Evaluating Deadlock Detection Methods for Concurrent Software

James C. Corbett

Abstract— **Static analysis of concurrent programs has been hindered by the well known state explosion problem. Although many different techniques have been proposed to combat this state explosion, there is little empirical data comparing the performance of the methods. This information is essential for assessing the practical value of a technique and for choosing the best method for a particular problem. In this paper, we carry out an evaluation of three techniques for combating the state explosion problem in deadlock detection: reachability search with a partial order state space reduction, symbolic model checking, and inequality necessary conditions. We justify the method used for the comparison, and carefully analyze several sources of potential bias. The results of our evaluation provide valuable data on the kinds of programs to which each technique might best be applied. Furthermore, we believe that the methodological issues we discuss are of general significance in comparison of analysis techniques.**

Keywords— **concurrency analysis, empirical evaluation, state space reduction, symbolic model checking, inequality necessary conditions, Ada tasking.**

I. Introduction

Ada tasks arm software developers with the power, and dangers, of concurrency. With this power, many systems composed of cooperating agents can be concisely specified, but if the communication protocol used by these agents is faulty, the resulting program may contain concurrency errors such as deadlock or starvation. Concurrent software is increasingly a part of safety critical systems, thus research into methods and tools for increasing the reliability of such software is badly needed [35].

When a concurrent system is modeled as a set of communicating finite-state components, static reachability analysis provides a method for automatically detecting most concurrency errors. Its application in practice, however, has been limited by the *state explosion problem*: the number of states in a concurrent system tends to increase exponentially with the number of processes. Many techniques have been proposed to combat this explosion, including state space reductions [20,29,40,42], symbolic model checking [4, 32], compositional techniques [43], abstraction [6], dataflow analysis [16,30], and integer programming techniques [1, 34].

Although each of the techniques excels on certain examples, complexity results indicate that they are all heuristic—when a concurrent system is modeled as a set of communicating finite-state automata, analysis of even simple properties like deadlock is *PSPACE*-hard [38]. In addition, empirical data comparing the performance of the different techniques are rare. This is understandable given the effort of constructing even a single analysis tool, the variations among the models and input languages used by various existing tools, and the difficulty of conducting a fair and meaningful evaluation. Nevertheless, this kind of data is essential for assessing the practical value of the techniques and in assisting developers in selecting a technique for their particular application. There is growing recognition that software engineering research must focus on "designing and carrying out experiments that yield quantifiable and reproducible results" [39].

In this paper we make two contributions. First, we examine the methodological issues of empirically comparing different deadlock detection techniques for Ada tasking programs. Second, we evaluate the performance of three analysis techniques and report the results of this evaluation. Specifically, we evaluate the efficacy of a partial order state space reduction [20], symbolic model checking [4], and inequality necessary conditions [1] in detecting communication deadlocks of Ada tasking programs. While we found that none of the techniques was clearly superior to the others overall, there was significant variation in the performance of the techniques on particular programs. Our data provide some indication of the kinds of programs to which each technique might best be applied. Although our evaluation is narrow in scope, being restricted to one property of one class of concurrent system, we believe that our basic approach is broadly applicable. Anyone conducting an empirical comparison of automated verification techniques for finite-state concurrent systems would benefit from our experience.

This paper is organized as follows. Section II defines the model of concurrent software used throughout the paper. Section III gives an overview of different approaches to the state explosion problem and provides a brief description of the three techniques evaluated. Section IV discusses the method used for the comparison and discusses the issues that arise in such an evaluation. Section V presents the results of the experiments and draws some conclusions about the relative strengths and weaknesses of the techniques evaluated. Section VI considers several alternative models of concurrent software and explores how such models might have affected the results of our evaluation. Fi-

This research was supported by National Science Foundation grant CCR-9308067.

The author is in the Department of Information and Computer Sciences at the University of Hawai'i at Mānoa, Honolulu, Hawaii 96822 (corbett@hawaii.edu).

A preliminary version of this paper appeared in the Proceedings of the International Symposium on Software Testing and Analysis, 1994.

nally, Section VII summarizes our experience in conducting the evaluation.

II. Model

To make the presentation (and later, the evaluation) more uniform, we adopt a canonical model of concurrent software. There are many different kinds of concurrent systems, ranging from gate level hardware, to asynchronous network protocols, to Ada tasking programs. Here, we chose a model that seemed the most natural for representing the class of systems over which we plan to conduct the evaluation: Ada tasking programs. The selection of the model is one of many issues that can affect the outcome of an evaluation, as we will discuss in Section IV-E.

Formally, we model a concurrent system as a set of communicating finite-state automata (FSAs) $M_1, \ldots, M_n$ where $M_i = (S_i, \Sigma_i, \Delta_i, s_{0,i}, F_i)$ and

- S_i is a set of *local states*
- Σ_i is a set of *actions*
- $\Delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a *transition relation*
- $s_{0,i} \in S_i$ is the *start state*
- $F_i \subseteq S_i$ are the *final states*

We write $(s_i, a, s_i') \in \Delta_i$ as $s_i \xrightarrow{a} s_i'$ where Δ_i is clear. Each task in an Ada program is represented by an automaton M_i. The local states typically represent points of control within a task and may also encode the values of finite-ranged variables that are critical for modeling the synchronization behavior of the task. The concurrent system modeled by this set of automata can be defined as another FSA $M = (S, \Sigma, \Delta, s_0, F)$ where:

- $S = S_1 \times \ldots \times S_n$ are *global states*
- $\Sigma = \bigcup_i \Sigma_i$
- $((s_1, \ldots, s_n), a, (s_1', \ldots, s_n')) \in \Delta$ iff $\forall i = 1, \ldots, n$:

$$(a \notin \Sigma_i \wedge s_i = s_i') \bigvee (a \in \Sigma_i \wedge s_i \xrightarrow{a} s_i')$$

- $s_0 = (s_{0,1}, \ldots, s_{0,n})$
- $F = F_1 \times \ldots \times F_n$

Since we model Ada rendezvous, we may assume tasks synchronize in pairs and thus each symbol $a \in \Sigma$ is in the alphabets of either one or two of the M_i. Symbols in the alphabet of one M_i are called *internal actions* and represent execution of code within the corresponding task; symbols in the alphabets of two M_i are called *communication actions* and represent a rendezvous between the corresponding tasks. A *deadlock state* is a non-final global state with no out transitions[1]. A *terminal state* is a final global state with no out transitions. A *trace* of M is a prefix of a string accepted by M.

To illustrate the techniques, we will use a simple concurrent system comprising two tasks, each of which can either synchronize with the other or perform some internal action. The FSAs used to model this system, as well as the automaton for the system itself, are shown in Fig. 1.

[1]This is not the only definition of deadlock in current use. In particular, some definitions require a cyclic wait.

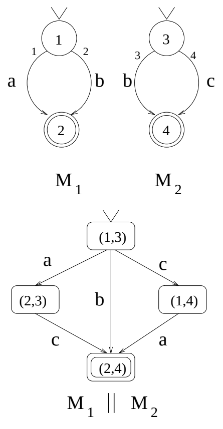

Fig. 1. Example Concurrent System

III. Methods

In this section, we review the basic approaches to the state explosion problem, discuss how we selected the techniques to evaluate, and provide a brief overview of the three techniques selected for evaluation.

A. Approaches to the State Explosion Problem

Many different techniques have been proposed to combat the state explosion problem. Among them:

State-space reductions make a standard reachability analysis more efficient by reducing the number of states that must be explored to verify a property. They range from *virtual coarsening* techniques [29,40], which coalesce internal actions into adjacent external actions, to *partial order* techniques [20,42], which alleviate the effects of representing concurrency with interleaving, to *symmetry* techniques [31,37] which take advantage of symmetries in the state space.

Symbolic model checking [4,32] uses a symbolic representation of a system's states, which is sometimes much more compact than an explicit enumeration. These techniques have proven especially successful in verifying hardware.

Compositional techniques [5,7,43] exploit modularity in a system by dividing it into smaller subsystems, verifying each subsystem, and then combining the results of these analyses to verify the full system. If the subsystems have simple interfaces, such a hierarchical analysis can be quite effective.

Abstraction [6] reduces the number of states by ignoring some state information. For example, static analysis of software often considers the state of a process to consist only of its control location and ignores the values of its variables. The behaviors of this abstract model are a superset of the behaviors of the actual program. This results in a *conservative approximation:* if the technique reports a property holds for all behaviors in the model, then it must hold for all behaviors of the program. If the property does not hold for all behaviors of the model, however, it may or may not hold of all behaviors of the program since the counterexample may be a behavior of the model but not a behavior of the program. Thus an important issue for any conservative analysis is its *accuracy*: how closely do the behaviors of the model match the behaviors of the program.

Data flow analysis can be employed to yield a conservative analysis of a program's properties, from potential cyclic deadlocks [30] to general safety properties [15, 16]. Although these techniques do not really analyze the flow of data in a program, they employ the same algorithms to find the fixed point of a set of flow equations.

Integer programming has been used in the analysis of certain kinds of deadlocks [34] and in a conservative analysis for general safety and liveness properties [1, 12]. These techniques reduce the verification of a property to a question about the integral solutions of linear systems.

B. Selecting Methods to Evaluate

Of all the techniques available, we selected a partial order state space reduction [20], symbolic model checking [4], and inequality necessary conditions [1] to evaluate. We selected these three techniques for several reasons. First, they represent three very different approaches to the state explosion problem and might be expected to perform best on different kinds of systems. Second, they are all "base case" techniques for verifying properties of systems or parts of systems without further decomposition. Compositional and abstraction techniques are part of a divide and conquer strategy for verification. They may be used to simplify the system into manageable pieces, but at the lowest level these pieces must be verified by some other technique. We expect compositional and abstraction techniques to play a vital role in the verification of any large system, regardless of the technique used to verify the base case subsystems. Finally, we selected these three techniques because an implementation of each was available to us, as discussed in Section IV-C.

In the worst case, the time complexity of all of these techniques is exponential in the size of the system [1, 4, 20]. In practice, the techniques may have a much lower complexity on certain kinds of systems. In the remainder of this section, we provide a brief overview of the three techniques selected for evaluation.

C. Partial Order State Space Reduction

The simplest deadlock analysis technique is to enumerate the states M (the automaton representing the system) and search for deadlock states. This is often intractable since the number of states of M is usually an exponential function of n (the number of tasks). One possible solution to this problem is to construct a machine $M' = (S', \Sigma, \Delta', s_0, F')$ with $S' \subseteq S$, $\Delta' \subseteq \Delta$, and $F' \subseteq F$ such that M has some property P (e.g., a deadlock state) if and only if M' has property P. Then we can test for P in M by testing for P in M', which may be much smaller. This general approach is called *state space reduction*.

Some of the most powerful state space reductions use partial orders. These techniques are based on the observation that much of the state explosion is due to the modeling of concurrency with interleaving. For example, if the FSAs in Fig. 1 each perform their internal actions, then the concurrency of these actions would be represented the presense of the traces ac and ca in M. While this produces simple semantics, it greatly increases the size of M.

Partial order semantics represents the actions of a concurrent system as a set of partially-ordered events. Two events are related by the partial order if one must precede the other given the semantics of processes and their interaction. In the example above, a single partial order in which a and c are unrelated represents both possible traces. Properties such as deadlock depend only on the partial order of the events that occur and not on the particular linearization of those events. Thus if M' contains at least one trace for each partial order, we may check for deadlock in M by checking for deadlock in M'. *Conditional stubborn sets* and *sleep sets* are two techniques for constructing such an M'. These techniques involve identifying sets of transitions that "commute" (do not disable each other) at each global state and then firing only one transition from each set. In the example of Fig. 1, $\{((1,3),a,(2,3)),((1,3),c,(1,4))\}$ is one such set of transitions enabled in the system's start state. By firing only one of these transitions, we represent only one possible order of the two events, reducing the number of states generated. See [19, 20] for details.

D. Symbolic Model Checking

Another approach to making deadlock detection more tractable is to use a different representation for M. State-space searches typically generate the states of M explicitly and store them in a hash table. The state space of real systems is often very large, but also very regular. For example, consider a simple system modeling an n-bit counter with states $0, 1, \ldots, 2^n - 1$ and a transition from each state i to state $(i+1) \bmod 2^n$. For $n = 32$, the size of a typical machine register, this system is far too large for state enumeration techniques, yet the transition relation is given "symbolically" by $\Delta = \{(i, a, (i+1) \bmod 2^n)\}$.

