Comparing Finite-State Verification Techniques for Concurrent Software

George S. Avrunin* James C. Corbett† Matthew B. Dwyer‡

Corina S. Păsăreanu‡ Stephen F. Siegel*

November 15, 1999

Abstract

Finite-state verification provides software developers with a powerful tool to detect errors. Many different analysis techniques have been proposed and implemented, and the limited amount of empirical data available shows that the performance of these techniques varies enormously from system to system. Before this technology can be transferred from research to practice, the community must provide guidance to developers on which methods are best for different kinds of systems. We describe a substantial case study in which several finite-state verification tools were applied to verify properties of the Chiron user interface system, a real Ada program of substantial size. Our study provides important data comparing these different analysis methods, and points out a number of difficulties in conducting fair comparisons of finite-state verification tools.

1 INTRODUCTION

Automated or partially-automated techniques for checking the correctness of concurrent programs range from formal verification that the program satisfies a complete specification to dynamic methods that examine the outcome of executions of the program or check assertions during such executions. Different techniques are, of course, useful for different purposes. For the most critical modules, for example, the extra assurance of formal verification may justify its effort and expense, but formal verification is likely to be impractical if not infeasible for large and complex programs. Dynamic techniques such as testing, on the other hand, while an essential part of software development, examine only a single execution at a time. Especially for concurrent programs, which may display very different behaviors in response to the same input data due to changes

*Laboratory for Advanced Software Engineering Research, University of Massachusetts, Amherst, MA 01003-4515, {avrunin,siegel}@cs.umass.edu

†Department of Information and Computer Science, University of Hawai‘i, Honolulu, HI 96822, corbett@hawaii.edu

‡Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506-2302, {dwyer,pcorina}@cis.ksu.edu

in the relative order of events in different components, such techniques may fail to detect serious faults.

A third class of techniques consists of finite-state verification methods such as model checking, necessary conditions analysis, or data flow analysis. Such techniques can consider all possible executions of a concurrent program, but generally cannot be used to show that a program satisfies a complete specification. Instead they check a particular property or collection of properties, such as freedom from deadlock or the mutually exclusive use of certain resources, that should hold on all executions of the program. Because these methods can be automated relatively easily, at least in comparison to the theorem-proving required for formal verification, but still consider all possible executions of the program, they have the potential to play an extremely important role in the development of high-quality concurrent software. With the spread of distributed applications running over the Internet and the widespread adoption of languages such as Java that provide support for concurrency, the need for developers of concurrent programs to use such methods will continue to increase.

A variety of finite-state verification techniques have been proposed, and several of these have been implemented as prototype tools. Experience in using these prototypes to analyze concurrent software, however, is still largely limited to simple academic programs (some of which can be scaled to large sizes) and indicates that, while all of the proposed techniques can check important properties for some programs, they all encounter problems with at least some programs that approach realistic size and complexity. Moreover, basic theoretical results tell us that virtually all the questions we want to answer about concurrent systems are at least *NP*-hard, so no single approach is likely to be practical across the broad range of concurrent programs. Indeed, preliminary experience with the prototype tools suggests that considerable skill in properly formulating models of the programs and their properties is necessary to successfully analyze real programs. Developers of concurrent systems will thus need guidance about which techniques and tools are most appropriate for a particular analysis problem and about how those techniques and tools can be used effectively, but no such guidance is now available, whether from analytic or empirical studies.

As part of a project to help provide such guidance, we have conducted a substantial case study in which several different finite-state verification tools were used to check properties of the Chiron user interface system—a large concurrent Ada program that was not written to illustrate finite-state verification techniques. In particular, we compared the performance of SPIN, SMV, INCA, and FLAVERS in checking properties of the event dispatch mechanism in Chiron as the system is scaled. This paper reports the results of this study and makes three main contributions.

First, we provide another example of the successful application of finite-state verification tools to realistic concurrent programs. We found a deadlock in the Chiron system that had been observed but not explained by the Chiron developers, and were able to verify several important requirements of the system.

Second, we have accumulated a considerable body of examples and data that are available to other researchers. The models we used and the properties we checked, expressed in the different input formalisms of the various tools, and the complete data from our analyses are available on the web and contribute to a common body of information that can be used by other researchers comparing tools and by the developers of finite-state verification tools.

