An Empirical Comparison of Static Concurrency Analysis Techniques

A.T. Chamillard
Computer Science Department
HQ USAFA/DFCS
USAFA, CO 80840 USA
achamill@cs.usafa.af.mil

Lori A. Clarke
Computer Science Department
University of Massachusetts
Amherst, MA 01003 USA
clarke@cs.umass.edu

George S. Avrunin
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003 USA
avrunin@math.umass.edu

ABSTRACT
This paper reports the results of an empirical comparison of several static analysis tools for evaluating properties of concurrent software and also reports the results of our attempts to build predictive models for each of the tools based on program and property characteristics. Although this area seems well suited to empirical investigation, we encountered a number of significant issues that make designing a sound and unbiased study surprisingly difficult. These experiment design issues are also discussed in this paper.

1 INTRODUCTION

Concurrent systems are inherently more complex and difficult to understand than sequential ones. Thus, it is imperative that cost-effective techniques be developed for checking that the behavior of these systems meets specified requirements. One class of techniques used for analyzing concurrent software is static analysis, where compile-time information is employed to prove properties about a system. A variety of static concurrency analysis techniques have been proposed, including reachability analysis (e.g., [Hol91, GW91, Pel94]), symbolic model checking (e.g., [McM93]), inequality necessary condition analysis (e.g., [CA95]), and data flow analysis (e.g., [DC94]).

We are interested in understanding the relative strengths and weaknesses of these techniques in terms of their ability to prove properties of concurrent programs, such as freedom from deadlock and the mutually exclusive use of resources. Although proving these properties is typically NP-hard and most of the static concurrency analysis techniques have exponential worst case analysis times, each has been shown to be applicable to some interesting problems. Thus, information about average case analysis time and consumption of other types of resources may help differentiate between the techniques in practice. While average case performance is difficult to derive formally, experimentation can help develop estimates for each of the techniques.

In general, we would like any static analysis technique to be *conservative*; for a given property, the analysis must not overlook cases where the property fails to hold. To ensure conservativeness, and in some cases to improve tractability, techniques often overestimate the behavior of the program being analyzed. This overestimate may lead to the tool falsely reporting that the property may be violated. When a tool reports that a property may be violated, but in fact these possible violations do not correspond to actual program behaviors, then this is called a *spurious result*. For example, if the program representation contains paths that can never be executed in the program (commonly called *infeasible paths*), the tool may report that the property fails to hold when it only fails on infeasible paths. Analysis accuracy is therefore also an important measurement to consider.

Previously, Corbett [Cor94, Cor96] reported on empirical studies comparing the performance of several tools in detecting deadlock in Ada tasking programs and we have built upon that work. Our experiment differed from that of Corbett in that we also considered a data flow analysis tool, we examined program-specific properties in addition to deadlock, we considered analysis accuracy and other measures of performance, we carried out careful statistical analysis of the results to check for various biases, and we developed preliminary predictive models for analysis time, rate of failure, and accuracy of analysis.

Our long term goals are to use a variety of static analysis tools, on a large, representative sample of programs and properties, to study the average case performance of those tools and to develop predictive models to help an analyst estimate the analysis time, rate of failure, and accuracy of analysis of each tool given a program and property to be checked. As discussed in this paper, we are still a long way from achieving these goals. Only limited conclusions can be drawn from our empirical comparison, and many of the models developed during our preliminary predictive modeling efforts do not have strong predictive power. On the other hand, our work does provide interesting insights about the tools and about such experimental studies.

The following section discusses the issues we faced and the tradeoffs we had to make while designing the experiment. The third section provides the results of our empirical comparison of the various tools and the fourth section describes our predictive modeling efforts. The final section presents our conclusions and a discussion of future work.

2 EXPERIMENT DESIGN ISSUES AND TRADEOFFS

Researchers have pointed out the limited number of empirical studies in software engineering [TLP+95]. Such studies can be extremely difficult to carry out, however, especially if human subjects are involved and the measures are subjective (e.g., code understanding and effectiveness of design methodologies). From this point of view, a comparison of static concurrency analysis techniques seems like a straightforward empirical study -- there is relatively little human involvement in the experiment, and there is little subjectivity involved in the analysis of the results.

When we designed our experiment we expected to encounter a number of difficulties common in experiment design. For example, while we would like to compare the performance of a number of static analysis techniques on a representative sample of concurrent programs, validating a representative sample of the kinds of properties the developers of such programs would be interested in checking, we do not believe such representative samples can be selected at this time. We would also like to compare the performance of the techniques in a way that would include the full cost of using those techniques, including human resources, but there does not seem to be a straightforward way to measure the human effort involved. In addition to these expected difficulties, there were additional concerns that arose that in hindsight would be common for this type of experiment.

In this section we discuss all these difficulties and explain the choices we made. We also provide an overview of our experiment, including the programs selected for the experiment, the tools used for analysis, the properties included in the experiment, and the measurements we used to evaluate performance.

2.1 Tools in the Experiment

We must actually compare analysis tools rather than analysis techniques. On the one hand, this is reasonable since developers would actually use the tools and not the techniques. On the other

hand, most of the tools are research prototypes that have not been highly optimized and thus the results may not actually give an honest assessment of the strengths of the underlying techniques. In our experiment, we considered the reachability analysis tools SPIN and SPIN plus Partial Orders (SPIN+ PO), the symbolic model checking tool SMV, the inequality necessary condition analysis tool INCA, and the data flow analysis tool FLAVERS. Although there are a number of other concurrency analysis tools, the selected tools represent most of the main approaches to static analysis of concurrent software.

SPIN

The Simple Promela INterpreter (SPIN) [Hol91] performs reachability analysis, in which the reachable states of the program being analyzed are enumerated and the property of interest is checked on the reachable state space. The program is described in the PROMELA language [Hol91], a language that was developed for the specification of communication protocols. Given a PROMELA description of the program and property, SPIN constructs a C program which, when compiled and executed, performs the actual analysis. SPIN automatically checks for deadlock. Other properties to be checked must be specified using *never claims* or *assertions*. In a never claim, the property is represented as a Finite State Automaton (FSA) that should never reach an accept state. Assertions are expressions that evaluate to true or false and are inserted at user-selected points in a PROMELA program. If at any time during the state space generation a potential deadlock state is found, the FSA for a never claim reaches an accept state, or an assertion evaluates to false, the tool reports the error and terminates.

SPIN + Partial Orders

In the worst case, the size of the reachable state space can grow exponentially in the number of tasks in the program. The partial orders approach of Godefroid and Wolper attempts to reduce the size of the reachable state space through the use of *sleep sets* [GW91].

This method has been implemented as an addition to SPIN, and thus we refer to the resulting tool as SPIN+PO. Like SPIN, the SPIN+PO tool takes input in the form of PROMELA, converts that input into a C program, and checks for deadlock automatically. The current version of SPIN+PO does not support the use of never claims for the specification of the property of interest, so non-deadlock properties are specified as assertions embedded in the PROMELA input. SPIN+PO checks those assertions, just as SPIN does during state space generation, and reports a violation and terminates if an assertion evaluates to false.

We note that another partial order addition [Pel94] to SPIN has been implemented, but since it does not currently support the use of rendezvous, we did not consider it for this experiment.

SMV

The Symbolic Model Verifier (SMV) [McM93] performs symbolic model checking [BCM+90], in which the program state space is represented symbolically rather than explicitly. Although SMV was originally designed as a hardware verification tool, it can also be used for analysis of concurrent software. The program is described in the form of a transition relation for the program states. The property of interest is specified in the temporal logic Computation Tree Logic (CTL). If the property is ever false, SMV reports the violation and terminates.

INCA

The INCA tool implements the Inequality Necessary Condition Analysis technique [CA95], in which necessary conditions for an execution of the system to violate a property are formulated as a system of inequalities. The program is specified in an Ada-like language or in the S-Expression Design Language (SEDL). Properties are given as ω-star-less expressions, which specify sequences of event symbols. If there is no integer solution to the system of inequalities there can be no execution that violates the property, and therefore the property must hold for all executions.

FLAVERS

The FLow Analysis VERifier for Software (FLAVERS) tool [DC94] performs data flow analysis to check properties of concurrent programs. The tool accepts a set of Control Flow Graphs (CFGs), annotated with the events of interest, as the specification of the program to be analyzed. The property of interest is specified as a Quantified Regular Expression (QRE), which gives a regular expression for a set of event symbols and specifies whether the expression should hold on all (or no) paths. Checking for deadlock using FLAVERS is not currently supported. Many other properties, however, can be specified as QREs. The tool reports cases for which the data flow analysis shows that there may be an execution on which the property is violated.

2.2 Programs in the Experiment

To get a representative sample of concurrent programs, we would need to randomly select the programs for the experiment from the population of all concurrent programs. This is not feasible for several reasons. We need to restrict the experiment to concurrent programs written in languages with well-defined concurrency constructs. Because Ada is one of the few commonly used languages supporting concurrency, we restricted our experiment to concurrent Ada programs. The population of concurrent Ada programs available for such an experiment is quite limited. Thus, for this study we selected the toy programs that have been discussed in the concurrency analysis literature, realizing full well that no general conclusions could be drawn from this sample but that interesting insights might be obtained.