One way to represent Δ symbolically is to encode the relation as a boolean function represented by an Ordered Binary Decision Diagram (OBDD) [3]. OBDDs represent many frequently occurring boolean functions very compactly (e.g., symmetric functions, addition). An OBDD for a function $f(x_1, \ldots, x_n)$ and a total order $<$ on the boolean variables $x_1, \ldots, x_n$ is a rooted directed acyclic graph with the following properties. Each internal node v is labeled by some variable $var(v) \in \{x_1, \ldots, x_n\}$ and has exactly two children, denoted $lo(v)$ and $hi(v)$. Each leaf node is labeled by 0 or 1. For any internal node v, $var(v) < var(lo(v))$ and $var(v) < var(hi(v))$ (i.e., along any path from the root, variables appear in order). Finally, given an assignment of values to $x_1, \ldots, x_n$, the value of f is the label of the leaf node reached by traversing a path from the root node, following the arc to $lo(v)$ if $var(v) = 0$ and following the arc to $hi(v)$ if $var(v) = 1$. The OBDD for an example function

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

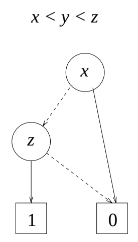

Fig. 2. OBDD

f is given in Fig. 2. Dashed arcs are used to connect v to $lo(v)$, solid arcs to connect v to $hi(v)$.

Any finite set S can be encoded as an OBDD over $\lceil \log_2 |S| \rceil$ variables by assigning each element $e \in S$ a unique n-bit sequence $\sigma(e)$ and building the OBDD for the characteristic function:

$$f(x_1, \ldots, x_n) = \begin{cases} 1 & \exists e \in S.\ \sigma(e) = x_1 \ldots x_n \\ 0 & \text{otherwise} \end{cases}$$

Relations can be similarly encoded since they are simply sets of tuples. The function f in Fig. 2 encodes the transition relation for M_1 in Fig. 1 where x represents the current state, y represents the input, and z represents the next state. The encodings for states 1 and 2 and symbols a and b are 0, 1, 0, 1, respectively.

Given the transition relation Δ, the set of reachable states R is the smallest set of states containing s_0 and including any state that is reachable from a state in R via Δ. The OBDD for R can be computed from the OBDD of Δ with fixed-point techniques that manipulate sets using their characteristic functions encoded as OBDDs. The OBDD for R can be used to check for reachable states with certain properties (e.g., deadlock). In fact, model checking of temporal logic formulas can be performed in this framework without ever constructing an explicit representation of M. See [4,32] for details.

E. *Inequality Necessary Conditions*

Yet another approach to making deadlock detection more tractable is to forgo any representation of the state space. To verify that a system has a property P, the technique generates necessary conditions for the existence of a trace of M violating P. If these conditions are not satisfiable, then M must have property P. If the conditions are satisfiable, however, then M may or may not satisfy P, since the conditions are necessary but not sufficient. If the conditions are strong (i.e., are rarely satisfiable if P holds) and easy to check, then such a technique can be quite effective. Unlike the previous two techniques, however, this kind of technique can yield an inconclusive result (of course, for any technique, an intractable analysis is inconclusive). Different kinds of necessary conditions have been used for deadlock analysis in Ada tasking. Masticola and Ryder [30] used dataflow techniques to search for potential cyclic waits. Here, we consider a more general technique that uses linear inequalities.

$$\begin{array}{rcll}
\textbf{Flow:} & & & \textbf{(state)} \\
1 & = & x_1 + x_2 & (1) \\
x_1 + x_2 & = & 1 & (2) \\
1 & = & x_3 + x_4 & (3) \\
x_3 + x_4 & = & 1 & (4) \\
\textbf{Communication:} & & & \textbf{(symbol)} \\
x_2 & = & x_3 & (\mathrm{b}) \\
\textbf{Hang:} & & & \textbf{(symbol)} \\
x_1 + x_4 & \leq & 1 & (\mathrm{b}) \\
\textbf{Require:} & & & \\
x_1 + x_4 & \geq & 1 &
\end{array}$$

Fig. 3. Inequality System

Necessary conditions in the form of linear inequalities have been used to verify a variety of different properties of concurrent system, including freedom from deadlock [1], general safety and liveness properties [8], and real-time properties [2,11]. The basic idea is to view each FSA M_i as a flowgraph and find a flow from the start state to some final state. This flow represents the path M_i takes in the trace being sought (i.e., the trace violating P). The flow through arc i is represented by an integer variable x_i. Flows are found by generating a flow equation at each state equating the flow into the state with the flow out of the state. There is an implicit flow of 1 into the start state, and an implicit flow of 1 out of the final state. Additional inequalities are generated to enforce some consistency among the paths taken by the FSAs.

Consider the example in Fig. 1. Let action b represent a synchronization between tasks 1 and 2. Let internal action a represent task 1 becoming permanently blocked waiting for task 2 to synchronize on b, and let internal action c represent task 2 becoming permanently blocked waiting for task 1 to synchronize on b. The inequalities representing necessary conditions for the existence of a trace in which some task becomes permanently blocked are given in Fig. 3. The flow equations find flows in each FSA; the communication equation requires that the b communication occur the same number of times in the two tasks; the hang inequality prevents both tasks from becoming permanently blocked waiting for the same communication action (i.e., events a and c cannot both occur); the requirement inequality requires that one task become permanently blocked (i.e., one of events a or c must occur). Notice that the inequalities have no integral solution, proving that deadlock is impossible. See [1,12] for details.

IV. Method for Comparison

The performance of an analysis technique depends on many different factors, including the examples to which it is applied, the property to be verified, the quality of the technique's implementation, the way in which problems are modeled/specified in the input language, and possibly other parameters specific to the particular technique (e.g., the OBDD variable ordering for symbolic model checking). A method for comparison must control all these factors in such a way that the resulting performance data gathered may be meaningfully compared. In this section, we describe a method for empirically evaluating deadlock detection techniques for Ada tasking programs.

A. Selection of Examples

As noted in Section II, there are many different kinds of concurrent systems. We restrict the scope of our evaluation to one kind of concurrent system, Ada tasking programs, for several reasons. First, this allows us to use a very simple interleaving model of concurrency in which only one type of communication must be represented. Asynchronous protocols are more naturally represented using a model with explicit message buffers. Hardware circuits are more naturally represented using models that allow the next state of a component to depend on the states of many other components and allow many components to change state simultaneously. A model that could represent everything would probably represent nothing well. Second, given our limited resources, the results of an evaluation over a limited class of systems is more meaningful since the range of examples provides a better coverage of that particular class. Third, we are familiar with this class of systems and have a collection of examples to use. Of course, this restriction in the class of systems limits the scope of our results. The experiments described in Section V must not be interpreted as an evaluation of the techniques in general, but only as an evaluation of the techniques as applied to Ada tasking programs.

Given this restriction on the class of systems, the technique should be tried on as many examples as possible. Many examples are scalable in some parameter and applying a technique to several sizes of such an example gives some indication of the scalability of the technique. We collected as many real Ada tasking programs as possible and also used standard benchmark examples from the concurrency analysis literature. The examples analyzed are listed in Section V-A. Our choice of Ada reflects a standard in the field of concurrent software analysis [1,14,28,30,43,46].

B. Selecting a Property

We used the techniques to test for deadlock in the communications protocol used by the tasks. We selected deadlock since it is almost always an undesirable property in this setting and is essentially the same for all examples (i.e., a program deadlocks if and only if its automaton contains a deadlock state, as defined in Section II). More complex properties, such as mutual exclusion or starvation, are more system specific and often require more knowledge of a program than is present in its source code. As with the restriction to one class of systems, our restriction to one particular property limits the scope of our results. We note, however, that the verification of any safety property can be reduced to a check for deadlock [21].

C. Implementation

The implementation of a technique can greatly affect its performance. The developer of a technique has a strong incentive to implement the technique as efficiently as possible in order to demonstrate the technique's effectiveness. Therefore, we decided to use tools written by the developers of the techniques to insure a good implementation of the techniques evaluated. The only alternative to using available implementations would have been to implement the techniques ourselves, guided by technical reports describing their details. Besides requiring much more effort, this approach would likely have produced implementations significantly inferior to those crafted by experts with years of experience with a particular technique.

Patrice Godefroid, Didier Pirottin and Pierre Wolper of the University of Liege have implemented a partial order technique [20] as an extension to a very fast protocol analyzer called SPIN, which was written by Gerard Holzmann at AT&T Bell Labs [25]. We refer to SPIN with the Partial Order package installed as SPIN+PO. Note that the latest version of SPIN (version 2.0) supports a different partial order technique which has significantly less memory overhead than the Liege package; it does not yet include the partial order reductions for synchronous communication, however, and so was not used in this evaluation. Kenneth McMillan of Cadence Systems has implemented the Symbolic Model Verifier (SMV) [32]. Both of these tools are publicly available, stable, and reasonably robust. Unfortunately, the Inequality Necessary Condition Analyzer (INCA), like many research tools, has none of these desirable properties, but since we constructed most of the components that are currently in use, this was not a problem. INCA is descended from the constrained expression toolset [1], but it supports more powerful analysis techniques [11,12] and is more efficient.

D. Specifying the Examples

Given our decision to use existing implementations of the techniques, we are faced with a problem: each analysis tool has its own unique specification language. We see two solutions to this problem. We could specify each example directly in each tool's specification language, or we could specify each example in some canonical form and then generate the input for each tool automatically from this canonical form. The first approach is technically simpler since it avoids the tricky issues involved in automatic generation of input. On the other hand, if we specify an example in two different specification languages, on what formal basis can we say that these specifications represent the same system? This issue is more serious than it may appear since specification languages can be very different and slight changes in the way a system is specified can produce large variations in the performance of a tool. We must be sure that specifications of the same example in different languages are formally equivalent before we can meaningfully compare the performance of the techniques on the example. We believe this requires using the second approach.

The next step is to decide on a canonical form for the examples. One possibility is to use the input language for one of the tools as the canonical form. The problem with

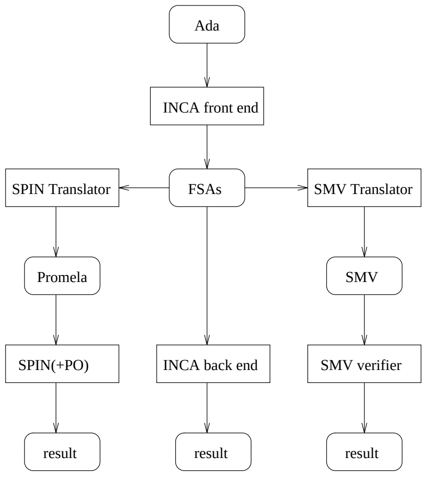

Fig. 4. Generation of Input

this approach is that the semantics of most specification languages are sufficiently complex and varied that translating one into another is very difficult. Each language has constructs that might be awkward to represent in another, thus the tool whose input language is considered canonical might enjoy an unfair advantage in the evaluation. Since all of the tools, regardless of their input language, represent a concurrent system as a finite-state transition system, we instead use the more abstract model of concurrent systems from Section II as the canonical form for all of the examples. This model has simple semantics, thus it is relatively straightforward to embed in the richer semantics of the specification languages.

Given each example specified as a set of communicating FSAs, we use translators to automatically generate semantically equivalent input for the different tools. The translators for SPIN+PO and SMV are described in Sections IV-F and IV-G, respectively. The back end of INCA (which performs the analysis) accepts communicating FSAs directly, thus no translation is required. The communicating FSAs representing each example are generated automatically from an Ada-like specification by the front end of the INCA tool. This aspect of the comparison method is summarized in Fig. 4.

Although it may seem that we are using the input language for INCA as the canonical form, we view the front end of INCA only as a tool for constructing the canonical form. INCA constructs the FSA for each task using standard techniques for constructing control flow graphs [17]; these automata are very similar to those produced (internally) by SPIN or any other tool that constructs a finite-state representation of the control flow of imperative code. Thus we do not believe that using the front end of INCA to produce the communicating FSAs conveys any advantage on the back end of INCA, which performs the analysis. While it is true that the input for INCA does not go through a translator, we could easily have written an INCA translator that reversed the mapping performed by the INCA front end (i.e., converted the communicating FSAs back to the original Ada-like specification) and then applied the whole of INCA to this translated input. Since the same communicating FSAs would be analyzed in either case, however, the performance of INCA would not have changed. The real issue is whether our choice of canonical form introduces a bias; we discuss this issue in Section IV-E.