Third, we have obtained clear evidence of the sensitivity of the analysis tools to details of the particular model of the system being analyzed. Models that are formally equivalent in important ways yield very different performance by the different tools, and we do not yet have a good understanding of the source of these differences in performance or the ability to predict them. This raises a number of important questions for the transfer of finite-state verification technology to industrial practice. It also points out how cautious researchers must be about using translations from one specification formalism to another. Even translations that are sound and have been reasonably well validated on a particular class of systems may introduce significant bias when applied to other kinds of systems.

The next section briefly reviews related work. Section 3 gives an overview of the Chiron user interface system, and Section 4 describes how we analyzed the system with the different analysis tools. Finally, Section 5 summarizes the data we collected, and Section 6 concludes.

2 PREVIOUS WORK

Much of the research on the application of finite-state verification tools to software systems has concentrated on the specification and design phases of software development (e.g., [1, 3, 18]). Among the advantages of using finite-state verfication at such early stages of development are the smaller size of specifications and designs, compared to the programs implementing them, and the well-known observation that errors detected at the design stage can be corrected much more cheaply than those caught after implementation. But there are also many circumstances in which it might be appropriate to apply such tools to code. For instance, it is necessary to check that the code actually implements a design correctly and, when an existing system is being modified, designs for the existing components may simply be unavailable. Some work has therefore investigated the application of finite-state verification to concurrent code. For instance, Young, et al. [23] applied a reachability-based tool to part of the Chiron user interface development system, and Masticola [20] used data flow methods to check for deadlock in a number of relatively small Ada programs. All of these investigations have focused on a single tool.

To our knowledge, the only studies comparing the performance of several different finite-state verification tools on software are those carried out by Duri, et al., by Corbett, by Chamillard, and by Dong, et al. The paper by Duri, et al. [10] compared a variety of state-space reduction methods, separately and in combination, for deadlock detection using a Petri-net based reachability analyzer and applied these techniques to a number of small Ada programs. This work involved building Petri net models of the Ada programs, and applying reduction methods to these models and their reachability graphs. A single reachability analyzer was used to search for deadlock.

Corbett [6] and Chamillard [2] compared tools using several different approaches to finite-state verification. Their studies also used small Ada programs from the concurrency literature and communication skeletons of a few real programs of relatively small size. The tools compared by Corbett and Chamillard use different formalisms for describing the program to be analyzed and for specifying the properties to check. Corbett compared three tools for checking deadlock. One of these tools uses an Ada-like language to describe the program and constructs a model of the program as a collection of communicating finite-state automata (FSAs). Corbett constructed automated translators that used this collection of FSAs to create representations of the program being analyzed in the input languages for SPIN and SMV. He validated these translations by consultation with the developers of the SPIN and SMV and by some comparisons with hand-translated versions of the original Ada programs. Chamillard's study included an additional finite-state verification tool that could work directly from Ada source code, and also considered a number of application-specific properties in addition to deadlock. He made use of Corbett's translations, and carried out a statistical analysis to check several aspects of the translation process for bias.

Neither Corbett nor Chamillard was able to draw any definitive conclusions, although both observed that there was considerable variation in both the absolute and relative performance of the tools across the range of concurrent programs considered. Corbett's work suggested that communication structure, i.e., the topology of communication among the tasks in the system, had a significant effect on SPIN and SMV, while the size of individual tasks was more important for INCA. Chamillard attempted to build predictive models of tool performance and failure using a large number of standard measures of the programs being analyzed; while he was able to construct a fairly good model predicting failure, his work suggested that the metrics he applied to the example programs did not reflect the features of those programs that most affected tool performance.

Dong, et al. [9] report on the application of five different model checkers to analyze the *i-protocol*, an optimized sliding window protocol implemented in C as part of the GNU UUCP package. The authors used their own model checker, XMC, as well as SPIN, SMV and several other tools. Abstract models were constructed by-hand in each tool's input language that represented the essential behavior of the 1500 line C program; several different people participated in this model construction effort. The results of the study showed that XMC was superior to both SPIN and SMV. Gerard Holzmann, the author of SPIN, has recently considered the details of this empirical study [17] and identified two significant methodological problems with the study relative to the use of SPIN to check the i-protocol. First, SPIN was invoked with inappropriate parameters that hurt its performance. SPIN is a complex tool and it is difficult to arrive at a setting of its parameters that optimizes its performance. In the i-protocol study, however, the investigators overrode the default settings for SPIN, causing it to allocate much more memory in performing its check than was necessary; with default settings SPIN's performance was better than that of XMC. The second methodological problem identified by Holzmann is that the XMC and SPIN models used in the study were not equivalent. It may be difficult to construct identical models in different tool input languages, due to differences in the features of those languages. In this study, however, the models were dramatically different, with the SPIN model consisting of more than twice as much information per-state than the other models. With less than 1 hour of work and minor changes to the model's description, Holzmann was able to produce a model whose per-state information was half that of the model used in the study. Using this model and the corrected settings SPIN was faster than XMC by several orders of magnitude. This analysis of the i-protocol study findings illustrates the importance of trying to eliminate possible sources of bias in the use of different analysis tools. For this study, the bias was sufficient to declare one tool superior to another when the opposite seems to be true.

