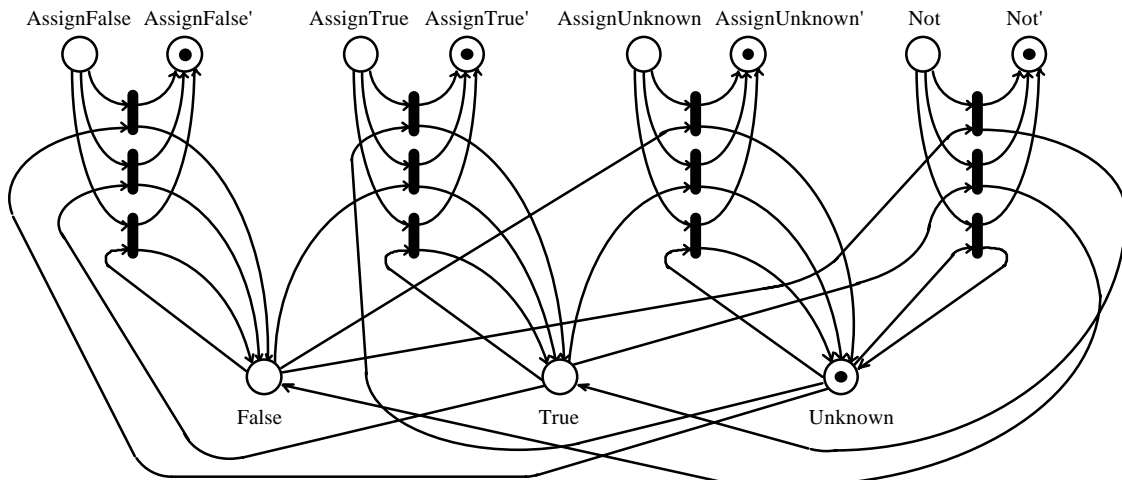

Figure 6. Boolean Variable Subnet

To use the additional information provided by the variable subnet, we need to connect the variable subnet to the Petri net. Figure 7 illustrates the revisions to the Petri net using the example shown in Figures 1 and 2. The variable subnet for the BranchCond variable is abstracted to facilitate understanding. In Figure 7, a T, F, or U on an arc represents a connection to the True, False, or Unknown value place in the BranchCond Subnet. Also, connections between transitions and operation prime places are as described below, but are omitted from this figure for clarity.

modifying regions. If we assign BranchCond the value true initially in the caller2 task then the corresponding place (place 6 in Figure 7) corresponds to a modifying region. 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
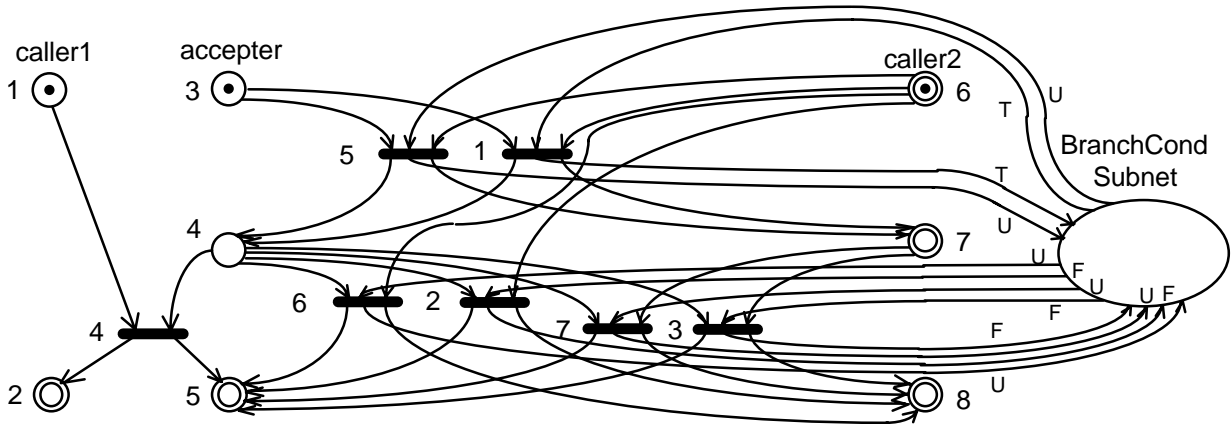

Figure 7. Petri Net With Variable Subnet Added

A variable subnet is connected to the Petri net for a program in two cases: at transitions controlled by the variable and at transitions leading into or out of places corresponding to regions that modify the value of the variable. In the first case, 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 (transitions 1, 2, and 3 in Figure 7), 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 (transitions 5, 6, and 7 in Figure 7) 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 any required modifications to the variable have been completed before we use the variable's value. 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.