A description of the algorithm used by the front end of INCA to translate our Ada-like specification language into communicating FSAs is given in [9]. Since the details of this translation are extensive and probably beyond the scope of this paper, here we give only an example of a sample specification and the FSAs generated from it. Fig. 5 shows our Ada-like specification for the basic dining philosophers problem and Fig. 6 shows two of the FSAs generated by the INCA front end from this specification (we selected this example because it has the smallest specification and is probably the most familiar). The constant `Problem_Size` is set by the INCA front end so that varying sizes of the example can be generated from the same source code. The task discriminant `I` is set to the index in the array at which the philosopher/fork task is placed. A rendezvous between two tasks is modeled by a shared symbol that encodes the caller, the acceptor, and the entry. For this example, the FSA state encodes only the syntactic location within the source code. For many examples, the values of a few small ranged variables (e.g., flags or counters) must be encoded into the task state for accurate modeling of the task's synchronization behavior; adding variables to the task specifications causes INCA to perform this encoding.

E. Identifying Potential Bias

Our model of concurrent systems may introduce a bias against SPIN and SMV. While our model is very natural for representing Ada tasking programs, it is not particularly appropriate for representing asynchronous protocols or hardware, the domains for which the other tools were designed. We believe this bias is much worse for the OBDD-based technique for two reasons. First, we use an interleaving model of concurrency rather than a simultaneous model (in which multiple actions can occur simultaneously). OBDD-based techniques generally perform better on simultaneous models, although this depends on the communication structure of the system [32]. Second, the encoding of task variable values within a monolithic task state may increase the size of the OBDDs needed to represent the transition relation of the task's FSA. We elaborate on this second effect.

Large FSAs are generated by *data intensive* tasks containing variables that must be encoded into the state of the task for accurate modeling of the task's synchronization behavior. When a system is directly specified in the SMV input language, SMV can encode the states more efficiently for representation by OBDDs. For example, consider again the n-bit counter with states $0, \ldots, 2^n - 1$, and transition relation $\Delta = \{(i, a, (i+1) \bmod 2^n)\}$. Since addition can be represented very efficiently with OBDDs, the OBDD representing this transition relation would be much smaller if

```
-- Number of philosophers/forks
N : constant := Problem_Size;

type Fork_Range is range 0 .. N - 1;

task type Philosopher (I : Fork_Range);

task type Fork (I : Fork_Range) is
  entry Up;
  entry Down;
end Fork;

Forks : array Fork_Range of Fork;
Phils : array Fork_Range of Philosopher;

task body Philosopher is   -- Philosopher I
begin
    loop
        -- Pick up right, then left fork
        Forks((I + 1) mod N).Up;
        Forks(I).Up;
        -- Eat, put down forks
        Forks((I + 1) mod N).Down;
        Forks(I).Down;
    end loop;
end Philosopher;

task body Fork is    -- Fork I
begin
    loop
        accept Up;   -- Fork picked up
        accept Down; -- Fork put back down
    end loop;
end Fork;
```

Fig. 5. Ada-like Source Code for Dining Philosophers Example

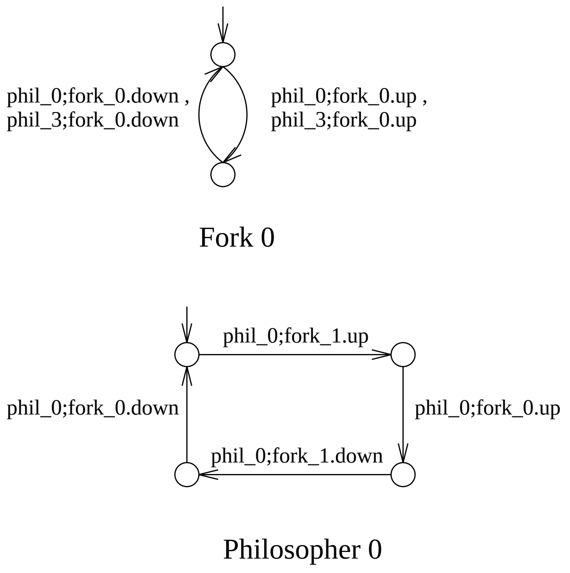

Fig. 6. Two FSAs for Dining Philosophers Example ($N = 4$)

the state's number (in binary) were used by SMV as the state's encoding. INCA, however, might present the states $0, \ldots, 2^n - 1$ to SMV in an arbitrary order, resulting in an arbitrary binary encoding for each state that does not expose the regularity of the transition relation. In practice, if a task has only one variable and its FSA contains essentially one state per value of this variable, then this type of bias is not introduced since the states are presented to SMV in order. However, if the task has multiple variables, then the position of a state in some ordering chosen by the translator does not reveal the values of the variables encoded in the state, and thus introduces the bias.

```
/* Message type */
mtype = { synch };

/* Rendezvous channel b */
chan b = [0] of { byte };

proctype M1()    /* FSA 1 */
{
state1:
  if
  /* internal action a */
  :: skip -> goto endstate2
  /* rendezvous on channel b */
  :: b?synch -> goto endstate2
  fi
endstate2:
  0 ;        /* halt */
}

proctype M2()   /* FSA 2 */
{
state3:
  if
  /* rendezvous on channel b */
  :: b!synch -> goto endstate4
  /* internal action c */
  :: skip -> goto endstate4
  fi
endstate4:
  0 ;       /* halt */
}

init { atomic { run M1(); run M2() }}
```

Fig. 7. Promela Generated by Translator

The model was chosen to match the domain of the examples, not the analysis tools. Since we are conducting the evaluation over a domain for which SPIN and SMV were not designed, the introduction of some bias seems unavoidable. INCA was built to analyze Ada tasking programs, thus it does not suffer from this bias. In Section VI, we explore the extent of the bias by considering different models.

F. SPIN Translator

Here, we describe the how the communicating FSAs representing a concurrent system were translated into Promela, the input language for SPIN. Promela is a guarded command language like CSP [24], with a C-like syntax. It directly supports communicating processes, thus the translation is relatively straightforward.

Each FSA is represented by a process created using the `proctype` declaration. Each state of the FSA is represented by a statement labeled with the name of the state. The statement implements the transitions out of the state it represents using `goto` statements guarded by the action causing the transition. Fig. 1 shows a pair of communicating FSAs and Fig. 7 shows the Promela code generated for these FSAs (with some comments added by hand).

The `if` statement in Promela is like the alternative command in CSP, allowing an arbitrary selection among alternatives whose guards are true. There is one alternative for each transition out of the state. If the state has no out transitions, then the statement for that state is 0, which is not executable, causing the process to halt. Each alternative representing a transition on action A to state i consists of a `goto` statement to state i's label guarded by a statement representing A. Internal actions are always enabled and are represented by the statement `skip`. Communication actions are discussed below. If a state label begins with the string "end", then it is considered a final state of the process for purposes of deadlock detection.

Promela supports communication between processes via channels which are declared like ordinary variables having type `chan`. Channel variables must be initialized, indicating the number of messages that the channel can buffer. We use channels with zero capacity to implement synchronous communication. We declare such a channel for each communication action in the FSAs. Each communication action is shared by two processes, one of which is designated the sender, and the other, the receiver. The Promela statements `c!m` and `c?m` are used to send and receive a message of type `m` to/from a channel `c`, respectively. Since the channels have zero capacity, these statements are executable (i.e., may be chosen in the `if` statement for a state) only when both processes containing the corresponding communication action are in states in which they can take that action. No information is passed in the messages, thus we declare and use a single message type `synch` using the `mtype` declaration.

Promela also supports the specification of safety and liveness properties. We do not generate such specifications in our translation, causing SPIN (and SPIN+PO) to search only for invalid endstates (i.e., deadlocks).

G. SMV Translator

We now describe how a set of communicating FSAs is translated into the SMV input language. Unlike Promela and Ada, the SMV input language was designed primarily for hardware specification. In the SMV language, systems are specified as a set of variables and a set of functions that define the next value of those variables in terms of the current values. Facilities for constructing, replicating, and connecting components are also provided. While this is convenient for specifying gate-level logic and even certain kinds of protocols, we found it awkward for specifying communication between sequential processes. Fortunately, the SMV language also provides an escape that allows the transition relation to be specified directly. This was meant specifically to facilitate writing translators from other languages into SMV and proved invaluable for this comparison.

The SMV specification we generate has four parts. The `VAR` part declares the state variables. A *state* is an assignment of values to the state variables. The `INIT` part is a boolean function of the state variables. A state is a legal initial state of the system if this function is true for that state. The `TRANS` part allows any transition relation to be specified. It is comprised of a single boolean function of

```
MODULE main
VAR
  x1 : { 1, 2 };    -- State of FSA 1
  x2 : { 3, 4 };    -- State of FSA 2
INIT   -- Initial state
  (( x1 = 1 ) & ( x2 = 3 ))
TRANS
  (  -- Transition on 'a'
     ( ( ( x1 = 1 )
         &
         ( next(x1) = 2 ))
       &
       ( x2 = next(x2) ))
    | -- Transition on 'c'
    ( ( x1 = next(x1) )
      &
      ( ( x2 = 3 )
        &
        ( next(x2) = 4 )))
    | -- Transition on 'b'
    ( ( ( ( x1 = 1 )
          &
          ( next(x1) = 2 ))
        &
        ( ( x2 = 3 )
          &
          ( next(x2) = 4 )))))
SPEC
  AG    -- On all paths globally:
     ( EX 1  -- There is a next state
       |     -- OR all FSAs in final
       ( x1 = 2 & x2 = 4))  -- states
```

Fig. 8. SMV Generated by Translator

the values of the state variables in the current state and their values in the next state. If this function returns true for a pair of states (s, s'), then there is a transition from state s to state s'. Finally, the `SPEC` part is a temporal logic formula in the logic CTL [32] specifying a property that the system must satisfy.

We represent a set of communicating FSAs as follows. For each M_i, we declare a state variable x_i to hold its current state. The `INIT` function $\bigwedge_i (x_i = s_{i,0})$ forces each M_i to begin in its start state. The `TRANS` function encodes the transition relation of the entire system in one large boolean formula; its construction is described below. Finally, the `SPEC` formula `AG ((EX 1) | FINAL)` is used to search for deadlocks, where `FINAL` is a formula that is true if the state is a final state. In CTL, this translates as "Along all paths always, either there exists some next state, or the current state is final". Formula `FINAL` is given by

$$\bigwedge_i \left(\bigvee_{s \in F_i} x_i = s \right)$$

The main part of the translation is the construction of the `TRANS` formula. We use an interleaving model in which at most one transition can occur at a time. The `TRANS` function will consist of a disjunction of all possible global transitions the concurrent system can make. Global transitions are of two types: internal actions (in which one M_i changes state) and communication actions (in which two M_i change state). For each state variable x, $next(x)$ represents the value of x in the next state, while x represents its value in the current state. The disjunct representing the occurrence of internal action transition $s \xrightarrow{a} s'$ in M_i is

$$x_i = s \wedge next(x_i) = s' \wedge \bigwedge_{j \neq i} x_j = next(x_j)$$

This requires that M_i move from state s to state s' while all other M_i remain in the same state. Similarly, the disjunct representing the occurrence of communication action transition $s \xrightarrow{a} s'$ in M_i synchronously with the communication action transition $q \xrightarrow{a} q'$ in M_j is

$$\begin{aligned} x_i = s \quad & \wedge \quad next(x_i) = s' \\ & \wedge \quad x_j = q \wedge next(x_j) = q' \\ & \wedge \quad \bigwedge_{k \neq i,j} x_k = next(x_k) \end{aligned}$$

Each possible pairing of matching communication action transitions produces a distinct global transition. The SMV input generated for the system in Fig. 1 is shown in Fig. 8. SMV uses & and | for *and* and *or*, respectively.

V. Experiments

This section presents the results of the experiments conducted using the comparison method described in Section IV. We applied the SPIN+PO, SMV, and INCA to seventeen families of scalable examples and to several non-scalable programs, measuring the amount of time and memory used by the tools to check for deadlock. For each scalable example, we applied each tool to several sizes of the example to gauge the scalability of the technique. We also applied SPIN, a straight reachability analyzer, to each example to provide a baseline for measuring the effectiveness of the other tools in curbing the state explosion problem. From the raw performance data for the scalable examples, we derived a numerical measure of how fast the resource requirements grew with the problem size. Using these growth rates and the raw data for the non-scalable examples, we were able to correlate the scalability of each technique to certain features of the examples and thus characterize the kinds of programs to which each technique might best be applied.

This section is organized as follows. Section V-A presents a brief description of the examples used in the evaluation. Section V-B discusses some issues that arose in modeling these examples for analysis. Section V-C describes the general approach we used (i.e., what analyses should be run). Section V-D presents the raw data from the analyses and the details of its collection. In Section V-E, we develop a numerical measure for the rate of growth of the time/memory required by the tools as an example is scaled up. Finally, in Section V-F, we use these growth rates to draw conclusions about the scalability of the various techniques on different kinds of programs.