For this study, we selected 11 scalable programs from the concurrency analysis literature for our experiment. Some of the programs had already been coded in Ada by members of the Arcadia project [Kad92] to demonstrate testing and analysis tools, while for others we acquired the INCA inputs used by Corbett [Cor96] and converted them to Ada programs. Since the size of the programs included in the experiment can be increased by including more tasks into the system, it is also interesting to consider how the performance of the tools changes as the problem size is increased. Toward this end, we collected experimental data for a range of program sizes.

There are a number of ways we could select the sizes of the programs. One way is to find a range of sizes that most of the tools can analyze, which allows comparison between the analysis tools on the same input domain of programs, properties, and sizes. For example, we can run all the tools on an arithmetic progression of sizes, choosing our maximum size as the size on which at least one of the tools fails to complete the analysis. This was the approach taken by Corbett [Cor96] and in our experiment. As noted by Corbett, however, by choosing to consider only those sizes that can be analyzed by most of the tools we may restrict some of the tools to only a small portion of their domains of applicability and may introduce significant bias against those tools if they incur large overhead on small programs but can be used on very large programs. A second approach would therefore be to select different sizes of the programs for each tool based

on the range of sizes that can be analyzed by that tool. This approach gives a clearer picture of each tool's performance, especially in terms of failures, but also precludes direct comparison of the analysis times used by each tool. Yet a third approach would be to select the range of program sizes based on the most effective tool for that program and to measure the failures of the other tools. We note below that the decision about which range of sizes to include in the experiment has implications for a number of other issues.

The programs and sizes selected for the experiment were:

- Cyclic program - provides a loosely synchronized ring of processes, where the processes start in order as the ring is traversed, but each process can complete its task at any time [Mil80]. Sizes: 4, 8, 12, 16, 20, and 24 tasks.
- Divide-and-conquer program - provides a set of solvers that can cooperatively solve a problem [ACD+94]. Sizes: 11, 21, 31, 41, 51, and 61 tasks.
- Standard Dining Philosophers program - includes a ring of philosophers with a single fork between a philosopher and its neighbor to the left. Sizes: 4, 8, 12, 16, 20, and 24 tasks.
- Dining Philosophers with Dictionary program - the philosophers also pass a dictionary around the ring, and the philosopher currently holding the dictionary cannot eat. Sizes: 4, 6, 8, 10, 12, and 14 tasks.
- Dining Philosophers with Fork Manager program- uses a single fork manager to keep track of the status of all the forks in the system. Sizes: 3, 4, 5, 6, 7, and 8 tasks.
- Dining Philosophers with Host program - includes a host that only lets a certain number of philosophers sit in the ring. Sizes: 5, 7, 9, 11, 13, and 15 tasks.
- Gas Station program - provides a simulation of a self-service gas station [HL85]. Sizes: 4, 5, 6, 7, 8, and 9 tasks.
- Hartstone program - based on the hartstone benchmark program, which iteratively starts and stops a series of tasks. Sizes: 11, 21, 31, 41, 51, and 61 tasks.
- Memory management program - based on a set of conservative release and allocate memory management algorithms [For88]. Sizes: 5, 6, 7, 8, 9, and 10 tasks.
- Ring program - based on a simulation of token ring access to a resource [Cor94]. Sizes: 4, 8, 12, 16, 20, and 24 tasks.
- Readers/writers program - includes a set of readers and a set of writers that may be simultaneously accessing the same document, with the restriction that when a writer is accessing the document no readers or other writers can be accessing the document at that time. Sizes: 5, 9, 13, 17, 21, and 25 tasks.

To illustrate our description of the tradeoffs involved in designing an experiment to compare the various concurrency analysis tools, we consider the readers/writers program. This program uses a task for each reader, a task for each writer, and a single task to control access to the document. An example program showing one reader and one writer can be found in Figure 1. To increase the size of the example program, we add additional readers and writers with the same structure as `reader_1` and `writer_1`.

The performance of the analysis tools can be very sensitive to minor changes in the Ada program. For example, the control task in the readers/writers program has two unguarded select alternatives, at the `stop_read` and `stop_write` entries, since the structure of the reader and writer tasks ensures that no calls will be made to these entries unless a reader is currently reading

```
task body reader_1 is
begin
  loop
    control.start_read;
    control.stop_read;
  end loop;
end reader_1;


task body writer_1 is
begin
  loop
    control.start_write;
    control.stop_write;
  end loop;
end writer_1;
```

```
task body control is
  Readers : Natural range 1 .. 1 := 0;
  Writer  : Boolean := false;
begin
  loop
    select
      when (not Writer) =>
        accept start_read;
        Readers := Readers + 1;
      or
        accept stop_read;
        Readers := Readers - 1;
      or when (not Writer) and
              (Readers = 0) =>
        accept start_write;
        Writer := true;
      or
        accept stop_write;
        Writer := false;
    end select;
  end loop;
end control;
```

Figure 1. Ada Program for 1 Reader/1 Writer

or a writer is currently writing, respectively. INCA yields spurious results when checking for freedom from deadlock because of the unguarded select alternatives. Adding guards to these alternatives does not change the semantics of the program, but when we include these guards and model both the `Writer` and `Readers` variables, as discussed below, we are able to eliminate the spurious results from INCA. It might be argued on stylistic grounds that adding the guards makes the control task more robust and easier to understand or, to the contrary, more complex and less clear. The concern is that small differences in programming style can lead to significant variations in the performance of the analysis tools. Moreover, how programming style affects each tool is not well-understood, so we can not avoid biasing our results unintentionally through our choice of a particular style.

2.3 Translating the Ada Programs

To try to ensure the analysis tools were evaluating the same program, we started with Ada programs and converted those programs into the program representations for each tool. Because these conversions are difficult and potentially error-prone to perform manually, we used a largely automated conversion process. This process, depicted in Figure 2, took advantage of the tools we had on hand, and no attempt was made to optimize it. With this process, each Ada program was first converted to a set of CFGs using an automated conversion tool. The FLAVERS tool uses CFGs as input directly, so no further conversion was required for FLAVERS. For the remaining tools, we converted each CFG to SEDL using another conversion tool. The INCA tool uses the SEDL as the program representation, so no further conversion was required for INCA. The INCA tool was then used to generate a set of FSAs that were then converted to the program representations for SPIN, SPIN+PO, and SMV using a slightly modified version of Corbett's conversion tool [Cor96].

The CFGs that are automatically generated from the Ada program are a general abstraction of program control flow and were not explicitly developed to support one or more of the analysis

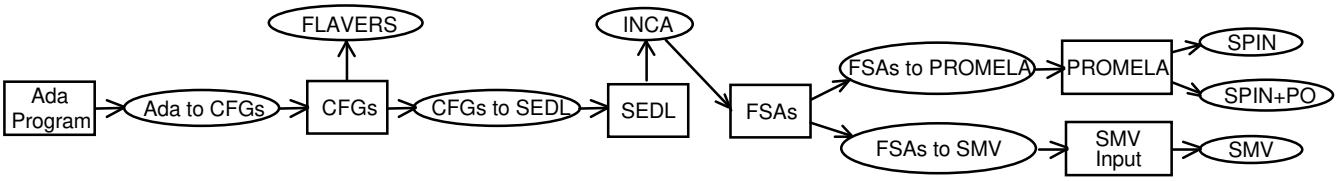

Figure 2. Translation Process

tools evaluated. Similarly, the FSAs that are created to represent the program are a general abstraction and were not tuned to one or more of the analysis tools. While we know that the CFGs and FSAs are valid representation of the program, using these representations could introduce bias in the experiment in some unknown manner.

It is not even possible to have each of the tools analyze exactly the same program. For example, PROMELA semantics are not exactly the same as Ada semantics. Specifying a PROMELA program to behave exactly as the corresponding Ada program would behave is not quite possible. We carefully considered the translation process to try to ensure the tools are analyzing equivalent programs, but differing tool semantics makes this difficult. While we believe our approach to this problem is reasonable, there may be other approaches that are more successful at providing equivalent programs to each of the tools.

There is also a concern that our translation process may introduce bias against some of the tools. For example, our translation process could introduce bias against SPIN and SPIN+PO since specifying each process as a distinct FSA is not standard PROMELA "programming style". Specifying each process as an FSA would preclude the use of techniques that take advantage of multiple instantiations of process types to provide more efficient state space generation. Our approach, however, greatly facilitates the process of translating from an Ada program to a PROMELA program.

Similarly, the standard SMV input specification does not allow two processes to change state simultaneously, as occurs in a synchronous rendezvous communication. SMV supports an alternative input specification style that is in terms of the state transition relation for the system, and we have used this style. This input specification style may bias our results against SMV, however, since techniques that organize the OBDDs to efficiently represent the multiple, duplicate processes can not be used. In addition, by using the transition relation for the system, we are forcing SMV to consider the entire state space of that system, rather than letting it represent the state space symbolically as it was designed to do.

Thus, although we made extensive efforts to accurately translate each program into equivalent representations for each tool, we cannot be sure that these representations are indeed precisely equivalent or that we have not inadvertently introduced bias against one or more of the tools.