In the second case, to effect changes to the variable values, we need to account for regions from the program (places in the Petri net) in which the variable is changed (by assignment, for instance); we call these regions

marked until the operation on the variable is completed, this ensures the modification is complete before the program exits the modifying region. Since the operation prime places have already been added to transitions 1 and 5 as described above, no further changes are required in Figure 7.

Note that a single region can potentially modify a given variable in several different ways. To simplify the description we assume a simpler model here, in which a single region modifies a given variable in one specific way. Note that more complicated modeling can be used to handle the more general case. Also note that since the region represented by place 6 in the Petri net would contain BranchCond := true, in our initial marking the AssignTrue place is marked (and the AssignTrue' place is unmarked).

Using a variable subnet as described above yields the Petri net shown in Figure 7. The corresponding reachability graph is shown in Figure 8, where the reachability graph nodes are annotated with the marked Petri net places as well as the marked value, operation, and operation prime places in the BranchCond Subnet. Again we see that the spurious result is no longer reported.

Information about variable values could also be incorporated using an FSM, with states of the FSM representing variable values and transitions in the FSM representing operations on the variable. While the FSM
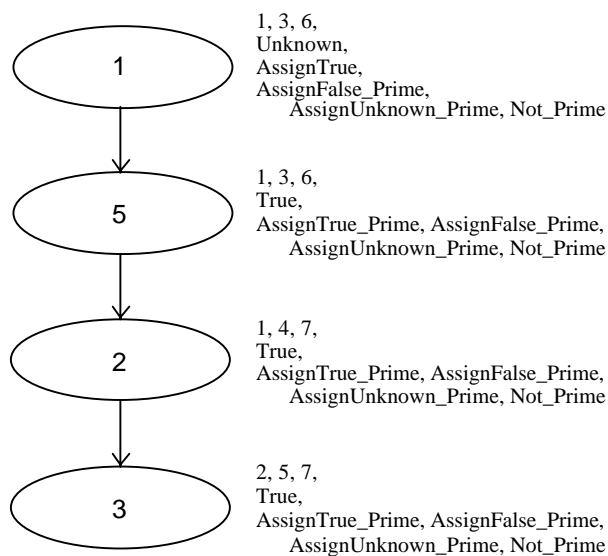
Figure 8. Reachability Graph Using Variable Subnet

would certainly be easier to understand than Figure 6, the difficulty comes when incorporating the FSM into the model. An FSM can not be "connected" to the Petri net as our variable subnets are, so the FSM would need to be used during reachability graph generation, potentially slowing down the generation process significantly. Representing variables with variable subnets provides the same accuracy improvements as would be provided with FSMs, while retaining a standard Petri net as the program model.

Choosing Between the Two Techniques

The two techniques described above give the analyst flexibility when determining what kind of additional information to include to improve analysis accuracy. In general, we expect the analyst to choose whichever technique appears more natural given the program being analyzed and the property of interest.

The impossible pairs technique seems 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. If the control flow decisions in the program are complicated, the impossible pairs technique may be more suitable than the variable values technique. The impossible pairs technique will tend to be expensive for programs for which the Petri net contains transitions that affect multiple members of impossible pairs, since the number of these transitions grows exponentially in the number of impossible pairs affected.

In the variable values technique, efficient algorithms for recognizing the regions that affect a variable's value are available. An analyst may also be able to easily identify those variables that are used in the program to

control communications. If the control flow decisions on those variables are not extremely complicated, recognizing the transitions controlled by the variable values and making the appropriate connections is relatively straightforward. The additional information added to the Petri net is based on the variable type, so the variable subnet for a variable with relatively few values (such as a boolean variable), used in relatively few locations, does not increase the Petri net size significantly. Limitations of this technique include the requirement to be able to statically determine variable values to gain accuracy improvement, the difficulties determining the proper connections to account for complicated conditions, and the rapid growth of the size of the variable subnet as the number of possible values of the represented variable grows.

5 Empirical Results

We have run experiments on a small set of programs to gather information about how the application of our approach affects the sizes of the Petri nets and reachability graphs for these examples. We hypothesize that our accuracy-improving approach can improve analysis accuracy without significantly impacting performance.

In each of the techniques presented, the size of the Petri net is increased by the places and transitions added to model the additional semantic information. On one hand, we expect the size of the reachability graph to grow as the size of the Petri net grows, since the upper bound on the size of the reachability graph is exponential in the number of Petri net places. On the other hand, we would expect the additional modeling in the Petri net to remove some infeasible paths from consideration, thereby reducing the size of the reachability graph. We perform the experiments to acquire preliminary indications of which scenario is more common and also to gain experience applying the approach.