A. Examples

This section lists the examples analyzed. We use m as the size parameter for scalable examples, and denote the size m version of scalable example X as X(m). Space permits only a brief description of each example. The Ada-like specifications of these examples, the communicating FSAs generated from them by the INCA front end, and the Promela and SMV inputs generated by the translators are available for anonymous FTP on `ftp.ics.hawaii.edu` in `/pub/corbett/eval.tar.Z`.

Some raw characteristics of each example are summarized in Table I, whose columns give the lines of code, the number of tasks, the number of unique rendezvous, and some additional features used in our analysis of the results in Section V-F. The lines of code given is for the Ada-like input to INCA; this is not an accurate measure of the size of the resulting system since all the scalable examples use arrays of tasks, and thus this measure does not vary with the problem size. For specifications extracted from real programs, we give the size of our specification rather than the size of the original program since the amount of unmodeled non-concurrent code is not relevant to our analyses. We count each distinct shared symbol as a unique rendezvous; this symbol encodes the caller, the acceptor, the entry name, and any parameters passed in the rendezvous.

The examples analyzed were:

Alternating Bit Protocol (ABP) A simple but often analyzed example modeled with 6 tasks representing two users, a sender, a receiver, and two lossy channels.

Border Defense System (BDS) This example, analyzed in [14,30], is the communication skeleton of a real Ada tasking program that simulates a border defense system. The original source code was written by T. Griest of LabTek Corporation and comprised 11K lines of Ada. We obtained the communication skeleton from the Concurrent Systems Software Laboratory at the University of Illinois at Chicago (as indicated in Table I, the skeleton is only 247 lines). The example has 15 tasks, but the skeleton of each is relatively simple.

Cyclic scheduler (CYCLIC) Milner's cyclic scheduler [7, 33] uses m scheduler tasks to keep m customer tasks loosely synchronized.

Divide and Conquer (DAC) A program modeling a divide and conquer computation by forking up to m solver tasks that proceed in parallel [2,11].

Dartes Program (DARTES) The communication skeleton of a fairly complex Ada program with 32 tasks [30].

Dining Philosophers (DP, DPH, DPD, DPFM) Although not a very realistic problem, it does contain a nontrivial deadlock and is probably the most commonly analyzed example [1,14,28,42,46]. We model each of the m philosophers and m forks with a task. These tasks synchronize to model forks being acquired and released. In addition to the standard version (DP), which can deadlock, we also analyzed several versions of the problem where deadlock is prevented. In the version with host (DPH), there is an additional host task with which a philosopher must synchronize before attempting to acquire her forks. This might model a real-world situation in which a task wishing

Problem	LOC	Tasks	Rends	Communication Structure	Task Size/ Structure
CYCLIC(m)	54	$2m$	$3m$	linear	$2m$ small
DAC(m)	41	m	$2m$	linear	m small
DP(m)	34	$2m$	$4m$	linear	$2m$ small
DPD(m)	47	$2m$	$5m$	linear	$2m$ small
DPFM(m)	50	$m+1$	$2m$	single star	m small, 1 large
DPH(m)	62	$2m+1$	$6m$	linear + single star	$2m$ small, 1 medium linear
ELEVATOR(m)	278	$m+3$	$11m+9$	multiple stars	$m+2$ medium, 1 large
FURNACE(m)	196	$2m+6$	$8m+6$	multiple stars	$2m+5$ small, 1 medium
GASNQ(m)	98	$m+3$	$12m+2$	multiple stars	$m+2$ medium, 1 large
GASQ(m)	141	$m+3$	$10m+2$	multiple stars	2 small, m medium, 1 large
HARTSTONE(m)	55	$m+1$	$2m$	single star	m small, 1 large linear
KEY(m)	388	$m+5$	$13m+8$	single star	3 small, $m+2$ medium
MMGT(m)	202	$m+4$	$5m+4$	multiple stars	4 small, m medium
OVER(m)	79	$2m+1$	$11m-4$	linear + single star	$m+1$ small, m medium
RING(m)	60	$2m$	$5m$	linear	m small, m medium
RW(m)	65	$2m+1$	$4m$	single star	$2m$ small, 1 medium linear
SENTEST(m)	198	$m+4$	$m+9$	single star	$m+3$ small, 1 large linear
ABP	166	6	18	linear	5 small, 1 medium
BDS	247	15	25	multiple stars	14 small, 1 medium
DARTES	1228	32	205	multiple stars	25 small, 7 medium
FTP(1)	572	9	76	multiple stars	3 small, 6 medium
FTP(2)	572	10	102	multiple stars	3 small, 7 medium
Q	615	18	59	multiple stars	10 small, 8 medium
SPEED	245	10	10	multiple stars	10 small

TABLE I
Characteristics of Examples

to use a resource must first get permission from a central server task. The host will allow at most $n-1$ philosophers to hold forks at any one time. In the dictionary version (DPD), the same effect is achieved by having the philosophers pass a dictionary around the table. The philosopher holding the dictionary cannot hold any forks. Finally, in the version with a fork manager (DPFM), philosophers pick up both forks simultaneously by rendezvous with a fork manager task, which records the state of all forks in lieu of the fork tasks.

Elevator(ELEVATOR) This program models a controller for a building with m elevators, using tasks to model the behavior of the elevators themselves. The size m version has $m+3$ tasks.

File Transfer Program(FTP) This program services requests from m users to transfer files over a network. Our version is an abstraction of the original program, which was too complex to analyze. The size m version has $m+8$ tasks.

Remote Furnace Program(FURNACE) This program manages temperature data collection for m furnaces. We analyze the original design presented in [45] abstracted slightly (e.g., we do not model the temperature data using the furnace identifier since we are only verifying freedom from deadlock, not proper transmission of data). The size m version has $2m+6$ tasks.

Gas Station(GASNQ, GASQ) This example, which models a self-service gas station, originated in [23] and has been analyzed in [1, 14, 28, 40]. Customers arrive and pay the operator for gas. The operator activates a pump, at which the customer then pumps gas. When the customer is finished, the pump reports the amount of gas actually pumped to the operator, who then gives the customer her change. We analyzed versions with one operator task, two pump tasks, and m customer tasks. We analyzed two different versions of this example. In the original version (GASQ) from [1, 23], the operator task queues customer requests and must keep track of which customers are waiting for each pump and in what order. In the non-queuing version (GASNQ) from [14], the operator does not enforce a first-come-first-serve order on the customers and must only record the number of customers waiting for each pump in order to activate the pump when any waiting customers remain.

Hartstone Program (HARTSTONE) The communication skeleton of an Ada program analyzed in [30] in which one task starts and then stops m worker tasks.

Keyboard Program (KEY) The communication skeleton of an Ada program analyzed in [30] that manages keyboard/screen interaction in a window manager. We scaled the program by making the number of customer tasks a parameter (m). The size m version has $m+5$ tasks.

Distributed Memory Manager (MMGT) The communication skeleton of an Ada program implementing the memory management scheme from [18] with m users. The size m version has $m+4$ tasks.

Overtake Protocol (OVER) An Ada version of an automated highway system overtake protocol in [22] for m cars comprising $2m+1$ tasks.

Q User Interface (Q) The Ada skeleton of an RPC client/server-based user interface with 18 tasks that is used by several real applications.

Token Ring Mutual Exclusion Protocol (RING) An Ada implementation of a standard distributed mutual exclusion algorithm in which m user tasks synchronize access to a resource though m sever tasks that pass a token around a ring.

Readers and Writers(RW) This example models a database that may be simultaneously accessed by any number of readers or a single writer. Each of the m reader tasks and m writer tasks must synchronize with a controller task before accessing and when finished accessing the database. This system has been analyzed in [1,14,28].

Sensor Test Program (SENTEST) The communication skeleton of an Ada program analyzed in [30] that starts up m tasks to test sensors. The size m version has $m + 4$ tasks.

Speed Regulation Program (SPEED) The communication skeleton of an Ada program analyzed in [30] with 10 tasks that monitor and regulate the speed of a car.

B. Modeling Issues

For a few of the examples (DPFM, ELEVATOR, GASNQ, GASQ), it is convenient for a task accepting an entry to be able to test the value of a call parameter before deciding whether to accept the call. This functionality is absent from Ada 83, although the `requeue` statement of Ada 95 effectively adds it. The same effect can be achieved in Ada 83 by using a different entry for each value of the critical parameter, though this often makes scaling the specification cumbersome. In our specifications, we employ a special `assume` statement for this purpose. For example, the operator task in GASNQ should not accept the CHARGE entry for a pump for which no customers are currently waiting. This can be expressed as:

```
accept CHARGE ( P : PUMP) do
  assume WAITING(P) > 0;
  ...
end CHARGE;
```

The `assume` statement is modeled after the `assuming` clause used by Yeh and Young in [44].

Although we believe all of our examples except the standard dining philosophers (DP) are free of deadlocks, our models of the Ada programs DARTES, KEY, and SPEED contain spurious deadlocks due to the presence of a global variable used for synchronizing task termination. Currently, global variables are not processed by the INCA front-end, although they can be represented in our model of concurrent systems using an additional FSA for each variable to hold the value of that variable, along with communication actions for testing and setting the value of the variable. Our model of the Q program also contains a spurious deadlock due to the abstraction of timing information (the program makes heavy use of conditional entry calls). Since we had so few examples that contain deadlocks, we chose to leave these spurious deadlocks in the models to provide more data on how quickly the tools can find a deadlock when one exists. One drawback is that these deadlocks are not subtle—unlike the deadlock in the dining philosophers problem, these deadlocks would be likely be found by random simulation.

Several of the examples (DAC, ELEVATOR, HARTSTONE, KEY, Q, SENTEST, SPEED) used the Ada `terminate` alternative to synchronize the termination of groups of tasks. In all of these examples, all tasks are declared within a single package; thus a task will select its `terminate` alternative exactly when all other tasks are either terminated or similarly blocked on `terminate` alternatives. We represent this in our model by making a state of a task FSA that can select a `terminate` alternative a final state.

C. General Approach

We ran SPIN+PO, SMV, and INCA on the examples described in Section V-A. We also ran SPIN, a straight reachability analyzer, on each of the examples to give a baseline for measuring the efficacy of the techniques in curbing the state explosion problem. For all examples, we measure the CPU time and memory consumed by the tools in performing the analysis.

For each scalable example, we selected four arithmetically increasing sizes ending near the maximum size that could be handled by all of the tools. This facilitates comparison of the tools, although it makes most of the measured run times small since the maximum size is set by the tool that performs worst on that example. The step value for the size growth was chosen to magnify the variation in the resource measurements. For most examples, this meant dividing the size range into roughly equal pieces (e.g., if the maximum size is 12, run sizes 3, 6, 9, 12). For a few examples, however, the resource requirements for one or more tools increased very quickly with the size and were thus very small until the size approached the maximum. For such examples, we chose to use larger sizes to minimize the number of small measurements, which are dominated by fixed overhead.

Rather than finding the largest size each tool can handle given certain resource constraints, as we did in the preliminary version of this paper [10], we simply measure the growth in the resources consumed as the example is scaled up (the calculation of these growth rates is described in Section V-E). We believe these growth rates are more meaningful than the maximum sizes gathered in [10] for several reasons. First, various kinds of constant overhead in the implementations are factored out. Second, for some of the examples, the use of translated input, not the tool itself, imposes the maximum size (see the discussion of the difficulty in scaling HARTSTONE and SENTEST in Section V-D). For larger sizes of such examples, either the INCA front end runs out of memory building the FSAs or (more often) the translated input, being much larger than a native specification, is too large for the tool. This is a limitation of our comparison method, although a more compact canonical model (e.g., the EFSAs of Section VI-B) and better translators, which use language constructs like Promela arrays to replicate components, would largely solve this problem. Third, the resource constraints, although reasonable, are somewhat arbitrary, especially the limit on CPU time. In [10] we used three hours, though one could argue for a higher or lower limit.

In Section V-E, we will investigate the relationship between the size of the concurrent system and the resources consumed by the tools to perform the analysis. Given a concurrent system in the model of Section II, we must define exactly what the size of this system is. We might use the number of reachable states in the model ($|S|$), but this obscures the state explosion by making the size measure itself explode as systems are scaled up. Also, it bears little relation to the size of the Ada program from which the model was derived. Alternatively, we might use the number of tasks (n). This is closer to the programmer's view of a program's size, but does not take into account the size of the tasks (e.g., the 2-state fork tasks in DP are counted the same as the 1400-state operator task of GASQ(4)). To account for task size, we might use the number of component states ($\sum_i |S_i|$) as the measure, but this obscures the state explosion that results from the use of data within a component (as in the gas station and elevator examples). Instead, we use the number of bits required to store the state of the task as a measure of the task's size, and the sum of these measures ($\sum_i log_2|S_i|$) as a measure of the size of the program. To avoid confusion, we call this measure of size the *scale* of the example and continue using the word "size" to denote the value of m in the scalable examples. We note that, for all of our examples, the scale is a linear function of the size, thus an arithmetic sequence of sizes of an example will have an arithmetic sequence of scales.