2.4 Properties in the Experiment

Our access to requirement specifications for these programs is very limited. Although we derive properties that we believe a developer would be interested in checking, we can not validate this claim. In our experiment, we checked for potential deadlock in all the programs in the experiment. We also selected one or two additional program-specific properties to check key aspects of the functional behavior of each program. Many of these additional properties check some form of mutual exclusion, although other kinds of properties were checked as well.

For example, we selected three properties to check for the readers/writers program. The first of these is deadlock, which occurs if the program reaches a non-terminal state in which none of the tasks can continue executing. The second property checks whether a reader can read an empty document. The third property checks whether two writers can ever be writing at the same time. Another property one would expect to check for this program is whether a reader and a writer can be accessing the document at the same time. We do not describe this property further because it is quite similar to the third property above.

To try to ensure an unbiased tool comparison, we tried to guarantee that the tools were evaluating the same properties. Automatic translation is even more difficult for property specifications than for program representations, however, because the semantics of the specification techniques are so different. For example, FLAVERS QREs are in terms of events while SPIN never claims and SMV SPECs are in terms of process states, and SPIN assertions must be embedded as program statements. Without an automated property translation process, our only assurance that we were specifying equivalent properties was a careful, manual reasoning process applied to each of the properties, for each of the tools. To illustrate the differences in property specifications, in Figure 3 we provide a SPIN never claim, SMV specification, INCA query, and FLAVERS QRE for checking that no reader reads an empty document. Since all reader tasks are identical, this property can be checked based on the results for any selected reader. Because we selected reader_1, checking this property involves recognizing the event that reader_1 has started reading and recognizing all events where some writer starts writing.

We encountered several occasions on which we found it necessary to add additional flags to the PROMELA input in order to recognize when particular events have occurred. While it is sometimes possible to infer the event occurrences from the sequence of states traversed by the processes, thereby avoiding adding these flags, this can be difficult for non-trivial programs. We therefore found it effective to use flags to keep track of the occurrence of events of interest. For instance, when a writer writes, we set a *wrote* flag to true as the writer moves from one state to the next; we made use of this flag in the property specification shown in Figure 3. We also found it necessary to add additional flags when checking an assertion that involves the states of more than one process.

Similarly, we found it intuitive to check event sequence properties with SMV by including flags that keep track of the occurrence of events of interest. It is also possible in SMV to avoid including these additional flags by specifying the property as an alternative CTL formula. Because adding the flags could increase the size of the state space, thereby adversely affecting the analysis times for SMV, we checked the event sequence properties using both specification techniques. We found that, in general, the alternative CTL specifications seemed more complicated (i.e., contained more terms and temporal logic operators) than those using additional flags, but this was not always the case. We also note that there were many alternative CTL specification from which we could have picked, so even our selection of the CTL specification used could affect the analysis times for SMV.

INCA queries are specified in terms of intervals, where events of interest are typically used to specify the start or end of the intervals. For some of the properties in the experiment, we found it intuitive to specify the INCA query using two intervals, which can cause a significant growth in the size of the inequality system. In these cases, we also used an alternative property specification involving a single interval with additional constraints.

SPIN Never Claim

```
never {
  do
    :: (wrote == true) -> break
    :: reader__1[reader_1_pid]@state_2 -> goto accept
    :: else -> skip
  od;
  do
    :: skip

  od;
accept:
  do
    :: skip

  od;
}
```

Comments

```
-- make sure the following never occurs:
-- start of first loop
   -- if any writer writes, exit the loop
   -- if reader_1 reads, go to accept state
   -- if neither of above, loop back
-- end of first loop
-- start of second loop
   -- infinite loop - if we get here,
   --    the property is not possible
-- end of second loop
-- accept state of never claim
-- start of accept state loop
   -- infinite loop - reader_1 reading before
   --    some writer writes
-- end of accept state loop
-- end of never claim
```

SMV Specification

```
AG ( reader_1_read -> any_writer_wrote )
```

```
-- on all paths it must always be true that reader_1
--    reading implies that some writer already wrote
```

INCA Query

```
(defquery "no_r1w"
   "nofair"
   (omega-star-less (sequence
      (interval :initial t
         :ends-with '((rend "reader_1;control.start_read"))
         :forbid '((rend "writer_1;control.start_write"))))))
```

```
-- define the query no_r1w
-- with no fairness constraints required
-- as a sequence of intervals such that
--   there is an interval from the start of the program
--   to the point at which reader_1 reads
--   in which writer_1 is NOT allowed to write
```

FLAVERS QRE

```
{reader_1_read, any_writer_wrote}

none
[-reader_1_read, any_writer_wrote]*;

reader_1_read;
[reader_1_read, any_writer_wrote]*
```

```
-- events of interest are reader_1 reading and
--    any writer writing
-- check that there is no path on which
--   events of no interest occur
--      (potentially multiple times)
--   followed by reader_1 reading
--   followed by any event
```

Figure 3. Example Property Specifications to Check that No Reader Reads an Empty Document

With the small, academic programs in the experiment, we knew which properties should be violated for each program, property, and set of modeled variables (see Section 2.5). If we specified a property that we knew should not be violated and the analysis reported that the property might be violated, we iteratively modified our property specification until we achieved the "correct" analysis result, or could no longer see reasonable ways to modify the property specification. In some cases, specifying the property was very difficult and reaching a correct property specification required many iterations. We used this iterative process to try to factor out our inexperience using the tools, since the original spurious results were often caused by our incorrect property specifications rather than by weaknesses in the tools. We believe that the spurious results measured in the experiment, using the final versions of the property specifications, therefore accurately represent the strengths and weaknesses of the tools rather than

our skill (or lack of it) specifying the properties. It can be argued, of course, that our original property specifications should be used, since they may better reflect how a " typical" user would specify the properties. Additionally, an analyst analyzing a real concurrent program probably does not know the "correct" analysis result, and so would not necessarily know when to iteratively modify the property specification.

Ensuring that each tool checks the same property is difficult. The property specifications that we chose for each tool imply the high-level property that we wanted to check but there are many other ways of specifying these properties, even for a single tool. The particular property specifications that we used could therefore introduce bias against one or more of the tools. Moreover, we have no evidence that the set of properties we selected are representative of the kinds of properties system developers would be most interested in checking. For instance, our choice of properties could reflect our past analysis experience, which was largely with event-based methods.

2.5 Improving Analysis Accuracy

For the tools we examined, users can exert some control over analysis accuracy by deciding which program variables are *modeled*. A representation of the program that allows all variables in the program to have all possible values at each point that they are used is certainly conservative, but is likely to include a large number of infeasible paths. If information about the values of the modeled variables (at certain points in program executions) can be determined statically and incorporated into the analysis, the accuracy of the analysis will be improved.

As part of our experiment, we improved the accuracy of the analysis results by selectively modeling variables. For example, the `Writer` variable in the control task of the readers/writers program ensures that only a single writer can be writing at a time. The `Readers` variable ensures that there is never a situation in which the reader is reading at the same time the writer is writing. By modeling the values of one or both of these variables, we can generate program representations that more accurately describe the control task behavior than one in which neither variable is modeled. The choice of which variables to model, however, may well introduce bias against some of the tools. For example, SPIN, SPIN+PO, SMV, and INCA all run faster when both variables are modeled in the readers/writers program.

The problem is that, for a program that contains a large number of variables, trying to model all the variables might make building the program representations or performing the analysis on those representations intractable. One approach would be to add accuracy to the program representations incrementally. With this approach, we would start without modeling variables, and would incrementally model additional variables until the analysis results meets the accuracy requirements; this is the approach we took in our experiment. Another approach [Cor94, Cor96] would model all variables initially, or at least all variables that directly impact the events in the property. If we discovered that this variable modeling led to intractable analyses, we would incrementally remove variable modeling until the analysis was tractable. For real programs, we believe an analyst would probably select a set of variables that seemed most relevant, rather than starting at either extreme.

2.6 Measurements in the Experiment

We are interested in using the results of our experiment to characterize the performance of each tool. One way to characterize tool performance is in terms of the consumption of resources,

especially analysis time. Another useful measurement is the failure rate for each tool. The analysis can fail because the tool takes longer than a specified period of time to complete the analysis, exhausts available memory, terminates with some internal error, or a program generated by the tool cannot be compiled (i.e., the C programs generated by SPIN or SPIN+PO). Although any time limit is somewhat arbitrary, clearly some time limit is necessary. Similarly, the memory available is limited by the configuration of the machine used for the experiment. The performance of the tools is also affected by the accuracy of their analysis results. Given the relative simplicity of the programs included in our experiment, we can determine the correct answer for each of the analyses and can, therefore, recognize spurious results reported by an analysis tool. Although this is an important characteristic of performance, it can be very difficult to measure for larger, more complex programs than those used in our experiment. As expected, there does not appear to be a single measure that will fully characterize any of these aspects of the tools. Instead, it is necessary to use a variety of these measures, but the best way to select the measures is not clear.

For example, we would like to measure the analysis time required for each tool to analyze a particular program and property. The first question is "What exactly constitutes the analysis time used by the tool?". We could consider the analysis time to be the time each tool takes to generate the analysis results starting from its native input. This time does not include the cost of translating the Ada programs into each tool's input language as part of the analysis time for that tool, on the grounds that this translation is an artifact of our methodology. This is the definition of analysis time that we use in our comparison. On the other hand, the translation is necessary to use the tools on Ada programs, and an analyst using one of these tools as part of the development of a concurrent Ada program would certainly want to know how long the full analysis would take, starting from the Ada code. Analysis time could therefore also be defined to include timing information for all the translation steps in the analysis process and for the compilation of the C programs generated by SPIN and SPIN+PO.