3 THE CHIRON USER INTERFACE SYSTEM

Chiron [15] is a user interface development system. The data and functionality of the application for which an interface is to be constructed must be organized as abstract data types (ADTs). These ADTs are depicted by one or more *artists* that maintain mappings between ADT objects within the application and visual objects appearing on the screen.

The basic architecture of Chiron distinguishes between *servers* and *clients*. The Chiron server manages all aspects of a user interace that are not artist- or application-specific. It maintains an internal data representation that is rendered on the screen via calls to the underlying window system. The server also listens for events from the window system (e.g., button pushes) and sends them to the appropriate clients. A single server can support multiple clients.

A Chiron client comprises the application, the ADTs to be depicted, artists for those ADTs, and some runtime components that provide coordination among these components. Figure 1 gives a slightly simplified view of the architecture of a client, showing the main components and the connectors between them. A *Client Initializer* is responsible for bringing up the initial client configuration. An *Artist Manager* provides an interface through which new instances of artists can be invoked dynamically and is used by the Client Initializer to construct the initial configuration of artists. The Artist Manager also can shut down an existing artist. The *Application* makes calls to its ADTs through wrappers that provide appropriate control of concurrent access and notify other Chiron components of changes in the state of the ADTs. The *Artists* maintain the graphical depictions of the ADTs. They indicate which events they are interested in by registering or deregistering with the *Dispatcher*, which routes the events from the ADTs to the appropriate artists. Artists manipulate their graphical depictions by making calls in the Abstract Depiction Language (ADL) that are routed to the server through the *Client Protocol Manager*. Events from the server, such as button pushes, are sent to the client through the Client Protocol Manager and distributed to the appropriate artists by the *Mapper*.

The Chiron architecture is highly concurrent and even a toy Chiron interface represents about 1000 lines of Ada code. It therefore offers a serious challenge for automated static analysis methods. Because some aspects of Chiron had been previously analyzed using the CATS toolset [23], however, we had some reason to believe that the challenge would not completely overwhelm the finite-state verification tools we intended to use. Furthermore, we have good communication with the Chiron developers at the University of California at Irvine and we could call on them for assistance in understanding the code and the behavior of Chiron and in identifying significant properties to check. Chiron therefore seemed to be a good choice for a subject program for a comparative study of analysis techniques.

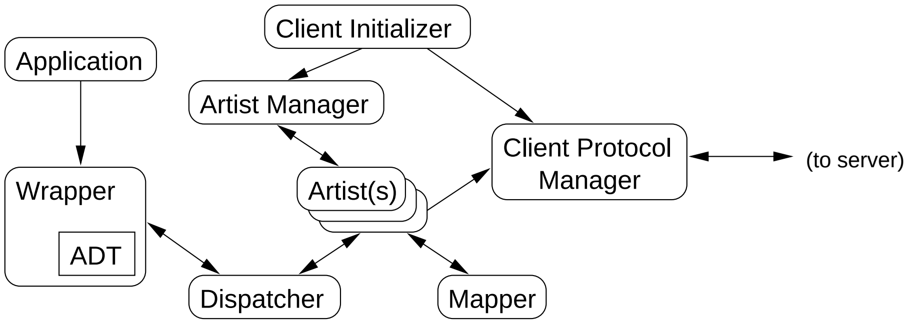

Figure 1: Chiron client architecture

4 METHODOLOGY

In this section, we describe how we applied the finite-state verification tools to a part of the Chiron user interface system.

4.1 Basic Approach

An empirical comparison of several finite-state verification tools presents some significant difficulties. First, analyzing even a single real program with a single tool is likely to be a very sizable project. Carrying out analyses of even a few real concurrent programs is an imposing task, but unless there were some way to ensure that these programs really represented the population of concurrent programs of interest, it would only provide hints about the general performance of the tools. Second, not only does the use of several tools magnify the effort needed, but it requires expertise in the use of those different tools. Finally, although such a study can avoid many of the difficulties faced by software engineering experiments with human subjects, the design of a sound study of this type is difficult and requires extremely careful attention to a number of possible sources of bias [2,6]. The present work is not intended to represent a full-scale comparison of finite-state verification tools, but rather is an exploratory study of the application of several such tools to a single program. Our goal is to learn more about the problems of comparing different tools and to gain some insights into the differences in their performance that might guide further research.