D. Raw Data

In this section, we present the data from our experiments along with various details of how they were collected.

All experiments were conducted on a SPARCstation 10 Model 51 with 96 MB of memory. Analysis times are reported in user CPU seconds collected using the `time` command of `tcsh` and the `(get-internal-run-time)` function of Allegro Common Lisp. Statistical analysis of the behavior of these functions reveals that the time they report has a near normal distribution with a standard deviation around 0.06. If a tool takes very little time on all sizes of an example, these small variations can have a significant effect on the growth rate we calculate in Section V-E. Therefore, if the measured run times of a tool on all sizes of an example are less than two seconds, then we run the tool 100 times and use the average time as a point estimate. For the INCA results, the analysis times reported include the translation from an Ada-like source language to the FSAs. For the other tools, the analysis times include only the actual run times of those tools—not the time to translate the Ada-like source to FSAs, nor the time to translate the FSAs into the tool's input language.

The tools themselves report the amount of memory they used; we trust these figures, but regard them as approximations. Memory usage is very difficult to measure externally since a tool will generally allocate more memory than it actually uses. Also, there is some variation among the tools as to what memory (i.e., code, stack, heap) is counted in the total reported. These differences are small constant factors, however, and do not significantly affect the rate of growth of memory usage as examples are scaled up.

For these experiments, we used version 1.5.10 of SPIN, version 2.0 of the partial order package for SPIN, an unofficial version of SMV dated 8/6/93, and version 3.2 of INCA. SPIN, SPIN+PO, and SMV are all written in C. INCA is written in Common Lisp, with the integer programming package written in FORTRAN.

SPIN+PO and SMV take various command line parameters that can affect their performance. We used the parameters suggested by the authors of those tools, which we found produced the best performance. SPIN+PO was run with the "proviso" disabled (the `-p` flag). The proviso causes SPIN+PO to generate more states than are needed for deadlock detection in order to allow state assertion checking. Since we are evaluating deadlock detection only, this is not needed and removing it improves the performance of the tool. SMV was run with the `-f` flag, which calculates the reachable states of the system before checking the CTL formula.

SPIN and SPIN+PO use arrays whose sizes are set by various command line parameters, including: the maximum number of processes, the size of the state vector, and the maximum search depth. While the default values for these parameters sufficed for most of the examples, we had to increase them to complete the analysis of some of the examples. The maximum number of processes (default: 32) was raised to 34 for DARTES and RW, and to 110 on HARTSTONE and SENTEST. The state vector size (default: 1024) was raised to 2048 on DARTES, HARTSTONE, and SENTEST. The maximum search depth (default: 10K) was raised to 100K on DP (for SPIN only). Parameters were raised uniformly for all sizes of a scalable example. This increased the memory usage unnecessarily for smaller sizes of those examples, but we feel that this is more consistent since we do not vary these parameters for the different sizes of the other scalable examples. In our analysis of performance in Section V-E, we will be concerned primarily with growth rates rather than magnitudes.

For SPIN and SPIN+PO, the analysis tool first generates C source code for an analyzer which is then compiled and run to perform the analysis. We do not include the generation and compilation times in our data. The translated input we generate is much larger than the equivalent Promela code that would be used to specify the same system, and takes much longer to generate and compile (e.g., for GASNQ(5), a native Promela specification takes 5 seconds to generate and compile, compared to 62 seconds for the translated input). Thus the real run times of these tools would be slightly larger.

The HARTSTONE and SENTEST programs have a very simple communication structure. Indeed, the number of states in these examples grows linearly with the problem size (m). We had difficulty scaling these examples to the limits of the tools, however, due to our use of translated input for SPIN and SPIN+PO. As noted above, the translated input is usually much larger than the equivalent native Promela specification. For large sizes of HARTSTONE and SENTEST, the generated analyzer was too big for our C compiler, and thus could not be run.

For SMV, the order in which the state variables are declared in the input file is the order in which they appear in the OBDDs. Since this greatly impacts the performance of the technique, we assisted the tool by providing a reasonable ordering for each example based upon our knowledge of OBDDs and some limited trials with different possible orderings. Unfortunately, there is no general algorithm to determine the best order for a particular situation, although heuristics exist. In general, the state variables for tasks that communicate were placed as close together as possible.

The examples DP, DARTES, KEY, Q, and SPEED contain deadlocks. For these systems, SPIN, SPIN+PO, and SMV display the sequence of state changes leading up to the deadlock state, and INCA finds a solution that is interpreted as a sequence of actions of each task. For all other examples, the tools correctly declared that the system was free of deadlocks.

The raw performance data are given in Tables II–III. The columns of the table show the problem name and size (if scalable), the scale ($\sum_i log_2|S_i|$) rounded to the nearest integer, the number of reachable states ($|S|$), and the memory (in megabytes) and CPU time (in seconds) consumed by the four tools to perform the analysis. A dash indicates the analysis could not be completed with 96 MB of memory and a day of CPU time. We used SPIN to generate the number of reachable states for each example except DARTES, DP(12), and KEY(5), on which SPIN ran out of memory. Note that SPIN was able to complete the analysis of these systems by finding a deadlock state before generating the entire state space. Those comparing these numbers with those in [10], which used INCA version 3.1, may notice slight differences for some of the examples. The front end of INCA has been extensively modified in version 3.2 to support the EFSAs described in Section VI-B.

E. Analysis

In this section, we consider what our data suggest about the scalability of the evaluated techniques. In general, it is difficult to characterize the scalability of an analysis technique. Complexity results indicate that there must exist problems on which a technique will not scale. In practice, we are more interested in the average case, but it is difficult to know what an average program looks like. Techniques may scale well on certain kinds of programs and poorly on others. Furthermore, most techniques are sufficiently complex that it is hard to estimate their cost on a particular problem a priori. The best we can do is examine the performance of the technique on a nontrivial collection of programs and try to determine the kinds of programs on which the technique seems to scale well.

Table III gives performance data for seventeen scalable examples. For each scalable example, tool, and resource, we have a set of four points $\{(x_i, y_i)|i = 1, 2, 3, 4\}$ where each x_i is the scale of a different size of the example and each y_i is the the amount of the resource consumed by the tool on that size. By looking down each column, we can get an informal sense of how quickly the resource requirements are growing with the scale of the problem. Such examination of the data is tedious, however, and makes comparisons difficult. We explored graphing the raw data, but the range of the measured resource units is too great to plot all the data for each example on a single graph, and using many separate graphs with different scales does not facilitate comparison. We tried selectively plotting only certain data or using mathematical transformations to make the data fit (e.g., log-linear or log-log graphs), but we found that the resulting graphs were at best difficult to interpret, and at worst extremely misleading. In the end, we decided to obtain a numerical measure of the rate of growth of each resource of each tool on each example.

We want to measure how quickly the resource requirement (y_i) is growing with the scale (x_i) of the example. At first, we considered fitting a curve of some kind to the points $\{(x_i, y_i)|i = 1, 2, 3, 4\}$ and using a parameter of the fitted curve to estimate the growth rate (e.g., we might use a linear fit and take the slope, or fit the points to 2^{ax+b} and use the parameter a). Unfortunately, the underlying forms of the actual resource functions generating the data are unknown. Furthermore, they are clearly not of a single form; some of the data appear to be linear, while some appear highly exponential. Rather than make unjustifiable assumptions about the form of the actual resource functions, we simply estimate how much faster the function appears to be growing on the right side of the interval $[x_1, x_4]$ than on the left size. Specifically, for each data set $\{(x_i, y_i)|i = 1, 2, 3, 4\}$, we calculate a growth rate for the resource function by taking the ratio of the slope of the line segment connecting $\{(x_2, y_2), (x_4, y_4)\}$ to the slope of the line segment connecting $\{(x_1, y_1), (x_3, y_3)\}$. (We considered using the ratio of the slopes of the line segments connecting $\{(x_1, y_1), (x_2, y_2)\}$ and $\{(x_3, y_3), (x_4, y_4)\}$, but in some data sets, consecutive resource measurements are equal and thus the slopes of these segments is zero.) We also determined the growth rate of the state space for each example in a similar way (i.e., by using the points $\{(x_i, |S_i|)|i = 1, 2, 3, 4\}$ where S_i is the set of global states of the i^{th} size of the example). The growth rates obtained are shown in Table IV.

This growth rate measures only the apparent curvature of the actual resource function and factors out any fixed overhead (e.g., the memory taken by the analyzer code) or constant factors (e.g., the units in which the resources are measured, the speed of the machine used to run the tool). Consequently, we may not only compare the growth rate of a single resource for different tools and examples, but we may directly compare the growth rates of different resources. On the other hand, our measure conceals the actual slope of the growth function over the interval $[x_1, x_4]$

			SPIN		SPIN+PO		SMV		INCA	
Problem(size)	Scale	States	Mem	Time	Mem	Time	Mem	Time	Mem	Time
ABP	14	113	1.29	0.67	1.71	0.84	1.70	1.03	7.13	7.23
BDS	24	36097	8.67	22.03	3.74	11.41	1.97	2.60	7.14	4.40
DARTES	90	—	1.65	1.30	2.63	4.02	—	—	15.41	26.46
FTP(1)	34	104911	49.07	133.32	33.67	202.15	3.93	18.39	14.29	206.44
FTP(2)	40	—	—	—	—	—	9.63	694.88	—	—
Q	53	123597	1.31	0.87	1.82	0.98	27.92	536.50	10.13	13.09
SPEED	16	8690	1.32	0.82	1.66	0.76	1.84	2.79	6.79	4.39

TABLE II
Raw Data for Non-Scalable Examples

(e.g., both $f(x) = x$ and $g(x) = 100x$ have a growth rate of 1.0). Although the growth rate is clearly more important than such constant factors in the limit, in practice large constant factors may have a significant impact on the scalability of a technique over the range of sizes on which its tool can be run. Thus, if the slope of a resource function over the interval $[x_1, x_4]$ is large, then this should be considered along with the function's growth rate in estimating the scalability of the technique on the example.

As discussed in Section V-C, we chose to measure growth rates rather than determine the maximum sizes each tool could handle given certain resource constraints. One concern with this approach is that the behavior of the tool on large sizes of an example may be dominated by different factors than its behavior on the small sizes over which we measure the growth rate. In other words, the measured growth rate may not be an accurate characterization of the scalability of a tool. We validate our scalability measure by comparing the growth rates calculated here with the maximum sizes determined in [10] on a couple of examples.

First we consider DP, an example on which most of the tools can be scaled to much larger sizes than those shown in Table III. For this example, SPIN, SPIN+PO, and INCA all exhausted the 64 MB limit imposed in [10], while SMV exceeded the three hour time limit instead. The memory growth rate was 5.9 for SPIN, 2.7 for SPIN+PO, 0.5 for SMV, and 1.0 for INCA. From these rates, we would expect SPIN+PO to be able to handle significantly larger sizes than SPIN, and INCA to be able to handle much larger sizes. In fact, SPIN exhausted its memory at size 14, SPIN+PO at size 22, and INCA around size 325. Note that, while it would be difficult to predict the maximum sizes without additional information, they are roughly consistent with the growth rates.

We also consider the data intensive example GASNQ. In [10], SPIN and SPIN+PO exhausted the memory limit, while SMV and INCA exhausted the time limit. The memory growth rate is 9.1 for SPIN and 7.6 for SPIN+PO. From these rates, we might expect SPIN+PO to be able to scale a bit farther than SPIN. In fact, both tools exhausted their memory at size 6. Since the memory growth rate is high, each additional customer adds a great deal of memory, and SPIN+PO could not take the extra step without exceeding the limit. The memory growth rates for SMV and INCA are 1.5 and 1.8, respectively. As expected, memory is not a problem for these tools on this example. The time growth rate is 4.0 for SMV and 2.9 for INCA. From these rates, we might expect INCA to be able to scale a bit farther than SMV. In fact, both tools exhausted the time limit at size 10. Examining the raw data reveals that the time function for INCA has a much larger slope than the time function for SMV, so again, the maximum sizes seem to be consistent with the growth rates. We therefore believe that our growth rates provide a useful characterization of the behavior of the tools on larger sizes of the examples, and thus that they are a reasonable measure of scalability.