One way to compare analysis times is to compare the mean analysis times for each tool, where the tools with the lower mean times would fare best in the comparison. Unfortunately, outliers can have a significant effect on the mean (although standard deviation and scatter plots can help identify this problem). For example, a tool with consistently small analysis times, except for a few very large analysis times, could easily have a larger mean analysis time than a tool that has consistently larger analysis times but no outliers. Also, if a tool fails on a large number of cases but has small analysis times for those cases on which it does not fail, this tool would have a lower mean analysis time than a tool that fails less often but is slower on the cases for which it does not fail. Because means are commonly used for comparison, we compared mean analysis times for the analysis tools.

Another way to do the comparison is to count the number of cases for which each tool has the fastest analysis time; tools with the largest numbers of "fastest cases" would fare best in this comparison. This measure also has problems, however. Specifically, a tool that consistently had the second or third fastest analysis times, but seldom had the fastest, would do worse in this comparison than a tool that had the fastest analysis times more often than the first tool, but generally had the slowest analysis times. We would like a measure that not only captures how well a tool compares to the others for each case, but also includes some (indirect) measure of consistency. We therefore did not use counts of fastest cases to compare the analysis tools.

Another way to compare analysis times is the average ranking for each tool. For each case, we rank the tools (1 = fastest, 2 = second fastest, etc.) based on analysis time. For each tool, we then

average these rankings across all cases and use this average for comparison, where tools with the smallest average ranking would fare best in this comparison. This average can still be affected by outliers, but because the worst ranking a tool can have on a given case is given by the number of tools in the experiment, the effect of outliers is not of much concern. Because it is an average, this measure also (indirectly) includes consistency. Using the average ranking should help identify tools that perform very well on some programs but very poorly on others. This measure seems to be a reasonable way to compare analysis times, so we also used average ranking to compare the analysis tools.

Thus, the choice of what to compare for analysis times is a difficult one. All of the above measures have advantages and disadvantages, but the most useful way to make use of these measures is not obvious.

It would also be interesting to measure the rates at which analysis time and consumption of other resources, such as memory, grow as the size of the programs being analyzed increases. Corbett proposed a measure of such growth for individual programs for a single property [Cor96]. Although Corbett provides some justification for his calculation of growth rate, it is a relatively crude measure. Moreover, it is not clear how we could extend this calculation to combine data from multiple programs and properties.

Another issue is how to compare tool failures. In our experiment, any analysis that took longer than 5 hours of CPU time was classified as a failed analysis. Analysis cases that exhausted available memory, terminated with some internal error, or could not be compiled (for SPIN and SPIN+PO) were also classified as failures. Because we selected the range of program sizes in the experiment so that most of the tools could complete the analyses, our sample does not provide the best basis for comparing failures. All the tools failed on at least one analysis case, however, so we chose to proceed with our failure comparison.

One way to compare failures is to compare the counts of failure cases for each tool; the tools with the lowest number of failed cases would fare best in the comparison. Another way to compare failures is to use percentages of failures instead of failure counts. For each tool, we calculate the percentage of analysis cases on which that tool failed. We can then compare these percentages across the tools, where the tools with the lowest failure percentages would fare best in the comparison. For both these comparisons to be meaningful, all the tools need to be run on the same analysis cases, as they are in our experiment. Since the two measures are equivalent, we used percentages of failures for our comparison.

A third issue is how to compare spurious results. One way is to use counts of the spurious results for comparisons. Because a spurious result would only be counted on an analysis case that did not fail, however, and because the tools are unlikely to fail on the exact same number of cases, comparing spurious result counts is problematic. We therefore chose not to use this measure for comparison.

Another way to compare spurious results is to use percentages of spurious results for comparison. For each tool, we calculate the percentage of analysis cases on which the tool yielded spurious results, and we used these percentages for comparison.

It is difficult to immediately discern and compare the percentage of successful analysis cases for the tools by using only failure or spurious result percentages. We define a successful analysis case as one that runs to completion (does not fail) and yields the correct answer (does not give a spurious result). The percentage of successful analysis cases may provide a better point of comparison than failure or spurious result percentages used in isolation.

One important quantity that we did not measure in our experiment is how much user effort is required to apply each of the analysis tools. This effort includes adding flags to the PROMELA and SMV inputs for some of the properties, specifying the properties using each tool's specification formalism, and determining the cause of spurious results using each tool's output. Especially given the iterative property specification process described above, user effort can be significant. Unfortunately, measuring user effort in an unbiased manner would require an experiment using multiple human subjects, with the extensive additional experimental design concerns such a human experiment entails. In addition, such an experiment seems premature until we have a better understanding of the strengths and weaknesses of each tool.

As expected, there does not seem to be single measure that we can use to capture analysis time, failures, or spurious results for comparison. Instead, it is necessary to use a variety of these measures. In our experiment we used mean analysis times, ranked analysis times, failure percentages, spurious result percentages, and successful analysis case percentages. It is not clear, however, that this is the best selection from among the measures discussed above and other possible measures. Also, because we chose to selectively model (and not model) important variables in the programs to examine analysis accuracy, it is somewhat misleading to simply compare the measures over all the analysis cases. Instead, we compared the measures for the cases in which we modeled the minimal set of variables required to prove each property.

2.7 Checking for Bias

We want to ensure that our experimental design avoids bias against one or more of the tools as much as possible. There are a number of potential sources of bias introduced by the choices we make about which programs to analyze, how to translate the programs, and how to formulate the properties in the experiment. We can check statistically whether some of these potential biases affect our experimental results.

For example, the sizes of the Ordered Binary Decision Diagrams (OBDDs) used by SMV are sensitive to the order of the variables in the SMV input. To account for this, we checked the tool's performance using the variable ordering that results from our automatic translation and also using the REORDER option, which applies a heuristic reordering algorithm before generating the OBDDs for the system. As previously noted, SMV tends to be more efficient when processes are used rather than an explicit specification of the global transition relation, since using the transition relation forces SMV to consider the entire state space of the program. Modeling the semantics of the Ada rendezvous using the semantics of the SMV processes is not possible, however. Thus, we may have introduced bias against SMV by explicitly specifying the global transition relation, but we are unable to check for this bias.

There is also a potential bias introduced against SMV when we use additional flags and embed operations on those flags in the system transitions in order to check certain properties. Since this could degrade SMV's performance by increasing the size of the state space, we also specified event sequence properties using CTL specifications that do not require additional flags.

We specified properties for SPIN using both never claims and assertions to ensure that our choice of property specification technique did not bias our results against SPIN.

In our implementation of the readers/writers problem, none of the accept statements have bodies. This does not affect the tools using inputs based on FSAs because the accept bodies are collapsed into single FSA states. Similarly, it does not affect FLAVERS, since this tool optimizes the empty accept bodies away. It is not clear, however, whether this affects the performance of

INCA. Therefore, for INCA we ran the analysis cases on a version of the program that had no accept bodies and on a version containing accept bodies. We also note that most examples of INCA input that we have seen represent sets of identical tasks as arrays of task types, while the Ada program we used as a model contains each reader and writer task specified uniquely. Since it is unclear how this affects INCA performance, we ran the analysis cases on INCA with a conversion from the Ada program and with a conversion from a version with arrays of reader and writer tasks. Finally, for some properties, we found it intuitive to specify the INCA query using two intervals, which can cause a significant growth in the size of the inequality system. In these cases, we also specified the queries using single intervals with additional constraints and ran the analysis cases using both types of queries.

Unless it can be formally established that these variations have no effect on the experimental results, it is important that any experiment using these tools check for potential bias. The variations, or *configurations*, that we considered in our experiment are shown in Table 1. Given the various static analysis tools in the experiment, the configurations in Table 1, the 11 programs, the 6 sizes of each program, and the properties we checked in the experiment, we needed to execute 1,812 distinct (tool, configuration, program, size, property) tuples. To account for the possibility of run time variations, we needed to execute each of these tuples a number of times; since we did not observe significant variation in these run-times, we opted to execute each tuple 5 times. We therefore conducted a total of 9,060 runs, some of which took hours to complete. Thus, checking for bias greatly increased the amount of time required to conduct the experiment but seemed essential because of the sensitivity of the tools and the numerous opportunities for bias.

To check for potential biases we performed a standard form of hypothesis testing. In hypothesis testing, a *null hypothesis* (H_0) and an *alternative hypothesis* (H_1) are formed, a set of data is collected, and the probability of collecting that set of data given that the null hypothesis is true is calculated. If this probability is very small (less than 0.05 is typically considered significant), we can reject the null hypothesis (and accept the alternative hypothesis) with a small probability of doing so incorrectly. Incorrectly rejecting the null hypothesis is called a Type I error, while not rejecting the null hypothesis when we actually should have is called a Type II

Tool	Property Style	REORDER	Accept Bodies	All Arrays
SPIN	Never Claims			
	Assertions			
SMV	Additional Variables	No		
	Additional Variables	Yes		
	Alternative CTL Spec	No		
	Alternative CTL Spec	Yes		
INCA	Multiple Intervals		No	No
	Multiple Intervals		No	Yes
	Multiple Intervals		Yes	No
	Multiple Intervals		Yes	Yes
	Additional Constraints		No	No
	Additional Constraints		No	Yes
	Additional Constraints		Yes	No
	Additional Constraints		Yes	Yes

Table 1. Tool Configurations

error. Nonetheless, if we do not reject the null hypothesis, we have not proved it -- we have simply been unable to reject it given the data at hand.