Our approach was to choose a single concurrent program of moderate but substantial size, to identify a number of properties of that program that would be of significant interest to the developers, and to use several different finite-state verification tools to check those properties. We expected to have to use various abstractions to produce models small enough to be handled by the tools, and perhaps to have to make other changes in the code.

We initially chose to use three finite-state verification tools: SPIN [16], a reachability-based model checker that explicitly enumerates the state space of the system being checked; SMV [21], which uses Ordered Binary Decision Diagrams to encode subsets of the state space; and INCA [7], which generates systems of inequalities that must be satisfied by an execution of the system that violates the property being checked and tests for consistency of those inequalities. SPIN can check Linear Temporal Logic [19] formulas. SMV checks properties expressed in Computation Tree Logic [4], a branching-time temporal logic. INCA checks queries that describe intervals in a trace of an execution [7]. These tools represent three very different algorithms for finite-state verification, and were used by Corbett and Chamillard in their earlier work.

4.2 Specification of Correctness Properties

In order to study the extent to which finite-state verification methods are effective on a real software system such as Chiron, we need a collection of correctness requirements describing the intended behavior of Chiron. Like most real software systems, Chiron does not have a complete formal description of its intended behavior. What exists is documentation in the code and information in the minds of its developers. Fortunately, the Chiron developers were willing to assist us by providing information about the intended behaviors of the system. This formed the basis of the collection of 10 properties that we checked in this study. The properties that we chose to examine focus on the event dispatching mechanism.

It is important to note that finite-state verification tools are applied to reason about partial rather than total specifications of system's correctness. In practice, one can specify relatively weak necessary conditions for correct system behavior and the analysis of such a specification may still find a defect. There are a wide range of partial specifications that one may write which describe some aspect of a systems execution, e.g., the event dispatching mechanism in Chiron. While we believe that the specifications we check are realistic and reflect the Chiron developers' understanding of the intended behavior, we do not claim that these are exactly the specifications that the developers would have written. For the purpose of our study, however, they serve as good examples of realistic system properties.

Rendering an English language requirement in the specification language of a finite-state verification tool, like SPIN, SMV or INCA, can be a significant challenge. For our study, there is an additional complication. For each requirement, we need to produce LTL and CTL formulas and an INCA query formalizing it. In order for performance comparisons of the tools to be meaningful, we must be sure that those formalizations are consistent. In other words, we must be sure that the process of translating specifications from informal to formal notations was consistent with respect to meaning; otherwise we run the risk of biasing the results by presenting one of the tools with a property that is easier to check.

To address this issue, whenever possible we used a specification patterns system [11] to help translate from English to formal specifications. The patterns system defines a collection of 40 different parameterized specification templates in 5 different specification formalisms; LTL, CTL and the INCA query language are supported. Patterns are organized by English language statements of *intent* which describe a class of permissible system behaviors. Each pattern has associated with it a specification template which is a skeletal specification. Templates are parameterized by descriptions of the specific states or events that are to be related by the specification. The entire pattern system is available at [22], including all of the specification templates, documentation for how to define parameters, and examples of its use.

4.2.1 Requirement Formalization Methodology

Formalizing an English language requirements statement proceeds in several steps.

The first step was to agree on English versions of the requirements statements that were relatively unambiguous; there were several requirements that were significantly rewritten or split into multiple requirements in this process. The final statement of the Chiron requirements we used in the study are given in Figure 2.

The second step was to identify the semantic features of the program which are mentioned in the requirement. For example, a specification might mention the occurrence of a specific rendezvous (e.g., between the dispatcher and a specific artist), or that a program variable has achieved a specified value(e.g., variable `e1_sz` is greater than zero). These semantic features were then assigned names (e.g., *dispatcherNotifyArtistEvent1*, *e1szGT0*).

The third step was to use the English requirements statement to guide the selection of the most appropriate specification pattern. The pattern can then be instantiated by substituting the names of the appropriate semantic features into the parameters in the specification templates.

To check specified properties, the names of the program features defined in step two must be bound to the states of the model in which those features hold. The INCA toolset provides a predicate definition language that allows for the description of a collection of states to be associated with a specific name. The INCA toolset then encodes the definition of that name in the model checker input file, thereby allowing properties to be written in terms of predicate names. INCA supports event and state predicates. An event refers to the occurence of a rendezvous, a procedure call, or some other designated program statement. An event predicate is true in the state following an occurrence of the event. State predicates define the points at which selected program variables hold a given value. INCA's predicate definition capabilities are detailed in [14].