F. Results

We now discuss the implications of our growth rates for the scalability of the techniques on different kinds of programs. For convenience in our discussion below, we will refer to rates below 2.0 as *low*, and rates above 5.0 as *high*, and intermediate rates as *moderate*. We chose these boundaries such that rates near 1.0 (linear functions) would be low, and such that most of the state growth rates would be high, although we admit that this classification is somewhat arbitrary. A picture of the overall results is given in Fig. 9, which plots the growth rates from Table IV for each tool.

We were able to correlate the performance of the different tools to various features of the example programs, principally:

Comminication structure. We may view the communication structure of a program as a graph in which each task is represented by a node, and a (possible) rendezvous between two tasks is represented by an edge between the corresponding nodes. Our examples exhibited several different communication structures. In a *linear* communication structure, the tasks can be arranged in a line or ring such that each task communicates primarily with its neighbors. In a *single star* comminication structure, most communication is between one particular task and the other tasks. A couple of our examples exhibited a combination of these two patterns (i.e., a line or ring of tasks communicating with their neighbors and one central task). Finally, in a *multiple star* communication structure, several tasks communicate with many other tasks.

Task size/structure. As discussed in Section IV-E, *data intensive* tasks require that the values of certain task variables be encoded into the state of the task's FSA for accurate modeling of the task's synchronization be-

			SPIN		SPIN+PO		SMV		INCA	
Problem(size)	Scale	States	Mem	Time	Mem	Time	Mem	Time	Mem	Time
CYCLIC(3)	10	43	1.27	0.60	1.66	0.67	1.70	0.80	6.23	3.16
CYCLIC(6)	21	639	1.37	0.81	1.69	0.71	1.84	1.00	6.96	6.18
CYCLIC(9)	32	7423	2.96	3.81	1.73	0.81	1.90	1.79	7.70	28.36
CYCLIC(12)	43	74264	23.01	51.29	1.77	0.93	1.90	2.53	8.44	66.02
DAC(6)	18	222	1.29	0.82	1.67	0.68	1.70	0.72	6.51	3.92
DAC(9)	26	1790	1.54	1.10	1.70	0.71	1.84	0.93	7.02	4.95
DAC(12)	34	14334	4.07	5.97	1.74	0.76	1.84	1.35	7.52	5.99
DAC(15)	43	114686	28.92	61.35	1.79	0.85	1.90	1.86	8.02	7.64
DP(6)	18	729	3.13	1.06	1.70	0.89	1.77	1.10	6.73	3.79
DP(8)	24	6555	3.39	2.12	1.75	1.05	1.84	1.92	7.14	4.32
DP(10)	30	48897	6.03	9.06	1.88	1.73	1.90	3.13	7.55	5.01
DP(12)	36	—	20.50	41.20	2.24	3.56	1.90	8.78	7.96	5.74
DPD(4)	15	601	1.36	0.81	1.74	0.89	1.70	0.88	6.67	4.14
DPD(5)	19	3489	1.92	1.91	1.97	1.74	1.84	1.33	6.96	4.85
DPD(6)	23	19861	5.80	12.01	2.81	5.02	1.84	1.66	7.24	6.46
DPD(7)	27	109965	29.43	94.39	6.02	18.28	1.90	3.11	7.53	7.60
DPFM(4)	7	9	1.27	0.62	1.65	0.67	1.70	0.69	6.18	3.00
DPFM(6)	10	20	1.27	0.63	1.68	0.67	1.70	0.86	7.17	6.20
DPFM(8)	14	49	1.31	0.65	1.80	0.78	1.97	2.10	11.10	45.21
DPFM(10)	18	125	1.46	0.74	2.25	1.13	2.69	9.18	27.45	156.90
DPH(4)	17	513	1.36	0.64	1.83	1.39	1.70	0.93	6.95	4.59
DPH(5)	21	3113	1.94	1.80	2.73	6.36	1.84	1.55	7.42	5.56
DPH(6)	25	16897	5.61	8.74	8.95	43.79	1.90	2.60	7.92	6.66
DPH(7)	28	79927	24.67	50.93	50.22	359.95	1.97	8.37	8.47	8.09
ELEVATOR(1)	15	158	1.30	0.68	1.71	0.73	1.84	1.16	7.59	8.61
ELEVATOR(2)	20	1062	1.51	0.78	1.98	1.35	1.97	2.79	10.60	47.10
ELEVATOR(3)	26	7121	3.18	2.42	3.90	7.64	2.42	12.16	17.78	289.02
ELEVATOR(4)	31	43440	15.18	15.90	19.28	66.46	3.60	60.64	35.48	1265.02
FURNACE(1)	13	344	1.31	0.67	1.72	0.94	1.70	0.84	6.46	3.43
FURNACE(2)	18	3778	1.92	2.42	2.40	4.34	1.84	1.29	6.93	4.27
FURNACE(3)	23	30861	8.30	29.65	8.48	43.28	1.90	2.68	7.49	5.40
FURNACE(4)	27	214757	59.93	302.94	53.42	446.60	2.03	12.03	8.10	7.03
GASNQ(2)	18	193	1.31	0.66	1.74	0.92	1.84	1.29	7.70	14.20
GASNQ(3)	23	1770	1.72	1.09	2.15	1.71	2.16	4.04	10.55	76.99
GASNQ(4)	28	14847	6.01	5.74	5.14	9.88	2.82	16.47	16.03	259.01
GASNQ(5)	33	115184	44.26	63.40	28.05	76.76	3.67	65.33	25.47	779.06
GASQ(1)	11	19	1.27	0.58	1.66	0.64	1.70	0.82	6.36	3.67
GASQ(2)	17	181	1.30	0.64	1.73	0.81	1.97	1.40	8.38	20.12
GASQ(3)	23	1705	1.68	1.15	2.10	1.39	2.36	7.60	21.55	320.46
GASQ(4)	29	15431	5.53	4.77	4.98	7.36	4.98	246.60	32.41	21580.88
HARTSTONE(25)	45	53	1.30	0.64	1.94	0.74	1.90	2.03	9.31	11.56
HARTSTONE(50)	86	103	1.39	0.67	2.24	0.91	2.49	10.00	13.57	31.17
HARTSTONE(75)	126	153	1.54	0.72	2.58	1.14	3.60	31.72	18.29	56.20
HARTSTONE(100)	166	203	1.76	0.79	2.96	1.47	5.05	77.34	23.84	89.52
KEY(2)	23	537	1.31	0.83	1.74	0.90	2.69	5.85	8.07	7.95
KEY(3)	29	4924	1.39	1.04	1.78	0.98	4.33	18.93	9.07	10.81
KEY(4)	34	44820	1.43	1.11	1.83	1.06	7.73	68.27	10.07	14.04
KEY(5)	39	—	1.49	1.19	1.87	1.15	12.91	221.55	11.09	17.47
MMGT(1)	13	73	1.28	0.63	1.67	0.65	1.70	0.88	6.98	4.94
MMGT(2)	18	817	1.37	0.76	1.77	1.05	1.90	1.55	8.09	7.64
MMGT(3)	23	7703	2.38	3.14	2.54	4.13	1.97	2.77	9.23	10.28
MMGT(4)	29	66309	12.62	62.15	9.61	41.88	2.03	8.28	10.40	13.24
OVER(2)	13	65	1.28	0.54	1.68	0.70	1.70	0.84	6.60	4.15
OVER(3)	20	519	1.38	0.73	1.73	0.81	1.84	1.37	7.29	7.46
OVER(4)	26	4175	2.35	2.12	1.89	1.30	1.84	3.19	8.02	13.30
OVER(5)	32	33460	12.71	19.11	2.67	4.51	2.16	14.24	8.81	23.84
RING(3)	15	87	1.28	0.68	1.68	0.74	1.70	0.77	6.86	4.93
RING(5)	25	1290	1.51	1.00	1.75	0.84	1.84	1.29	8.29	18.80
RING(7)	34	17000	5.63	8.09	2.04	1.68	1.90	2.91	10.40	99.39
RING(9)	44	211528	67.93	147.49	3.19	5.46	1.97	8.79	13.39	634.77
RW(6)	15	72	1.29	0.78	1.74	0.96	1.70	0.95	7.41	4.81
RW(9)	22	523	1.43	1.38	2.13	3.20	1.97	1.30	8.92	7.66
RW(12)	28	4110	2.84	9.02	5.96	32.66	2.03	2.25	10.85	11.29
RW(15)	34	29642	14.82	79.10	44.66	413.99	2.10	3.05	13.22	16.02
SENTEST(25)	39	232	1.35	0.69	1.87	0.77	1.90	1.97	9.35	11.69
SENTEST(50)	64	282	1.46	0.76	2.02	0.87	2.36	6.90	11.42	15.90
SENTEST(75)	90	332	1.57	0.83	2.19	1.01	2.88	21.60	13.82	21.57
SENTEST(100)	115	382	1.70	0.93	2.37	1.15	4.06	46.20	16.67	28.40

TABLE III

Raw Data for Scalable Examples

		SPIN		SPIN+PO		SMV		INCA	
Problem	States	Mem	Time	Mem	Time	Mem	Time	Mem	Time
CYCLIC	10.0	12.8	15.7	1.1	1.6	0.3	1.5	1.0	2.4
DAC	7.5	9.3	11.0	1.2	1.6	0.4	1.4	0.9	1.2
DP	7.3	5.9	4.9	2.7	3.0	0.5	3.4	1.0	1.2
DPD	5.5	6.2	8.3	3.8	4.0	0.4	2.3	1.0	1.2
DPFM	2.3	4.2	3.2	3.3	3.7	3.2	5.2	3.6	3.1
DPH	5.4	6.1	6.9	7.6	9.5	0.7	4.7	1.2	1.4
ELEVATOR	6.1	7.3	8.7	7.9	9.4	2.8	5.3	2.4	4.3
FURNACE	7.7	9.2	11.5	8.4	11.6	1.1	6.5	1.3	1.6
GASNQ	7.7	9.1	12.3	7.6	8.4	1.5	4.0	1.8	2.9
GASQ	9.0	10.3	7.2	7.4	8.7	4.6	36.2	1.6	68.1
HARTSTONE	1.0	1.6	1.5	1.1	1.4	1.5	2.3	1.2	1.3
KEY	10.9	0.9	0.6	1.1	1.2	1.9	3.6	1.1	1.2
MMGT	7.8	9.3	22.2	8.2	10.7	0.4	3.2	0.9	1.0
OVER	8.7	11.5	12.6	4.8	6.7	2.5	5.9	1.2	1.9
RING	12.4	15.3	19.8	4.0	4.9	0.7	3.5	1.4	6.5
RW	7.8	9.4	10.2	10.9	14.0	0.4	1.5	1.4	1.4
SENTEST	1.0	1.1	1.2	1.1	1.2	1.7	2.0	1.2	1.3

TABLE IV
Time and Memory Growth Rates for Scalable Examples

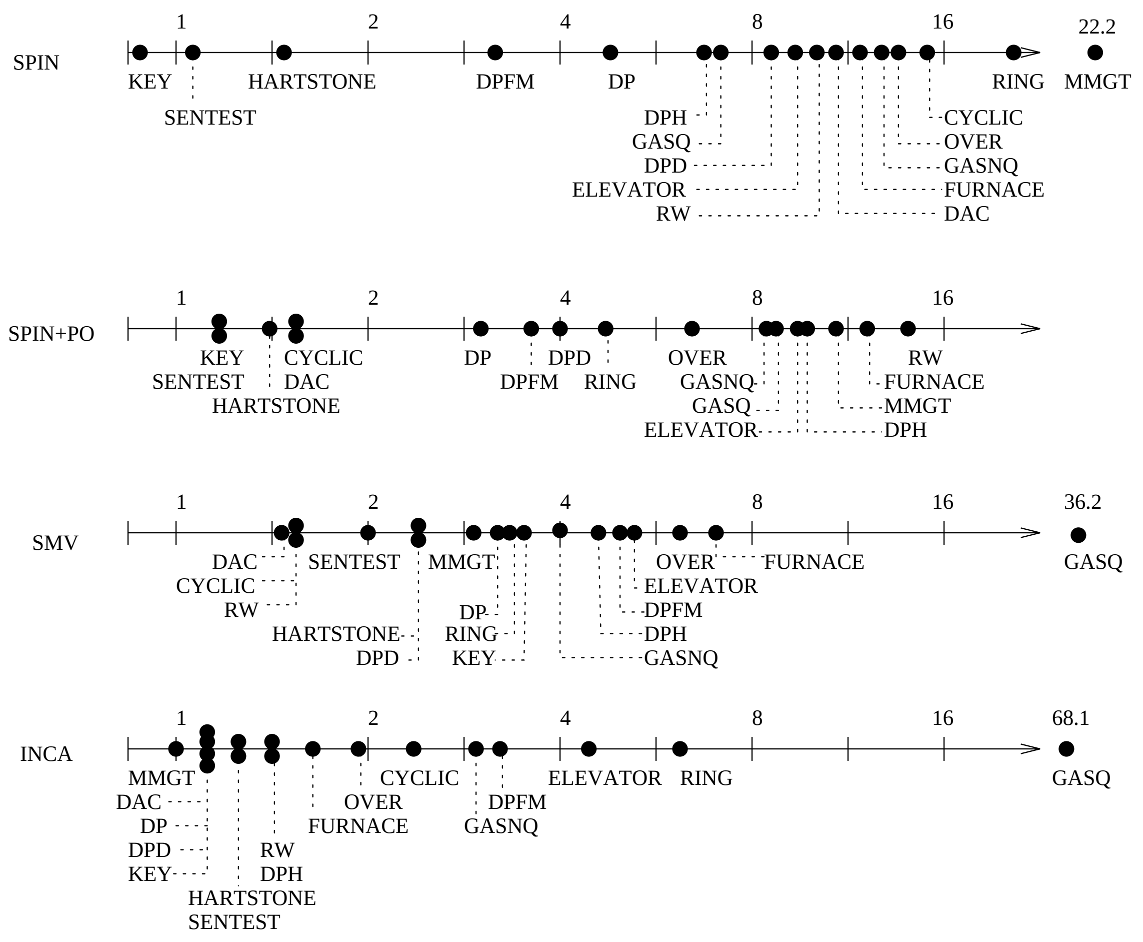

Fig. 9. Plot of Time Growth Rates (logarithmic scale)