As an example, consider the hypotheses for checking to see whether using assertions rather than never claims introduces bias against SPIN. The null hypothesis for our example was that analysis times using assertions were equal to analysis times using never claims. For our alternative hypothesis, we checked whether the analysis times were different. Hypothesis formulation for checking the other biases is similar.

To perform the actual hypothesis testing we used the sign test, a nonparametric test for comparing the means of two distributions [Spr93]. While the paired-sample t-test is a more common way to compare means when cases need to be paired in the calculation, the paired-sample t-test assumes that the differences between the cases are normally distributed. We used the Kolmogorov-Smirnov test to examine the probability that the distributions of differences were normal and found that we could reject the null hypothesis that they were normal in all cases. We therefore chose to use the sign test, which does not make assumptions about the distributions of analysis times or the distribution of the differences between them.

The results of the hypothesis testing in our experiment indicated that, with statistical significance, SPIN using assertions yielded smaller analysis times than SPIN using never claims, SMV using reorder yielded smaller analysis times than SMV without reorder, SMV using additional flags yielded smaller analysis times than SMV using a CTL specification without those flags, and INCA using unique tasks yielded smaller analysis times than INCA using arrays of tasks. However, these results do not prove that these potential areas of bias will affect an experiment using other programs, properties, or program sizes. For example, our statistical tests indicated that we did not introduce bias into our experiment using two intervals, rather than one with additional constraints, in our INCA queries, but we know that this choice would introduce bias for larger program sizes. Therefore, any experiment using these tools should incorporate a means of checking for any identified areas of potential bias.

3 COMPARISON RESULTS

In this section, we describe the result of our empirical comparison, with a more complete discussion provided in [Cha96]. We found that there was no single best tool for the programs and properties in our experiment, but our comparison results require some discussion.

Based on the results of our hypothesis testing, we simplify the discussion below by only presenting the results for SPIN using assertions, SPIN+PO, SMV using the REORDER option and additional flags, INCA using no accept bodies, unique tasks, and properties specified using multiple intervals, and FLAVERS. We note that data for FLAVERS is not included for three of the programs. The hartstone program uses arrays of task types (CFGs for tasks generated from an array of task types are not currently supported in FLAVERS), and our analysis of the memory management and ring programs indicated bugs that we have reported to the FLAVERS developers.

The data we used for our analysis time comparisons is provided in Table 2. Recall that we only compare analysis times (and failures and spurious results) for those cases in which we modeled the minimal set of variables required to prove each property. When we compared mean analysis times checking for deadlock, we found that SPIN+PO yielded the best mean times (16.24 sec), followed after a large gap by INCA (45.16 sec), which was followed after another large gap by SPIN (63.42 sec). For checking non-deadlock properties, INCA had the best mean analysis

	Deadlock		Other Properties	
	Mean	Average Rank	Mean	Average Rank
SPIN	63.42	2.15	79.02	2.05
SPIN+PO	16.24	1.97	54.50	2.57
SMV	113.65	2.14	53.93	1.86
INCA	45.16	3.23	12.99	2.70
FLAVERS	-	-	198.61	4.18

Table 2. Analysis Time Data

	Deadlock			Other Properties		
	Failures	Spurious Results	Successful Cases	Failures	Spurious Results	Successful Cases
SPIN	10.6	0.0	89.4	19.2	0.0	80.8
SPIN+PO	7.6	0.0	92.4	16.2	0.0	83.8
SMV	4.6	0.0	95.4	7.1	0.0	92.9
INCA	1.5	1.5	97.0	0.0	0.0	100.0
FLAVERS	-	-	-	13.2	0.0	86.8

Table 3. Failure, Spurious Result, and Success Percentages

times (12.99 sec) followed after a large gap by SMV (53.93 sec) and SPIN+PO (54.50 sec). When we used the average ranking described above, SPIN+PO (1.97), SMV (2.14), and SPIN (2.15) had the best average rankings checking for deadlock, while SMV (1.86), SPIN (2.05) and SPIN+PO (2.57) had the best average rankings checking non-deadlock properties. Note that comparing mean analysis times yielded significantly different results than comparing average rankings, particularly for non-deadlock properties. We point out that the average rankings for deadlock and non-deadlock properties do not add up to 10 and 15, respectively, as might be expected. This occurs because we do not give a tool a ranking (nor an analysis time) for a case on which it failed, although we recognize that this makes interpretation of the comparison results somewhat more difficult.

The data we used for our comparisons of failures, spurious results, and successful analysis cases are provided in Table 3. When we compared failure percentages checking for deadlock, we found that INCA (1.5%) yielded the smallest failure percentage, followed by SMV (4.6%) and SPIN+PO (7.6%). For checking non-deadlock properties, INCA (0.0%) again yielded the smallest failure percentage, followed after a large gap by SMV (7.1%) and FLAVERS (13.2%).

When we compared spurious result percentages we found that, for the cases in which they did not fail, SPIN, SPIN+PO, and SMV all had no spurious results checking for deadlock, followed closely by INCA (1.5%). For non-deadlock properties, all the tools had no spurious results.

Finally, we compared the successful analysis case percentages as described above. We found that INCA (97.0%), SMV (95.4%), and SPIN+PO (92.4%) had similar successful analysis case percentages checking for deadlock. For non-deadlock properties, the comparison reduces to a comparison of failure percentages since none of the tools yielded spurious results on non-deadlock properties.

We again caution the reader that drawing general conclusions from the results of this relatively small experiment is unwise since we know that the sample of programs and properties is not representative of the population of concurrent Ada programs and concurrency properties. We do

note that the experiment has provided significant insight into the strengths and weaknesses of each of the tools and has also given us valuable experience designing this type of experiment.

4 PREDICTIVE MODELING RESULTS

We hypothesize that there are certain characteristics of programs that affect the feasibility of analysis and the accuracy of the analysis results for those programs. In addition, we believe that certain characteristics of a property being checked may also affect feasibility and analysis accuracy for that property on a given program. Our primary goal was to use statistical regression techniques to determine how well each of the program and property characteristics predicts the values of the response variables (i.e., the measurements discussed above). The resulting regression equations could then be used as predictive models to predict each tool's analysis performance given a specific program and property.

4.1 Metrics

For our purposes, a *metric* is defined as a measurement of some characteristic of the program or property of interest. We divided our metrics into three categories: program metrics, internal representation metrics, and property metrics. The program metrics are used to capture characteristics of the Ada programs being analyzed. The internal representation metrics are used to capture characteristics of the set of FSAs for that program, the set of Task Interaction Graphs (TIGs, [LC89]) for that program, and the state space and transition relation for SMV. The property metrics are used to capture characteristics of the SPIN never claim and assertions, INCA query, and FLAVERS Property Automaton for each property. We treated the program, internal representation, and property metrics as predictor variables in the experiment.

The metrics were selected in a number of ways. Characteristics that affect analysis performance based on the theoretical bounds of the techniques are included, as are other characteristics that we believe may have an effect on analysis performance. Metrics that have been proposed in the concurrency analysis literature are also included.

Program Metrics

The program metrics are used to capture certain characteristics of the Ada program being analyzed. These characteristics include several measures of the size of the program, various measures of nondeterminism in the program and other characteristics of the program structure, and a metric indicating how many variables are modeled in the representations.

The theoretical upper bound on the number of possible program states for a concurrent program is exponential in the number of tasks in that program. We therefore included the number of tasks in the program (T) as one of the program metrics.

We suspect that the number of possible communications in a program affects the number of reachable states for that program. To calculate the number of communications, C_i, for a task T_i, we add the number of accept statements in the task to the number of entry calls in the task. We use two measures of communication size as metrics - the average number of communications for the set of tasks in the program, given by $C = \frac{1}{T}\left(\sum_{i=1}^{T} C_i\right)$ and the maximum number of communications in the set of tasks for the program, given by $MaxC = \max(C_i)$.