In several cases, we found it desirable to refine our definition of semantic features by modifying the finite-state model to reveal relevant program features in a slightly different way. For example, if we wish to state that "after an occurrence of *event1*, there are no more occurrences of *event1*" this is an instance of an *after, absence* pattern whose instantiated specification template for this property is:

```
[](event1 -> [](!event1))
```

Unfortunately, if *event1* occurs, this formula is false. The difficulty is that the `[]` operator will begin requiring the truth of `!event1` at the state where `event1` is true—LTL is a state-based formalism which does not naturally refer to the *occurrence* of *event1*. The solution is to model the state just after *event1* becomes false. While such a refinement to the model could be automated, in our experiments with Chiron, we found it convenient to add additional control points to the program tasks at appropriate points. We then named the states corresponding to those control points (e.g., *after_event1*), so that we could use them in specifications. For the example, the resultant specification would be:

0 The system does not deadlock.

1 An artist never registers for an event if she is already registered for that event and never unregisters for an event if she is not already registered for that event.

2 If *artist1* is registered for *event1* and dispatcher receives *event1*, it will not receive another event before passing *event1* to the artist.

3 Dispatcher does not notify any artists of *event1* until it receives *event1* from ADT.

4 Having received *event1*, dispatcher never notifies artists of *event2*.

5 If no artists are registered for *event1*, dispatcher does not attempt a notification upon receiving *event1* from the ADT.

6 Dispatcher never gives *event1* to *artist1* if *artist1* is not registered for *event1*.

7 If *artist1* registers for *event1* before *artist2* does, then once dispatcher receives *event1* from the ADT, it will not notify *artist2* before notifying *artist1*.

8 No artist attempts to register for *event1* when the size of the array used to store artists registered for *event1* is equal to the number of artists.

9 The program never terminates with an artist registered.

Figure 2: English Correctness Requirements

```
[](after_event1 -> [](!event1))
```

INCA uses an event-based formalism for system desription and we encountered a related problem when we needed to represent the fact that a certain condition holds at the time *event1* occurs. In this case, we used the standard technique of identifying events that correspond to the condition becoming true and becoming false, and requiring that *event1* occur after an occurrence of the event indicating the condition becomes true and before an occurrence of the event indicating that it becomes false.

4.3 Modeling the Source Code

We chose to analyze the event handling mechanism in the Chiron architecture since this involves the interaction of several concurrent tasks—the kind of code in which subtle errors often remain undiscovered. Below, we outline the steps required to check properties of Chiron's event-handling code.

Before we can analyze the Chiron source, we need to build a finite-state model of its behavior. To do this, we use two tools: an Ada-to-SEDL translator, and the INCA front-end. The first tool can translate Ada source code into SEDL [5], a subset of Ada in a Lisp-like syntax used by INCA. Parameters to this tool control which variables are modeled (i.e., put in the SEDL) and which are abstracted away (so branches depending on such a variable are simply taken nondeterministically). The INCA front-end then takes the SEDL and constructs a set of communicating finite-state machines (CFSAs) representing the behavior of the tasks. Each task state encodes the value of the task's variables. The CFSAs communicate by taking transitions on shared symbols that represent rendezvous, and also encode any parameters passed in the rendezvous.

Analysis with the inequality-based method of INCA is then performed on the CFSAs by the INCA back-end. To analyze the CFSA model using SPIN and SMV, we used the translators developed by Corbett [6] for his empirical studies, which encode the CFSAs in the input languages of those tools. Of course, translating from one modeling language to another can introduce bias in the comparison. The translations used by INCA were evaluated and found to be fair by Corbett, who tried various alternatives and also compared the performance of the tools on native specifications to their performance on translated specifications. See [6] for details on the translation and its validation.

Figure 3 gives an overview of the various analysis tools and artifacts used in our study. Solid lines denote automated tools/translations, while dashed lines denote transformations carried out by hand.

The Ada-to-SEDL translator and INCA front-end are primarily model constructors, not model extractors. Except for the ability of the Ada-to-SEDL translator to selectively include certain variables, these tools try to model everything they are fed. Since we want to focus on a specific aspect of the program's behavior, we had to remove the unimportant parts of the source code by hand. We were also forced to make some changes to the code in the dispatcher task, as explained below.