Whenever the approach is applied, the resulting reachability graph more accurately represents the program state space. However, this does not necessarily guarantee that the number of spurious results in the anomaly report will be reduced. For instance, if the states removed from the reachability graph are independent of the property being checked, the number of spurious results in the anomaly report will stay the same. For that reason, we consider our accuracy improvements as improvements in the reachability graph as a representation of the program state space, rather than as reductions in the number of spurious results in the anomaly report. While we expect that improving the accuracy of the reachability graph will commonly reduce the number of spurious results, whether or not this occurs in practice depends on the property being checked.

To perform the experiments below we modified an existing tool set. Tools to convert an Ada program to a TIG and a set of TIGs to a Petri net were already available. We developed a general tool to generate the reachability graph from a Petri net, and also built several specialized tools to include impossible pair information and variable subnets in the Petri net.

For the experiments described here, we used various sizes of the readers/writers problem and the gas station problem. 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. The notation gasXY indicates an instance of the standard gas station problem [HL85] with

resulting variable subnet and manually connect it to the original Petri net by recognizing interactions that are controlled by the variable value and also identifying 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 region.

The effects of using these techniques for the sample programs can be found in Table 1. In the table, NA means that no additional information is included in the Petri net for the program. Imp specifies a Petri net that includes information about impossible pairs and Var specifies a Petri net that includes one or more variable subnets.

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

Table 1. Effects of Approach on Petri Nets and Reachability Graphs

X customers and Y pumps.

For the impossible pair technique, identifying the impossible pairs in the program to be analyzed is done manually. Once we have identified which regions correspond to impossible pairs, we provide this information to a tool that scans the transitions in the Petri net 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 Petri net toolset. The toolset then automatically generates a variable subnet with the appropriate value and operation places. Currently, we only automatically build Boolean variable subnets. We then take the

For the Imp version of the Petri net for readers/writers problems, we model the impossible pairs resulting from whether or not a writer is present. These pairs were easy to recognize given the simple guards in the control task. Including this information improves the accuracy of the analysis by eliminating consideration of some infeasible paths through the program and reduces the size of the reachability graph as well.

For the Imp version of the gas station problems, we use impossible pairs to reflect the fact that if a customer enters an empty pump queue, then that customer gets their change before any other customer. Including information about impossible pairs in gas31 and gas51

yields reachability graphs with approximately twice as many nodes and arcs as the original reachability graph.

Including impossible pairs information in the Petri net can cause an increase in the reachability graph size because we encode not just the current program state, but also information about the path leading to that state. For example, consider the state in which customer 1 and customer 2 have both pre-paid the operator. Without impossible pairs information, this state is represented by a single node in the reachability graph. When we include impossible pairs information, the reachability graph contains one node for this state in which customer 1 entered the (empty) queue first, one node in which customer 2 entered the (empty) queue first, and one state in which neither entered an empty queue. In such cases, the improvement in accuracy comes at the cost of a larger reachability graph to be analyzed.

For the Var version of readers/writers, we model the WriterPresent variable that is included in the guards of the main select statement. Selecting this variable to be modeled and recognizing the appropriate connection points for the variable subnet were straightforward because of the basic operations on the variable and the simplicity of the guards containing the variable. We observe that, for instances of readers/writers larger than rw21, the technique yields two benefits: it improves the accuracy of the analysis by eliminating consideration of some infeasible paths through the program and it reduces the size of the reachability graph. For rw21, this technique increases the size of the reachability graph. 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 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. 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. In addition, we note that manually building the variable subnets was tedious. Although building the subnet for each queue element is straightforward, 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. Developing the model of

the customer queue was sufficiently time-consuming that we did not attempt this for the gas51 program.

For the readers/writers problem, the impossible pairs and variable value techniques implicitly model the "same" information (the value of the WriterPresent variable). It is therefore valid to directly compare the sizes of the resulting reachability graphs (since they have the same accuracy), and to note that the impossible pairs technique is more effective at reducing the size of the graph. On the other hand, the Imp Petri nets contain many more transitions than the Var Petri nets for this problem, so it may take longer to actually generate the (smaller) Imp reachability graphs. With both techniques, the accuracy of the reachability graph is improved; the reduction in size is a beneficial side effect.

For the gas station problem, our impossible pairs results are not comparable to the Var version, since we are not capturing the same information in our Petri net. The Var version captures a significant amount of state information for only a slight increase in reachability graph size, but manually adding the required variable value modeling was difficult. The Imp version captures less information than the Var version, and yields a large increase in reachability graph size, but including the modeling was straightforward.