One of the characteristics of concurrent Ada programs that makes them particularly difficult to analyze is nondeterminism. None of the metrics above try to account for nondeterminism in the

program being analyzed. Damerla and Shatz [DS92] propose several metrics that we also included in our experiment that are intended to quantify the nondeterminism in Ada programs. A metric called Alpha is used to account for the nondeterminism in entries when several tasks can make entry calls on those entries (entry nondeterminism). Alpha is given by $\sum_{i=1}^{e} (Calls_i - 1)$, where e is the number of entries not contained in selects and $Calls_i$ is the number of calls on entry i. The one is subtracted because an entry with only one caller is deterministic. A metric called Alpha' is similar to Alpha, but also takes into account the *clustering* and *spreading* of entry calls. Entry calls on a given accept are clustered when they occur in a single task; entry calls on a given accept are spread when they occur in multiple tasks. For example, if all the entry calls on a given accept are clustered in the same task, the entry nondeterminism for this accept should be 0. Alpha' for a particular accept a is given by $\text{Alpha}'_a = \Pi_i (x_i) * (2^T - (T+1))$, where x_i is the sum of entry calls in task i on the accept a and T is the number of tasks making calls on the accept. Alpha' is given by $\sum_{a=1}^{e} \text{Alpha}'_a$. The metric Beta is used to account for the nondeterministic selection of rendezvous within select statements (select nondeterminism). Beta is given by $\sum_{i=1}^{s} (Calls_i - 1)$, where s is the number of selects and $Calls_i$ is the number of calls on entries within select i. The one is subtracted because a select with only one call on an entry within the select is deterministic. Similarly to Alpha' , a metric Beta' is defined to account for entry call spreading and clustering. Beta' for a particular select a is given by $\text{Beta}'_a = \Pi_i (x_i) * (2^T - (T+1))$, where x_i is the sum of entry calls in task i on alternatives in the select and T is the number of tasks making calls on alternatives in the select. Beta' is given by $\sum_{a=1}^{s} \text{Beta}'_a$. The metrics Gamma (Alpha + Beta) and Gamma' (Alpha' + Beta') are used to account for total nondeterminism.

Levine and Taylor [LT93] propose a metric similar to Gamma called Cnd to account for nondeterminism. Cnd is given by $\sum_{i=1}^{e+s} (Calls_i - 1) + \sum_{i=1}^{s} (Called_i - 1)$, where $Calls_i$ is the number of entry calls on entry i and $Called_i$ is the number of select alternatives with one or more callers. The difference between Cnd and Gamma is that Cnd includes entry nondeterminism for all entries (as opposed to excluding those in selects) and counts the number of select alternatives with one or more callers when calculating select nondeterminism. To account for clustering and spreading, Cnd' is defined as

$$\sum_{i=1}^{e+s} \left(\prod_{j=1}^{T_i} x_{ij} * (2^{T_i} - (T_i + 1)) \right) + \sum_{i=1}^{s} \left(\prod_{j=1}^{R_i} z_{ij} * (2^{R_i} - (R_i + 1)) \right).$$

T_i is the number of task with calls on entry or select statement i, x_{ij} is the number of entry calls in task j on entry i, R_i is the smaller of T_i and the number of alternatives of select statement i with one or more callers, and z_{ij} is the number of entry calls in task j on entry alternatives in select statement i (if $R_i = T_i$) or the number of alternatives in select statement i with one or more calls in task j.

Levine and Taylor also propose a metric, Cif, for capturing the communication structure or information flow for the tasks comprising the program. Cif is given by

$$\left(\sum_{i=1}^{T} (in - edges_i)(out - edges_i) \right)^{\ln T},$$

where T is the number of tasks in the program, $in\text{-}edges_i$ is the sum of task entries and shared variables read in task i, and $out\text{-}edges_i$ is the sum of entry calls and shared variables written by task i. Cnd, Cnd', and Cif were also included in our experiment.

As discussed earlier, we sometimes chose to model certain variables to try to improve the accuracy of the analysis. When we did so, both the accuracy and the time to complete the analysis were almost always affected. While we capture some of the effects of this modeling indirectly through the metrics described above, we also explicitly included a metric, Vars, that specifies the number of variables that are modeled in the program.

Internal Representation Metrics

The internal representation metrics are used to capture characteristics of the set of FSAs for a given program, the set of TIGs for that program, and the state space and transition relation for the SMV representation of the program. These characteristics include several measures of the sizes of the representations and a measure of the graph theoretic complexity of the program in terms of TIGs.

As noted above, the upper bound for the number of states in a concurrent program is exponential in the number of tasks, T. When the program is represented by a set of FSAs, the upper bound is given by N^T, where N is given by $\frac{1}{T}\left(\sum_{i=1}^{T} n_i\right)$ and n_i is the number of states in task i. We therefore included N as a metric in the experiment. We also included the maximum number of states in an FSA, $MaxN = \max(n_i)$, as a metric, because a large number of states in an FSA for one of the tasks could significantly affect N. Wampler has proposed the metric $N^{T/2}$ as a good predictor of reachability graph size, at least for some programs [Wam85] and we included the WFSA (for Wampler, FSAs) metric in our experiment as well.

Because we believe the communications between the FSAs will affect the analysis, we included two measures of communication size for the FSAs, noting that in general transitions in the FSAs represent accepts or entry calls in the original program. We included the average number of transitions in the set of FSAs for the program, given by $TRANS = \frac{1}{T}\left(\sum_{i=1}^{T} Trans_i\right)$ and the maximum number of transitions in the set of FSAs for the program, given by $MaxTRANS = \max(Trans_i)$.

The above metrics can also be calculated for the set of TIGs for a given program (rather than the set of FSAs). We call the average number of nodes in the set of TIGs TN, the maximum number of nodes MaxTN, the average number of edges TE, the maximum number of edges MaxTE, and the Wampler metric WTIG (for Wampler, TIGs). We calculate these metrics for TIGs as well because a TIG is a conceptually different representation of a task than an FSA. The key difference is that the FSAs include information about choices in the task based on variable values, while TIGs abstract that information away. We note that the elision of variable information tends to yield TIGs that are smaller, in some cases much smaller, than the FSAs for the same tasks.

Levine and Taylor [LT93] propose a metric, called Cgt, intended to capture the graph theoretic complexity of the program. Cgt is given by E - N + T + 1, where E is the number of entry calls and accepts in the program, N is the number of TIG nodes in the program, and T is the number of tasks in the program. Cgt was included as a metric in the experiment.

In addition to the metrics above, we also included two characteristics of the SMV system. We include the total number of task states (SMV St) because this number is related to the total number of sequential regions in the program. We also include the number of transitions in the transition relation (SMV Tr) as a metric in the experiment, because each transition represents a possible communication in the program.

Property Metrics

We believe that characteristics of the property being analyzed might affect the feasibility and accuracy of analysis of that property on a given program. We therefore attempt to capture characteristics of these properties through certain metrics on the property specifications for the tools. The property metrics are used to capture characteristics of the SPIN never claim and assertions, INCA query, and the property automaton FLAVERS generates from the QRE for each property.

Since expressing a property as an FSA seems to be a general and intuitive technique, we included three metrics on FLAVERS Property Automata to capture the size of the property. We included the size of the event alphabets (i.e., number of events of interest) and the number of states in the automaton as metrics. We included the number of transitions in the automaton that do not directly lead to a violation of the property, which gives us another measure of the size of the property.

We can capture the number of events of interest in the property by considering the INCA query as well, so we included the number of distinct events in the INCA query as a metric. We also included the number of intervals in the INCA query, since multiple intervals in the query can significantly increase the size of the system of inequalities.

While FLAVERS QREs and INCA queries tend to be in terms of events, checking a property in SPIN entails specifying the property in terms of states. The SPIN never claim is essentially an FSA for the property, so we included the number of states and transitions in the never claim as metrics in the experiment. We also included the number of assertions and the number of assignments to variables used in the assertions as measures of the amount of information needed to check the property.

4.2 Building Predictive Models

Linear regression models can be used as approximations of the functional relationship between a response variable and a set of predictor variables [MP82]. We used linear regression to build our predictive models of analysis time based on the set of metrics selected for inclusion using the preprocessing discussed above.

While linear regression is a widely used for predicting continuous response variables, it is not appropriate for predicting dichotomous response variables [Agr84]. Because linear regression assumes that the response variable has a continuous range of values, it can not be applied when the response variable can only have two values (true and false, for instance). Logistic regression is the proper technique for these variables, so we used logistic regression to build our predictive models for failure and presence of spurious results.

It has been noted in the literature that high linear correlations between several (or many) of our predictor variables can cause problems [Bla70] in both of the regression techniques that we use. It was therefore necessary for us to preprocess our experimental data, removing predictor variables that are highly correlated to other predictors.

One relatively straightforward way to detect multicollinearity is to consider the pairwise Pearson correlation coefficients for the predictor variables [NWK85]. Pearson's correlation coefficient provides an estimate of the linear relationship between two variables x and y. The coefficient ranges from -1.0 to 1.0, with a coefficient magnitude close to 1.0 indicating a strong relationship and a magnitude close to 0.0 indicating no linear relationship. We note that a low correlation only indicates that the variables are not linearly associated; they could still be related in some non-linear way. If we find a high correlation coefficient between two predictor variables, this provides strong evidence that the variables are collinear, implying that we should elide one of them from the model.

Before omitting certain variables from the model, we would like some assurance that the correlation coefficients represent a systematic linear relationship and did not simply occur by chance. Standard statistical tests are not applicable, since our concern is about distributions of the correlation coefficients rather than distributions of the mean. However, we can use randomization tests, in conjunction with correlation, to test the hypothesis that two samples are linearly dependent [Coh95].