havior. Such tasks usually produce FSAs with many states, while tasks that are not data intensive usually produce FSAs with few states. We classify the size of a task as *small* if its FSA has fewer than 10 states, *medium* if its FSA has between 10 and 99 states, and *large* if its FSA has 100 or more states. Several of the data intensive tasks in our examples exhibited a linear structure—the states can be arranged in a line such that all transitions are between adjacent states. Such *linear tasks* result from modeling the value of a single integer variable used as a counter.

In Table I, we characterize the communication structure and task size/structure of each example. For the scalable examples, the task size classification is determined by the size of the tasks in the largest size of the example analyzed (e.g., the fork manager task has only 3 states in DPFM(2), but has 1025 states in DPFM(11) and is therefore classified as large in the table).

Of the deadlock-free scalable examples, SPIN exhibited low growth rates only for HARTSTONE and SENTEST, examples on which the state spaces grow linearly with the scale. Although the synchronization structure of these examples is very simple (a master task starts/stops m worker tasks), such a structure may not be uncommon in software. SPIN also exhibited low growth rates on KEY, an example for which SPIN is able to find the deadlock state without exploring a significant fraction of the state space.

SPIN+PO exhibited significantly lower memory growth rates than SPIN for CYCLIC, DAC, DP, DPD, OVER, and RING. The common feature of these six examples is that their communication structure is linear. This structure creates the many commuting transitions that allows the partial order technique used by SPIN+PO to achieve significant reduction in the state space. Although the partial order state space reduction helped in most cases (especially with the memory growth rate, which is generally the limiting factor for the state space tools), SPIN+PO exhibited somewhat higher growth rates than SPIN for DPH, ELEVATOR, and RW. These examples contain a single star communication structure. Unlike SPIN, SPIN+PO creates a data structure for each possible pair of synchronizing transitions, and thus performs poorly on examples with this communication structure. This memory overhead is an artifact of the implementation, not of the technique itself.

SMV exhibited low time growth rates for CYCLIC, DAC, and RW, and moderate time growth rates for DP, DPD, DPH, GASNQ, HARTSTONE, KEY, MMGT, RING, and SENTEST. Like SPIN+PO, SMV performed better on examples with linear communication patterns, although it also performed reasonably well on examples with a single star communication pattern. SMV performed worse on ELEVATOR, FURNACE, GASNQ, GASQ, and DARTES, examples whose communication patterns contained multiple stars. This is not surprising since such a nonlinear structure makes it difficult to find a good variable ordering for the OBDDs. SMV also exhibited moderate/high growth rates for programs with data intensive tasks. These programs include DPFM, which has a single star communication pattern, as well as ELEVATOR, GASNQ, and GASQ. Although symbolic model checking has been used primarily for the verification of hardware, our experience indicate that it may also prove effective for verifying software. Note that SMV performed better than SPIN on most of the scalable examples, exhibiting lower growth rates for time and much lower growth rates for memory.

INCA performs worst on DPFM, ELEVATOR, GASNQ, GASQ, and RING. Of these, DPFM, ELEVATOR, GASNQ, and GASQ all contain one data intensive task whose size grows rapidly as the example is scaled up. Unlike the other tools, INCA is generally not sensitive to the communication structure of the program, but rather to the kind of tasks that comprise it. The time required to solve the ILP problems INCA generates increases very rapidly with the size of the task FSAs, unless they have a simple linear structure, like those in DPH, HARTSTONE, RW, and SENTEST. As mentioned in Section III-E, INCA uses necessary conditions and thus its analysis can be inconclusive if these conditions are not strong enough. We note, however, that these conditions were strong enough for all of the analyses in this paper.

Meaningful comparisons are more difficult for the nonscalable examples. SMV is clearly worse than the other tools on Q and DARTES (large systems with simple deadlocks), but was clearly better on FTP (a large system with no deadlocks). This reflects a general trend in our experiments: SMV tended to be slower than the other tools in finding deadlocks. SPIN(+PO) and INCA use techniques that allow them to stop as soon as a deadlock state (or integral solution, in the case of INCA) is found. SMV must construct the OBDD for the reachable states of the system in its entirety before checking for deadlock states. We do not draw any conclusions from these data, however, since our sample contained too few programs with deadlocks, and all but one of the deadlocks were trivial and would likely have been found by random simulation. In general, it is much more difficult to evaluate the performance of analysis tools when they are used to find errors rather than prove their absence since the time it takes to find an error is very dependent on factors over which the analyst has little control (e.g., the order in which a reachability analyzer explores a state space). Note that SMV performed better than SPIN in finding the obscure deadlock in the standard dining philosophers (DP).

On each deadlock-free scalable example, SPIN exhibited a memory growth rate similar to the growth rate of the state space. SPIN+PO, SMV and INCA, however, each exhibited significantly lower memory growth rates on several examples, indicating that the techniques used by those tools tend to require more time than memory as a problem is scaled. Since memory is usually the scarcer of the two resources, this often allows these tools to tackle larger problems.

VI. Alternative Models

In our experience, the most controversial aspect of our comparison method is our choice of communicating FSAs as the canonical model. Several people have questioned whether the choice of this model and the use of translators does not bias the evaluation against one or more of the techniques. To address this issue, we explore alternative canonical models. First, we consider using an informal model and manually generated native specifications for each analysis tool (thus avoiding the use of potentially biasing translators). Second, we consider using a more complex canonical model in which data values are made explicit. Finally, we consider a simultaneous model of concurrency. In the end, we found that these alternative models either did not affect or actually degraded the performance of the tools, thus increasing our confidence in the validity of our results.

A. Native Specifications

Rather than use a formal canonical model, we might have used an informal model of each example (e.g., a prose description) and used this to specify the example in each tool's specification language directly. The SMV input language is intended for describing systems much different than the programs we analyze. As a result, we would have had to convert the programs to some kind of state machine just to encode them in the language. Thus we had little choice but to use a more abstract model of the examples to generate the input for SMV. Promela, however, can easily specify communicating processes. To determine what effect using native Promela specification might have had on the experiments, we selected two examples, coded them directly in Promela, and used SPIN (a straight reachability analyzer) to compare number of states in the native model with the number of states in the translated model.

The first example we selected was a version of the standard dining philosophers problem in which deadlock is avoided by having the first philosopher pick up her left fork first while all other philosophers pick up their right fork first. This example is representative of many of the programs we analyzed in that no variables are modeled; the states of the automata representing the program encode only the control location within each task. For this example, the number of states in the native model was exactly the same as the number of states in the translated model—a reassuring result.

The second example we selected was the non-queuing version of the common gas station problem (GASNQ). This example is representative of the programs in which the state of the task automata encode the values of task variables as well as the control location within the task. The story behind our selection of this example is interesting. We sent a draft of the predecessor to this paper [10] to Gerard Holzmann, the author of SPIN, to solicit comments on our use of his tool. At his request, we also supplied the generated Promela inputs. He was concerned that using translated inputs would unduly bias the evaluation, and gave us a version of GASNQ that he had directly coded in Promela. While our translated Promela could be scaled only to 5 customers, his native Promela could be scaled up to 50 customers—a worrisome result, suggesting that our translated Promela code was much inferior to a native Promela specification.

Upon closer examination of his code, however, we noticed that his version of GASNQ was not quite the same as ours. For those familiar with the problem, the difference was that our operator task allowed many customers to prepay and kept a count of how many customers had prepaid at each pump so that the pumps could be activated so long as any waiting customers remained. This causes a state explosion in the operator task as the number of customers is scaled up. Holzmann, who worked directly from the translated Promela without knowledge of the problem or reference to our Ada-like specification, specified a system in which the operator allows only one customer to prepay at each pump. For this system, the size of the operator task does not increase with the number of customers.

We pointed out this difference and wrote our own native Promela version of GASNQ to illustrate the structure of the program we intended to model. Being the first real Promela specification we had ever written, it was quite inefficient, and could be scaled only to 4 customers, one size smaller than our translated model. Holzmann then used our specification as a guide and modified his own version to allow multiple customers to prepay. This version could be scaled to 7 customers, two sizes larger than the translated Promela. When we examined this new version, however, we noticed that it too was not quite our GASNQ. Promela allows multiple processes to read from the same channel, while Ada allows only one task to accept an entry call. Holzmann's Promela used the same channel to represent several Ada entries, and achieved some reduction in the state space as a result. In this example, at most one task would call such an entry at any given time, thus the behaviors generated were the same. Without knowing that multiple entries sharing a channel would not be called simultaneously (something that would have to be verified independently), this reduction cannot be applied since it would not correctly model the synchronization behavior of the Ada tasks; in general, each entry must be modeled with its own channel. When we modified Holzmann's specification to use one channel per entry, the resulting model could be scaled to 5 customers, the same as the translated model, and at this size had roughly three times as many states as the translated model—again, a reassuring result.

This exchange gave us confidence that our Promela translator is not introducing a significant bias into the evaluation. More importantly, it also convinced us that the use of a canonical model is essential since seemingly small differences in the specification of a problem can produce large variations in the resulting model.

B. Extended Model

The simple model of Section II is well suited for programs in which little or no data must be represented. For data intensive programs, however, that model may introduce a bias, as discussed in Section IV-E. The hiding of variable

		SPIN		SMV	
Example	Model	States	Time	Trans	Time
DPH(5)	FSA	3113	1.80	448	1.55
	EFSA	3114	2.17	1403	2.95
GASNQ(2)	FSA	193	0.66	1084	1.29
	EFSA	5704	2.43	out of mem	

TABLE V
Experiments comparing EFSA and FSA models

values within a monolithic task state may hinder OBDD-based technique on such problems. In this section, we consider an alternative model in which variable values are explicitly represented in the state of the task automata. We extend the model of Section II by adding a *memory* to each FSA. A memory is an array of cells, each of which holds an integer value from a finite subrange. Transitions may be guarded with expressions over the memory values and may transform the memory with assignments. This very general model, which we call an EFSA (extended finite-state automata), is capable of representing most uses of data we have encountered in Ada tasking programs, including rendezvous parameters, arrays and records (arbitrarily nested), and reference parameters of inlined procedures.

We modified the INCA front end to build EFSAs from Ada-like specifications. We then wrote new translators for SPIN and SMV to generate input from this new canonical form. Both SPIN and SMV support arrays of integers, thus the translation was still relatively straightforward despite the richer semantics of EFSAs. We omit the formal definition of EFSAs and the details of these translators since both are extensive and are not used in the evaluation described in Section V. Note that by encoding the memory contents into the state of an automaton, INCA translates EFSAs into FSAs for use by the inequality necessary condition technique, which is thus not affected by the different model.

We applied the new translators to two examples and compared the performance of the tools on the inputs generated from EFSAs to their performance on the inputs generated from FSAs. The examples we used for this comparison were the dining philosophers with host (DPH) with five philosophers, and the non-queuing gas station (GASNQ) with two customers. The DPH example has only one task with data: the host task has a single variable, which is used as a counter. The GASNQ example has several tasks with data: the operator task has an array of two counters, and the customer tasks each have a variable storing the pump selected. The results of this experiment are shown in Table V, the columns of which show: the example, the model, the number of reachable states reported by SPIN, the analysis time (in seconds) for SPIN, the number of OBDD nodes in the transition relation generated by SMV, and the analysis time for SMV. See Section V-D for details on how the analysis times were obtained in all of the experiments reported in this paper.