Table 2 lists several properties of each program considered. Entries is the number of unique entries in the program and Entry Calls is the total number of calls on those entries. Variables provides the number of variables modeled in the Var version of the Petri net, with the number of possible variable values (including unknown) following in parentheses. For instance, for the Var version of the gas31 Petri net, we model 3 variables with 4 possible values and 1 variable with 5 possible values. Impossible Pairs provides the number of impossible pairs modeled in the Imp version of the Petri net. For the readers/writers programs, the numbers of variables and impossible pairs modeled stay constant as the problem is scaled. This occurs because the additional modeling is applied to the control task, which does not change as the problem is scaled. For the gas station problems, the

Program	Entries	Entry Calls	Variables	Impossible Pairs
rw21	4	6	1 (3)	7
rw22	4	8	1 (3)	7
rw23	4	10	1 (3)	7
rw32	4	10	1 (3)	7
rw25	4	14	1 (3)	7
rw52	4	14	1 (3)	7
gas31	10	17	3 (4), 1 (5)	6
gas51	14	27	-	20

Table 2. Program Properties

number of impossible pairs modeled grows as the problem is scaled because the modeling is applied in the operator task, which grows as the problem size grows.

6 Conclusions

Static analysis can be used to answer questions about properties of concurrent programs, although often with the inclusion of spurious results. We have identified an approach that can be used to improve the accuracy of Petri net-based analysis of concurrent programs. In several cases that we examined, the approach reduced 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 transitions that are currently enabled and disabled, and the variable subnet technique retains additional program state information in the form of the current values of selected variables.

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 Petri net to accurately reflect how the values influence the interactions of the program. The difficulty of doing this 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). Alternatively, we can sometimes account for 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 Petri net; the difficulty comes in recognizing the regions of the program that represent impossible pairs. Ultimately, the decision about which technique to use will fall on the analyst. For some programs, the impossible pairs may be easily recognized by the analyst, whereas for other programs, representing key variables that control communications in the program may seem more straightforward.

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 generally include infeasible as well as feasible paths through the program; 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 impossible pair places and the original Petri net. In all cases, the generated reachability graph represents more accurately the possible states of the program because of the additional information modeled.

We have examined how to incorporate accuracy-improving semantic information into Petri nets. It is not as easy to modify the semantics of other internal representations that are commonly used for analysis, such as control flow graphs, abstract syntax trees, and program dependency graphs. A complementary and somewhat similar approach is explored in [DC94], but instead of modifying the internal representation, the approach incorporates the additional semantic constraints in the analysis algorithms. Similarly, information about impossible pairs or variable values could be incorporated in the reachability graph generation algorithm rather than in the Petri net representation of the program. It is not clear how this would affect the size of the resulting reachability graph, but the added complexity in the algorithm might lead to a significant increase in reachability graph generation time. It is too early to determine when one approach might be superior to the other.

Because of various limitations, we have only demonstrated the viability of our approach on a small sample of programs. It is doubtful, however, that these programs are representative of the population of "real" concurrent 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 this approach on a wider range of program sizes and complexities.

For the programs examined here, we have manually detected variables and impossible pairs to model, then added them to the Petri net using partially automated tools. More support could be provided to the analyst through automatic recognition of variables that control interaction patterns in the program; these variables could then be automatically included in the Petri net or recommended as useful variables to model. Automatically detecting impossible pairs in the program may not be feasible except in simple cases, but further automating the process of modeling variables and impossible pairs is a potential area for future research.

It would also be interesting to make the tool interactive to determine the effects on analysis accuracy of representing other 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 Petri net to improve accuracy. Other constraints on

the control flow, such as sequences of certain statements that can never occur or must always occur, can be modeled with subnets and attached appropriately. More generally, any constraints that can be expressed with a subnet could be used to improve the accuracy of analysis results, as long as the analyst or an enhanced tool could determine how to attach the subnet appropriately. To ensure conservativeness, the modifications would need to be error-preserving, at least for the property being checked.

The results above support our hypothesis that modeling specific kinds of program state information in the Petri net can lead to cost-effective improvements in the accuracy of the corresponding reachability graph, and for some programs reduce the size of the reachable 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.

[Cha95] A.T. Chamillard. Improving static analysis accuracy on concurrent Ada programs: Complexity results and empirical findings. Technical Report TR 95-49, University of Massachusetts, Amherst, 1995.

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

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

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

[DBD+94] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transactions on Software Engineering and Methodology*, 3(4):340-380, October 1994.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62-75, New Orleans, Louisiana, December 1994.

[DCN95] Matthew B. Dwyer, Lori A. Clarke, and Kari A. Nies. A compact petri net representation for concurrent programs. In *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle, Washington, April 1995.

[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.

[HL85] D. Helmbold and D.C. Luckham. Debugging Ada tasking programs. *IEEE Software*, pages 47-57, March 1985.

[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.

[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 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.

[YTF+89] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis and Verification (TAV3)*, pages 200-209, 1989.