To conduct the randomization test, we randomly paired up values of the first and second variables and calculated the correlation coefficient. This gave us a single point in the distribution of correlation coefficients that are possible for our set of data, given the null hypothesis that the two variables are in fact linearly independent. We then repeated the random pairing and coefficient calculation many times (in our case, 1000) to build a distribution of possible correlation coefficients. We then took the correlation coefficient with the true pairing (i.e., matching variable values for the same analysis cases) and determined where this correlation coefficient falls on the generated distribution. If the coefficient fell below the 5th value in the distribution or above the 995th value (conceptually, $p < 0.05$), we rejected the null hypothesis with high confidence, i.e., we can state that there is a linear dependence between the two variables with only a small probability that we are wrong. We conducted the randomization test on all variable pairs that have a correlation coefficient magnitude greater than 0.75. We note that randomization tests do not provide results that are generalizable to populations, so the two variables could in fact be linearly independent over the set of all possible data. The tests do, however, provide sufficient power given our specific data set.

When we discovered a set of variables that were collinear to each other, we removed all but one of those variables from the regression analysis. The decision about which collinear variables to elide was not critical from the standpoint of the fit of the model we created, since the reason we were omitting the variables was because they provided essentially the same influence as the variables we included in the model. In an effort to make the predictive models more intuitive, however, our tendency was to prefer variables representing the program metrics over those representing the internal representation metrics, and to prefer simpler internal representation metrics over more complicated ones. The sets of collinear metrics and our selections for inclusion in the models are given in Table 4.

To determine how well a predictive model fits the data used, we need some measure of how well the model captures the variance in the data. For linear regression, the standard measure of this is the Multiple Correlation Coefficient Squared, or R^2. R^2 is given by $R^2 = 1 - \frac{SS_E}{S_{yy}}$. The

Set of Collinear Metrics	Selected Metric
{ N, MaxN, TRANS, TE, MaxTE, Cgt }	N
{C, Alpha, Gamma, Cnd, TN }	C
{ Cnd', Beta', Gamma' }	Cnd'
{ Cnd, Beta, Gamma }	Beta
{ TRANS, MaxTRANS, SMV Trans }	MaxTRANS
{ MaxC, MaxTN }	MaxC
{ WFSA, WTIG }	WFSA
{ T }	T
{ Vars }	Vars
{ Alpha' }	Alpha'
{ Cif }	Cif
{ QRE Alphabet, QRE Trans }	QRE Alphabet
{ Never States, Never Trans }	Never States
{ Assertions, Assignments }	Assertions
{ QRE States }	QRE States
{ Query Events }	Query Events
{ Query Intervals }	Query Intervals

Table 4. Collinear Sets of Metrics

residual sum of squares, SS_E, is given by $\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, which squares the difference between the actual and predicted value of the response variable (called the *residual*) at each data point. S_{yy} is a measure of the total variability in the response variable. Thus, R^2 measures how much of the variance in the response variable is captured by the predictive model. R^2 ranges from 0 to 1, with a magnitude near 1 indicating that the model explains most of the variance in the data.

In logistic regression, the deviance can be used to measure the amount of deviation captured by the fitted model. The deviance in logistic regression is analogous to the residual sum of squares in linear regression, since the deviance quantifies how much of the variance in the data is captured by a specific model (with a smaller deviance indicating a better fit). We found, however, that it was more useful to use the percentage of correct predictions for a given model to capture its predictive power.

For both of the regression techniques, unrealistically large coefficients or standard errors of the coefficients are indicative of numerical problems in the analysis. They can indicate multicollinearity that was not removed by our preprocessing, and they can also support the inference that the model has been overfit to the data. We therefore checked for these problem indicators in our models, as well as performing residual analysis to confirm that the underlying assumptions of the regression techniques have not been violated.

For the linear regressions we used four standard regression techniques: including all preprocessed variables in the linear regression (enter method), using backward elimination, using forward selection, and using stepwise regression. We then selected the regression model with the best fit, barring signs of overfitting. Similarly, for the logistic regressions we used three standard techniques: including all preprocessed variables in the regression, using backward stepwise elimination, and using forward stepwise selection. We then selected the model based on the percent of the predictions by the model that are correct, barring signs of overfitting or other numerical problems.

4.3 Predicting Analysis Time

We used linear regression to build the predictive models for analysis time, where analysis time was measured from the native input for each tool. Because analysis time is not meaningful for those analysis cases that failed, the regression only includes analysis cases that did not fail, regardless of whether or not sufficient variables were modeled to check the property. We would expect an analyst to use the predictive models for failure first to check whether or not the analysis will fail, then use the predictive models for analysis time if the analysis is not predicted to fail. Because most of the tools provide automatic checking for deadlock (or, for INCA, a "pre-canned" query), the property metrics are not meaningful for checking deadlock. We therefore generated two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables. The R^2 values for each of the linear regressions are provided in Table 5.

	Enter Method	Backward Elimination	Forward Selection	Stepwise Regression
SPIN, Never Claims				
Deadlock	0.415	0.387	0.387	0.387
Other Properties	0.350	0.333	0.333	0.333
SPIN, Assertions				
Other Properties	0.468	0.452	0.426	0.426
SPIN+PO				
Deadlock	0.218	0.178	0.156	0.156
Other Properties	0.128	0.052	0.052	0.052
SMV				
Deadlock	0.176	0.109	0.109	0.109
Other Properties	0.223	0.178	0.076	0.076
INCA				
Deadlock	0.572	0.537	0.537	0.537
Other Properties	0.902	0.897	0.897	0.897
FLAVERS				
Other Properties	0.500	0.304	0.304	0.957

Table 5. R^2 Values for Linear Regression Models

In general, an R^2 value greater than 0.800 can be considered to indicate a model with a reasonably good fit, and therefore reasonably good predictive power, at least across the domain of the given data set. Table 5 indicates that most of the models generated will not provide good predictions - only the model for INCA (checking other properties) seems to provide a good fit and therefore potentially good predictive power. In this model, which is provided in Figure 4, the QRE States, Never States, and Query Events metrics had the largest effect on predicted analysis time.

Analysis Time = 21.303073 + 7.24986E-08*Cnd' + 0.017519*MaxTRANS - 6.985987*QRE States - 2.814540*Query Events + 3.342872*Never States + 0.577033*Assertions

Figure 4. Predictive Model for INCA Analysis Time, Non-Deadlock Properties

4.4 Predicting Failure

We used logistic regression to build the predictive models for failure. The regression (obviously) included all analysis cases, both those that did and did not fail, regardless of whether or not sufficient variables were modeled to check the property. We note that using all cases for each tool yields different failure percentages than those presented in Section 3; failure percentages for all cases are therefore provided in Table 6. As discussed above, the property metrics are not meaningful for checking deadlock. We therefore generated two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables. The percent of correct predictions for each of the selected models are provided in Table 7.

	Deadlock			Other Properties		
	Failures	Total Cases	Failure Percentage	Failures	Total Cases	Failure Percentage
SPIN	15	120	12.5	39	180	21.7
SPIN+PO	10	120	8.3	34	180	18.9
SMV	11	120	9.2	13	180	7.2
INCA	1	120	0.8	0	180	0.0
FLAVERS	-	-	-	24	140	17.1

Table 6. Failure Percentages, All Cases

	Deadlock			Other Properties		
	Failure	Success	Total	Failure	Success	Total
SPIN	73.3	99.1	95.8	74.4	96.5	91.7
SPIN+PO	80.0	99.1	97.5	82.4	97.3	94.4
SMV	72.7	98.2	95.8	46.2	100.0	96.1
INCA	0.0	99.2	98.3	-	-	-
FLAVERS	-	-	-	66.7	94.8	90.0

Table 7. Percents Correct for Selected Failure Models

To illustrate how to interpret the data in the Table 7, we consider the percentages for the model using SPIN to check for deadlock. The percentage for failure indicates that 73.3% of the time the model predicts a failure that actually occurs in practice and 26.7% of the time the model predicts success when a failure actually occurs in practice. The percentage for success indicates that 99.1% of the time the model predicts a successful (non-failure) analysis that actually occurs in practice and 0.9% of the time the model predicts a failure when a success actually occurs in practice. The total percentage indicates that 95.8% of the time the model predicts success or failure correctly.

As we tried to run these regressions with the statistical package SPSS, we often encountered numerical problems, especially with the enter and backward elimination methods. When we encountered such numerical problems, we either selected one of the models that was successfully created or we removed the property metrics from the regression. We plan to investigate the cause of these numerical problems further, and will also consider trying these regressions with other statistical analysis packages.