Using a richer canonical model generally degraded the performance of the tools reading the translated input. This effect was greater for SMV, and for the more data intensive example GASNQ. For SPIN, the EFSA model of DPH was almost identical to the FSA model, but the EFSA model was much worse for GASNQ. Most of the extra states in the EFSA model result from the inability to express a certain kind of atomicity in Promela. In the FSA model, the memory updates for both processes involved in a rendezvous are performed atomically with the rendezvous (since the memory is encoded in the state). The EFSA translator employed the atomic sequence construct of Promela to simulate this, but the semantics are not quite the same. The problem is that it is not possible to make the memory updates in two processes part of the same atomic action, thus the EFSA model generated by SPIN must have additional states. The performance of SPIN+PO was similarly degraded using the EFSA model.

For SMV, the EFSA models for both examples required much larger OBDDs to represent the transition relation. We believe any benefit of representing the variable values explicitly was overwhelmed by the significant increase in the number of state variables required to store the task memories. In the case of GASNQ, we believe that array indexing caused the explosion in OBDD size that exhausted the memory. The Ada version of this example is most naturally coded by using an entry accepting the pump number as a parameter, which is then used as an index into an array storing the number of customers waiting for each pump. For comparison, we coded the example without the array using a separate entry for each pump and using two integer variables to hold the number of waiting customers. SMV was able to analyze the translated input generated from these EFSAs in just about twice the time it took to analyze the input generated from the FSAs—a slowdown comparable to that obtained for DPH.

After some experience with the EFSA translators, we were disappointed to find that they produced uniformly worse performance than the FSA translators. We believe it is possible that an improved EFSA translator might produce comparable or better performance for SMV. Hu *et al* [27] have explored the verification of higher-level specifications with OBDDs and use techniques that we have not attempted, such as interleaving the bits of memory cells that are functionally related and partitioning the transition relation. We have decided not to pursue this matter further at this time for several reasons. First, most of the Ada tasking programs we have collected are not data intensive, so this issue is not critical to our evaluation. Second, even for data intensive examples, it is not clear that representing a task's state symbolically with OBDDs, as is done in the EFSA model, will produce better performance than explicitly enumerating the task's states, as in done in the FSA model. Hu and Dill [26] report that state enumeration is more efficient than their OBDD-based techniques on most of the real-life protocols they have tried. Finally, we have tried to avoid the use of special purpose techniques for specific kinds of problems in favor of techniques that are generally applicable. Fully automatic tools will have to sacrifice some efficiency for generality and ease of use.

C. Simultaneous Model

Another potential bias against the OBDD-based technique is the use of an interleaving model of concurrency rather than a simultaneous one. In an interleaving model, events are totally ordered, thus exactly one event occurs at each step. In a simultaneous model, many events may occur simultaneously. OBDD-based techniques tend to perform better on simultaneous models [32], especially when the number of asynchronous processes is large. In this section, we convert the model of Section II into a simultaneous model, describe a translation scheme for this model into the SMV language, and compare the performance of SMV on this simultaneous model to its performance on the interleaving model.

We begin by redefining the composition of a set of FSAs $M_1, \ldots, M_n$ to be an FSA $M = (S, \Sigma, \Delta, s_0, F)$ where:

- $S = S_1 \times \ldots \times S_n$
- $\Sigma = 2^{(\bigcup_i \Sigma_i)}$
- $((s_1, \ldots, s_n), A, (s'_1, \ldots, s'_n)) \in \Delta$ iff $\forall i = 1, \ldots, n$:

$$(A \cap \Sigma_i = \emptyset \wedge s_i = s'_i) \bigvee (A \cap \Sigma_i = \{a\} \wedge s_i \xrightarrow{a} s'_i)$$

- $s_0 = (s_{0,1}, \ldots, s_{0,n})$
- $F = F_1 \times \ldots \times F_n$

In this model, each M_i can take at most one transition per step, but many M_i may take transitions simultaneously.

To translate this model into the SMV language, we use a different approach than the one described in Section IV-G. There, the transition relation of M was given as a disjunction of the transitions of the M_i. Here, we give the transition relation of M as a conjunction of formulae, each of which represents the legal behavior of one M_i. Given an action A of M, each M_i either participates in this action ($A \cap \Sigma_i \neq \emptyset$) or does nothing ($A \cap \Sigma_i = \emptyset$). Thus, for each M_i, we declare a state variable σ_i of enumerated type $\Sigma_i \cup \{\texttt{skip}\}$ that stores the action M_i took on the previous step. These variables are necessary to insure that each M_i synchronizes with at most one other M_i on each step. We also declare a state variable x_i of enumerated type S_i for each M_i as before.

The **INIT** function is $\bigwedge_i (x_i = s_{i,0})$ as before. The **TRANS** function is constructed as follows. If δ is an internal action transition $s \xrightarrow{a} s'$ in M_i, then let $TRANS_\delta$ be

$$next(\sigma_i) = a \wedge x_i = s \wedge next(x_i) = s'$$

If δ is a communication action transition $s \xrightarrow{a} s'$ in M_i synchronizing with M_j on action a, then let $TRANS_\delta$ be

$$next(\sigma_i) = a \wedge x_i = s \wedge next(x_i) = s' \wedge next(\sigma_j) = a$$

Finally, let $IDLE_i$ be the function $next(\sigma_i) = \texttt{skip} \wedge x_i = next(x_i)$. The **TRANS** function is then given by

$$\bigwedge_{i=1,\ldots,n} \left(IDLE_i \vee \bigvee_{\delta \in \Delta_i} TRANS_\delta \right)$$

We ran several sizes of the standard dining philosophers problem using the interleaving and simultaneous models.

Example	Model	Trans	Time
DP(5)	Interleaving	204	0.84
	Simultaneous	591	1.64
DP(10)	Interleaving	439	3.13
	Simultaneous	1311	18.71
DP(15)	Interleaving	674	26.93
	Simultaneous	2031	362.97

TABLE VI
Experiments with Simultaneous Model

These results are shown in Table VI, the columns of which show: the example (and size), the model, the number of OBDD nodes in the transition relation generated by SMV, and the analysis time for SMV. The dining philosophers systems have a ring structure on which the simultaneous model should perform well. Unfortunately, the addition of the action state variables, which are unnecessary for the interleaving model, makes the performance worse. We note that our original translation scheme for the interleaving model included action state variables and caused similar performance problems for SMV. We conclude that an interleaving model, like the one in Section II, is better than a simultaneous model for representing Ada tasking programs.

VII. Conclusion

We have explored the methodological issues involved in empirically evaluating deadlock detection techniques for Ada tasking programs. Among these issues are the selection of examples and implementations, the specification of the examples, and the analysis of the resulting behavior of the implementations. We chose to represent each program in a canonical model and apply tools implementing the techniques to inputs generated automatically from this canonical representation. In our analysis, we calculated a numerical measure for the rate of growth of the time and memory required by the tools to complete the analysis as the example is scaled up. We believe these issues are of general significance in the empirical comparison of analysis techniques.

We have conducted an empirical evaluation of three techniques for deadlock detection in Ada tasking programs: a partial order state space reduction, symbolic model checking, and inequality necessary conditions. No technique was clearly superior to the others, but rather each excelled on certain kinds of programs. The state space reduction and symbolic model checking techniques performed best on programs with a linear communication structure. For programs with a single star communication structure, symbolic model checking generally performed better than the state space reduction technique. Inequality necessary conditions performed well on programs with small or linear tasks, regardless of the communication structure.

While our evaluation gives some indication of the kinds of programs to which the evaluated techniques might best be applied, it is only the beginning. Considerable effort on the part of many researchers will be required to fully characterize the range of applicability of each technique. Such work will require a large collection of example programs, a common model, and an agreed upon method for evaluation. This paper takes an important first step towards such a rigorous evaluation of concurrency analysis techniques.

Acknowledgements

I thank George Avrunin for suggesting the slope ratio growth measure and for extensive and valuable comments on various drafts of this paper. I also thank Gerard Holzmann, Ken McMillan, and Patrice Godefroid for assistance and guidance in using their tools and for comments on a draft of this paper. Finally, I would like to thank Ugo Buy, Matt Dwyer, and Steven Masticola for supplying many of the examples, Jalaj Swami for assistance in conducting the experiments, Will Gersch for advice on the data analysis, and Gerard Holzmann for valuable discussions on the difficulty of evaluation.

References

[1] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.

[2] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automatic derivation of time bounds in uniprocessor concurrent systems. *IEEE Trans. Softw. Eng.*, 20(9):708–719, 1994.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. 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.

[5] S. C. Cheung and J. Kramer. Enhancing compositional reachability analysis using context constraints. In *Proceedings of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 115–125, Dec. 1993.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Syst.*, 16(5):1512–1542, September 1994.

[7] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, Jan. 1993.

[8] J. C. Corbett. Verifying general safety and liveness properties with integer programming. In v. Bochmann and Probst [41], pages 357–369.

[9] J. C. Corbett. An SEDL translator. Technical Report ICS-TR-93-03, Information and Computer Science Department, University of Hawaii at Manoa, 1993.

[10] J. C. Corbett. An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–215. ACM Press, August 1994.

[11] J. C. Corbett and G. S. Avrunin. A practical method for bounding the time between events in concurrent real-time systems. In Ostrand and Weyuker [36], pages 110–116.

[12] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, January 1995.

[13] C. Courcoubetis, editor. *Computer Aided Verification, 5th International Conference Proceedings*, Elounda, Greece, 1993.

[14] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Using state space reduction methods for deadlock analysis in Ada tasking. In Ostrand and Weyuker [36], pages 51–60.

[15] M. B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachusetts at Amherst, 1995.

[16] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In D. Wile, editor, *Proceedings of the Second Symposium on Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

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

[19] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Explosion Problem.* PhD thesis, Universite de Liege, 1994.

[20] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In Courcoubetis [13], pages 438–449.

[21] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 332–242, Aalborg, Denmark, July 1991. Springer-Verlag.

[22] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In Courcoubetis [13], pages 71–84.

[23] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[24] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[25] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[26] A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In Courcoubetis [13], pages 3–14.

[27] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with BDDs. In v. Bochmann and Probst [41], pages 84–96.

[28] G. M. Karam and R. J. Buhr. Starvation and critical race analyzers for Ada. *IEEE Trans. Softw. Eng.*, 16(8):829–843, 1990.

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

[30] S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 78–87, 1990.

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

[32] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

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

[34] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Trans. Softw. Eng.*, 15(3):314–326, 1989.

[35] L. Osterweil and L. Clarke. A proposed testing and analysis research initiative. *IEEE Software*, pages 89–96, Sept. 1992.

[36] T. Ostrand and E. Weyuker, editors. *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, New York, June 1993. ACM Press.

[37] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Systems Analysis, Modeling, and Simulation*, 8:293–303, 1991.

[38] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the Fifth ACM Symposium on the Theory of Computing*, pages 1–9, 1973.

[39] W. Tichy, N. Habermann, and L. Prechelt. Summary of the Dagstuhl workshop on future directions in software engineering. *ACM Sigsoft*, Jan. 1993.

[40] S. Tu, S. M. Shatz, and T. Murata. Theory and application of Petri net reduction for Ada-tasking deadlock analysis. Technical Report 91-15, EECS Department, University of Illinois, Chicago, 1991.

[41] G. v. Bochmann and D. K. Probst, editors. *Computer Aided Verification, 4th International Workshop Proceedings*, volume 663 of *Lecture Notes in Computer Science*, Montreal, Canada, 1992. Springer-Verlag.

[42] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics

and Theoretical Computer Science, pages 25–41, Providence, RI, 1991. American Mathematical Society.

[43] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, Oct. 1991. ACM SIGSOFT, Association for Computing Machinery.

[44] W. J. Yeh and M. Young. Compositional analysis of Ada programs using process algebra. Technical report, Software Engineering Research Center, Department of Computer Science, Purdue University, July 1993.

[45] W. J. Yeh and M. Young. Redesigning tasking structures of Ada programs for analysis: A case study. *Journal of Software Testing, Verification, and Reliability*, December 1994.

[46] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, pages 200–209, 1989. Appeared as *Software Engineering Notes*, 14(8).

James C. Corbett received the B.S. degree in computer science from Rensselaer Polytechnic Institute and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst. He is currently an Assistant Professor in the Department of Information and Computer Science at the University of Hawaii at Manoa. His research is directed toward devising practical techniques and building automated tools for the analysis and verification of concurrent and real-time software.