The dispatcher task is the center of the event delivery mechanism, thus we included the code for this task. The original Ada code used a dynamically allocated linked list to record the list of artists registered for each event. Since the number of artists is fixed at compile time, it is not clear why the developer chose to use a linked list rather than an array, but in any case, our model constructor (like most static analysis tools) cannot handle dynamic lists and was thus unable to represent the list. Unfortunately, without modeling the information in this list (and its effect on the control flow within the dispatcher task), few useful properties can be proved. We therefore changed the implementation of the dispatcher to use (fixed-sized) arrays to record this information; our model construction tools are able to handle this construct.

Other tasks relevant to the event handling mechanism include the artists, who register for and accept notification of events, and the ADT wrapper task, which starts the artists and generates the events (or more accurately, passes them on from the application). Since our focus is the event handling, we removed most of these tasks, leaving only stubs to model the relevant interaction of these tasks with the dispatcher. In particular, the artist stub tasks are started by the ADT wrapper, register for one or more events, and then wait to be notified that these events occurred (by the dispatcher). The specific data maintained by the artists and the artist's interaction with the application is abstracted away. The ADT wrapper task starts the artists and then delivers a sequence of (nondeterministically chosen) events to the dispatcher.

To obtain a tractable model, we arbitrarily limited the number of artists and events. Thus it must be emphasized that our analyses are not conservative, in the sense of covering all Chiron interfaces. Nevertheless, it is well known [8, 18] that most design errors are present in small versions of a system, so analyzing restricted models is an excellent way to find errors in the full (scaled) system.

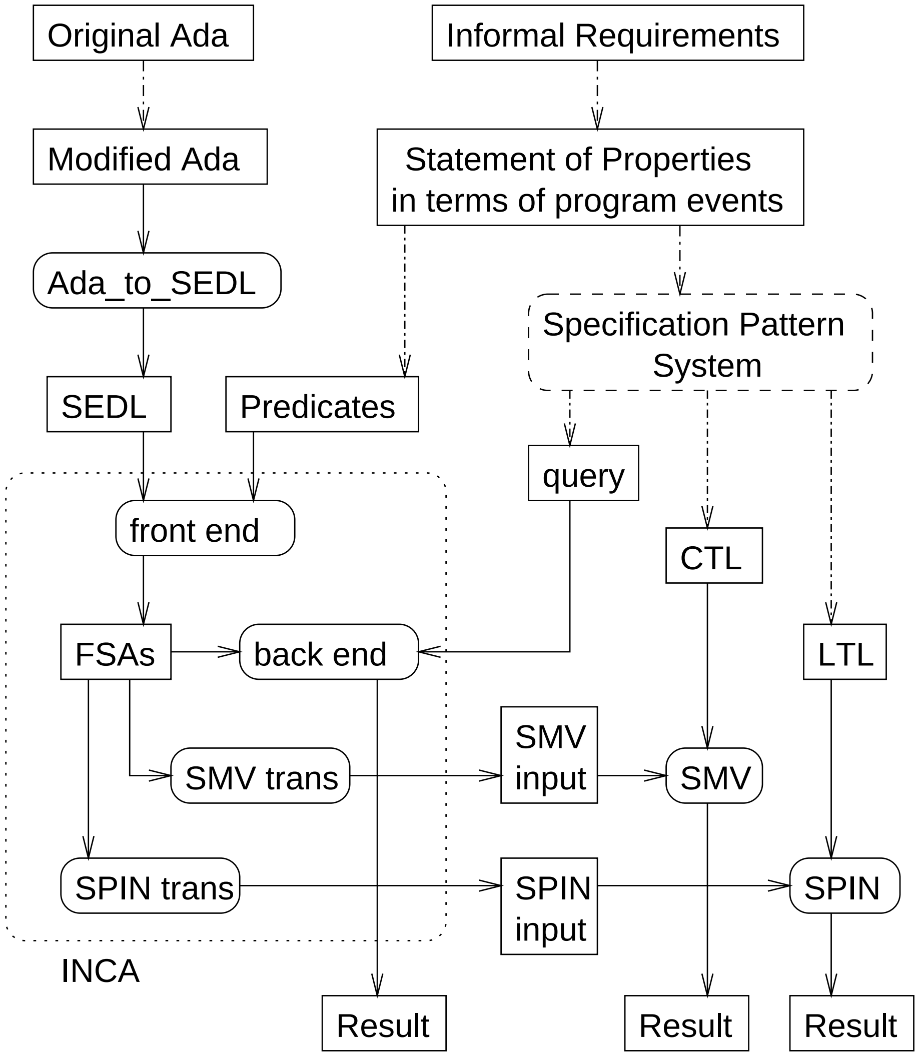

Figure 3: Analysis Tools and Artifacts

5 RESULTS

In this study, we used SPIN version 3.2.4, SMV version 2.5.3, and INCA version 3.4. All runs were made on a Sun Enterprise 3500 with two processors and 2 GB of memory, running Solaris 2.6. SPIN and SMV were compiled with the Sun C compiler, using the options provided by the tool developers. For INCA, we used Harlequin Lispworks 4.1.0.