Metric	SPIN		SPIN+PO		SMV		INCA		FLAVERS
	Deadlock	Other	Deadlock	Other	Deadlock	Other	Deadlock	Other	Other
N	0.08	0.37	0.05	0.21	0.39	-	0.05	-	0.22
C	-	-1.42	-	-1.43	3.69	-	-	-	0.31
Cnd'	-	-1.8E-08	-	-	-4.1E-09	-	-	-	-1.2E-08
Beta	-	-	0.07	0.23	0.10	-	-	-	0.13
MaxTRANS	-	-	7.0E-04	-	9.0E-05	6.0E-04	-	-	1.0E-04
MaxC	-	-0.13	-	-	-2.01	-	-	-	-0.69
WFSA	-	-	-	-	1.6E-19	-	-	-	1.2E-17
T	-	0.27	-	-	0.41	0.05	-	-	0.04
Vars	-	-0.23	-	-	-0.14	-	-	-	0.89
Alpha'	1.3E-07	5.9E-07	-	7.3E-07	-1.0E-04	-	-	-	0.01
Cif	8.0E-09	-	-	-	-6.7E-07	-	-	-	-2.8E-06
QRE Alphabet	-	-	-	-1.37	-	-	-	-	-
Never States	-	-2.82	-	-	-	-	-	-	-
Assertions	-	-	-	-	-	-	-	-	-
QRE States	-	-	-	-	-	-	-	-	-
Query Events	-	-	-	-0.80	-	-	-	-	-
Query Intervals	-	-	-	-	-	-	-	-	-
Constant	-4.04	6.61	-6.78	7.39	-14.58	-4.29	-11.09	-	-4.98

Table 8. Coefficients in Selected Failure Models

Table 7 implies that we can build reasonably good predictive models for failure within our data set. Note that, for INCA (checking other properties), we did not have any failures, so we were unable to build a model in this case. The coefficients for each metric in each of the selected models are provided in Table 8. We note that there does not seem to be a single metric or set of metrics with a large effect on predicted failures that consistently appears in the models.

4.5 Predicting Spurious Results

We used logistic regression to build the predictive models for spurious results. Because a result can not be spurious in an analysis case that fails, the regression only includes analysis cases that did not fail, regardless of whether or not sufficient variables were modeled to check the property. We note that using all non-failure cases for each tool yields different spurious result percentages than those presented in Section 3; spurious result percentages for all non-failure cases are therefore provided in Table 9. As discussed above, the property metrics are not meaningful for checking deadlock. We therefore generate two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables. The percent of correct predictions by the selected models are provided in Table 10. We experienced the same numerical problems as in the failure regressions, and we followed the same approach to resolve them.

Table 10 implies that we can build reasonably good predictive models for spurious results within our data set, though we note that the percents correct are not quite as high as those for the failure models. In all of the models, the number of variables modeled in the analysis had by far the largest effect on whether or not spurious results were predicted. This is not surprising, however, because we added variable modeling explicitly to remove spurious results.

	Deadlock			Other Properties		
	Spurious Results	Total Cases	Spurious Percentage	Spurious Results	Total Cases	Spurious Percentage
SPIN	35	105	33.3	11	141	7.8
SPIN+PO	38	110	34.5	12	146	8.2
SMV	34	109	31.2	17	167	10.2
INCA	54	119	45.4	23	180	12.8
FLAVERS	-	-	-	13	116	11.2

Table 9. Spurious Result Percentages, All Cases

	Deadlock			Other Properties		
	Spurious	Not Spurious	Total	Spurious	Not Spurious	Total
SPIN	85.7	90.0	88.6	72.7	98.5	96.5
SPIN+PO	81.6	90.3	87.3	75.0	97.8	95.9
SMV	76.5	92.0	87.2	58.8	96.7	92.8
INCA	92.6	96.9	95.0	65.2	96.8	92.8
FLAVERS	-	-	-	69.2	97.1	94.0

Table 10. Percents Correct for Selected Spurious Result Models

4.6 Summary of Predictive Modeling Results

Before performing our regression analyses, we preprocessed our data to remove metrics that were collinear with others since this collinearity can cause problems in both the linear and logistic regression techniques. This preprocessing reduced the number of program metrics included from 26 to 11, and reduced the number of property metrics from 9 to 6. We conducted randomization tests to ensure we have not removed metrics with apparent (but not real) collinearity; the results of these tests indicate that we only removed metrics that were truly collinear in this dataset.

The results of our linear regressions are disappointing. We used a threshold of 0.800 for the R^2 value to indicate a good fit, and 9 out of the 10 linear models we built have R^2 values less than 0.54. Because these models do not capture much of the variance in the experimental data, they are unlikely to provide good predictive power for real programs. We also checked to see if one or more of the metrics commonly appear in the models, indicating that there are certain characteristics of the program or property that affect the analysis times for all the tools. We find no such common characteristics in the linear regression models.

The results of our logistic regressions to predict failure of analysis runs are more encouraging. For all our selected predictive models for failure, the overall percent correct value was greater than 90%. This indicates that these models may provide reasonable predictive power for real programs. We again do not find any common characteristics that appear in all the models.

The results of our logistic regressions to predict spurious results for analysis runs are also encouraging, with all our selected predictive models having overall percent correct values greater than 87%. Again, this implies that these models may provide reasonable predictive power for real programs. All of the models had the strongest effect on the results from the number of variables modeled. This is not surprisingly, because when an analysis run yielded a spurious result, we added additional variable modeling to try to improve the accuracy of the analysis.

5 CONCLUSIONS

We have conducted an experiment comparing a number of static concurrency analysis tools, checking several properties on each of several Ada programs appearing in the concurrency analysis literature. This paper describes the results of that experiment, as well as a number of important issues and tradeoffs we faced during its design.

To ensure that an empirical comparison is valid, the experiment must be designed with great care. While an empirical comparison of static concurrency analysis tools appeared to be a relatively straightforward endeavor, it was surprisingly difficult to design an unbiased experiment. In this paper, we have presented the significant issues and tradeoffs involved in designing such an experiment. These issues include the difficulties associated with selecting programs for the experiment, ensuring each tool analyzes the same program, selecting realistic properties and ensuring each tool checks the same property, avoiding or at least recognizing biases introduced in the experiment, and deciding what to measure and how to compare and interpret those measures.

The results of our experiment have changed our future plans. While our predictive models for failure and spurious results were encouraging, our predictive models for analysis time were disappointingly weak. We believe that it will be difficult to develop such predictive models because the performance of each tool seems to be so sensitive to a wide variety of issues and interdependencies. A future goal was to perform case studies on several "real" programs we have acquired. We now believe that most of the tools will not be robust enough to handle such programs and that conducting such case studies will probably entail a large amount of effort trying to generate representations of the programs that the tools can successfully use.

Empirical comparisons of static concurrency analysis tools are essential to determine which techniques are useful and to provide valuable feedback to researchers in concurrency analysis. Carrying out such a comparison initially appeared to be fairly straightforward, avoiding many of the most difficult problems plaguing empirical work in software engineering. Even this relatively straightforward experimental work, however, turns out to involve a large number of difficult issues and to require tradeoffs between conflicting goals. Despite these problems, such work is extremely important. We believe our findings, as reported in this paper, should help others undertake similar studies.

Acknowledgments

We thank Gleb Naumovich, Jay Corbett, Patrice Godefroid, and Rajesh Prabhu for their helpful contributions to our experiment. This research was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under contract F30602-94-C-0137, and by NSF grant CCR-9407182.

References

[Agr84] Alan Agresti. *Analysis of Ordinal Categorical Data*. John Wiley & Sons, New York, 1984.

[ACD+94] G.S. Avrunin, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automatic derivation of time bounds in uniprocessor concurrent systems. *IEEE Transactions on Software Engineering*, 20(9):708-719, 1994.

[Bla70] Hubert M. Blalock, Jr. Correlated Independent Variables: The Problem of Multicollinearity. In Edward R. Tufte, editor, *The Quantitative Analysis of Social Problems*. Addison-Wesley Publishing Company, Massachusetts, 1970.

[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.

[Cha96] A.T. Chamillard. An empirical comparison of static concurrency analysis techniques. Ph.D. Dissertation, University of Massachusetts, Amherst, 1996.

[Coh95] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, Massachusetts, 1995.

[CA95] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97-123, 1995.

[Cor94] James C. Corbett. An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 228-239, Seattle WA, August 1994.

[Cor96] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *Transactions on Software Engineering*, 22(3):161-180, 1996.

[DS92] Srinivasarao Damerla and Sol M. Shatz. Software complexity and Ada rendezvous: Metrics based on nondeterminism. *Journal of Systems and Software*, 17(2):119-127, February 1992.

[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 LA, December 1994.

[For88] Ray Ford. Concurrent algorithms for real-time memory management. *IEEE Software*, pages 10-23, September 1988.

[GW91] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third Workshop on Computer Aided Verification*, LNCS vol. 575, pages 417-428, July 1991.

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

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Kad92] R. Kadia. Issues encountered in building a flexible software development environment: Lessons from the Arcadia project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE5)*, pages 169-180, Tyson's Corner VA, December 1992.

[LT93] David L. Levine and Richard N. Taylor. Metric-driven reengineering for static concurrency analysis. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 40-50, Cambridge MA, June 1993.

[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.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[MP82] Douglas C. Montgomery and Elizabeth A. Peck. *Introduction to Linear Regression Analysis*. John Wiley & Sons, New York, 1982.

[NWK85] John Neter, William Wasserman, and Michael H. Kutner. *Applied Linear Statistical Models*. Richard D. Irwin, Inc, Illinois, 1985.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377-390, Stanford CA, June 1994.

[Spr93] P. Sprent. *Applied Nonparametric Statistical Methods*. Chapman & Hall, London, 1993.

[TLP+95] Walter F. Tichy, Paul Lukowitz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science; A quantitative study. *The Journal of Systems and Software*, 28(1):9-18, January 1995.

[Wam85] Gordon Kent Wampler. Static concurrency analysis of Ada programs. Master's thesis, University of California, Irvine, 1985.