For our initial tests, we used each tool to verify, or find a counterexample to, each of the 10 properties listed in Figure 2 for a system with 2 artists and 2 events. We steadily incremented the number of events until memory capacity was exceeded for one or more of the properties.

The results of these initial experiments were simultaneously encouraging and discouraging. On the positive side, the 3 tools all detected a deadlock that can arise when the dispatcher is trying to inform an artist of an event and the artist is trying to unregister for an event. (The Chiron developers had seen this deadlock when the system was shutting down, but had not determined precisely how it arose.) The 3 tools were also able to verify the remaining properties in Figure 2 for small systems. But we were not able to analyze systems with more than 6 events, indicating that our methods were far from scaling to realistic programs.

Closer examination of the data indicated two possible sources of difficulty. The first of these involves the construction of FSAs by INCA. Essentially, INCA builds the FSA for a particular task by constructing what amounts to a control flow graph for that task, expanding this graph by creating states at each node to represent all possible combinations of values of the task's variables, and then using a form of constant propagation to eliminate unreachable states. Although a number of optimizations are applied to reduce the size of the intermediate graphs, this procedure can blow up on tasks with a large amount of local data. In the case of Chiron, the Dispatcher task maintains an array for each event that records the artists registered for that event and the order in which they have registered, leading to a large number of possible values for the cells of the arrays. With more than 6 events, the current implementation of INCA could not build and prune the intermediate graph used to construct the Dispatcher FSA. This suggests that INCA is simply not suited for analyzing the Chiron system, and that the limitations of SPIN and SMV arise from the limitations of INCA as a front-end. This is consistent with Corbett's earlier work, which had suggested that INCA did not perform well on systems in which a single task had a particularly large FSA.

The second possible problem, however, involves the actual translated input for SPIN and SMV created by INCA. For SPIN, the translator represents each state of an FSA as a location in the Promela code and describes the possible transitions in terms of these locations. Thus, the range of possible values of variables at a single location in the Ada code is explicitly represented by separate locations in the Promela input for SPIN, rather than implicitly represented through different values of Promela variables. Although this approach produces a model with a small number of states to explore, it leads to large input files that take a long time to process. (Note that analysis with SPIN involves constructing a C program from the Promela code, and compiling that program.) Similarly, because the SMV process input formalism cannot efficiently represent the rendezvous construct of Ada, the translation makes use of a second mechanism in SMV for modeling the program to be analyzed, the TRANS input format. This requires explicitly describing the program as a transition system, and again results in large input files. For instance, for the 2-artist, 5-event Chiron system and property 5, the input file for SPIN is 584 KB and the input file for SMV is almost 2 MB. Although the translations had been validated for relatively small programs, we were concerned that the translations themselves made the analysis difficult for SPIN and SMV.

To explore the first issue, we decomposed the Dispatcher task into a subsystem with a separate task that maintains the array for each event, together with a single interface task that receives the requests for registration and unregistration and the notification of events, and passes them to the appropriate task responsible for a particular event. We constructed this subsystem so that no additional parallelism is introduced—if internal communications of the Dispatcher subsystem are hidden, the new system is observationally equivalent to the original one—but each task now maintains only a single array and the explosion in the size of the intermediate graphs does not occur. We refer to this version as the decomposed dispatcher version, in contrast to the original version.

To explore the second issue, we translated the Ada code (modified and abstracted as described in the previous section) into Promela by hand, going through several versions as we refined the Promela code to improve the performance of SPIN. We translated both the original and decomposed systems, producing what we call "native SPIN" versions, as opposed to the translated "INCA-SPIN" versions. (We also attempted to produce a hand translation into the SMV process formalism but several problems in representing Ada's concurrency constructs, including the lack of support for rendezvous-style communication and the difficulty in modeling the guarded select statement, prevented us from producing a reasonable translation. It is certainly possible to introduce extra state variables to mimic rendezvous, but earlier work had shown us that this led to worse performance than the translated systems.)

We then collected data on the verification of the properties in Figure 2 for both the original and decomposed versions of Chiron, using INCA, SMV (with translated input), INCA-SPIN, and native SPIN.

At about this time, a new implementation of the FLAVERS data flow analysis tool [12, 13] became available, and we also analyzed both the original and decomposed versions with FLAVERS. FLAVERS constructs annotated control flow graphs for the tasks of the program from the Ada code, and adds edges connecting nodes in different tasks based on the possible interleavings of events from the tasks to build a Trace Flow Graph. Paths through this graph then represent possible sequences of events that could be seen on executions of the program. FLAVERS uses a regular expression-based formalism to represent the property to be checked, and propagates states of an FSA representing the property through the Trace Flow Graph to check whether the property holds. FLAVERS provides a mechanism for adding information, in the form of additional FSAs called *feasibility constraints*, to eliminate infeasible paths and increase the precision of the analysis. The current implementation of FLAVERS has a number of limitations that affected the analysis of Chiron, many of which required considerable further modification of the Ada code. First, FLAVERS cannot check for deadlock, so we did not check property 0 with FLAVERS. Second, because a single call or accept statement in Ada may be executed with different values of the parameters, it is not possible to insert events that identify the parameter values passed in a rendezvous. It

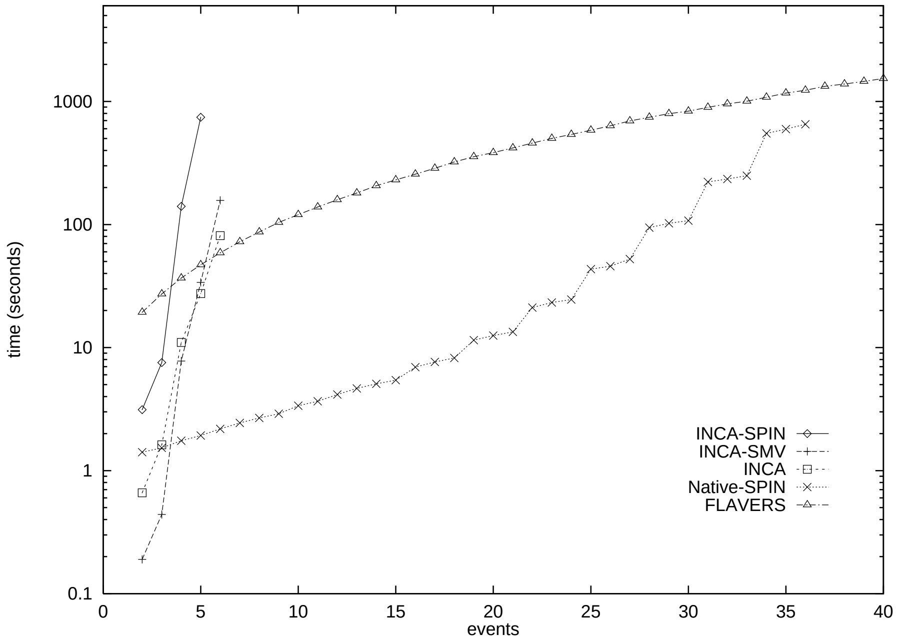

Figure 4: Tool Performance for Property 5 with Original Dispatcher

was therefore necessary to modify the Ada code to create separate calls and entries for each possible set of parameter values that could be passed in a given rendezvous. Finally, because FLAVERS cannot represent the use of a variable as an index into an array or directly handle situations where the value of one variable is set to the current value of another variable, it was necessary to unroll all the loops in which arrays are searched and updated, and to add explicit case statements to handle the updating of the arrays. All of these modifications could presumably be carried out automatically using appropriate compiler and partial evaluation technology, but we made the changes by hand.

INCA, SPIN, and SMV were able to detect the deadlock and verify all the other properties in Figure 2 for the original version as well as the decomposed version. FLAVERS was able to verify the non-deadlock properties in both versions. Figure 4 shows the times for the various tools in verifying a typical property (number 5) on the original dispatcher version, while Figure 5 shows the times for verifying the same property on the decomposed version. Note that the time axis is logarithmic in both figures. (All of the data from this study are available at `http://laser.cs.umass.edu/verification-examples/chiron/`.)

The input for INCA is generated from the Ada source by our Ada-to-SEDL translator. The data shown for INCA include the time needed for generating a system of inequalities from the SEDL input and the query specifying the property to be checked, and the time to solve that system of inequalities. The data labeled INCA-SPIN in the

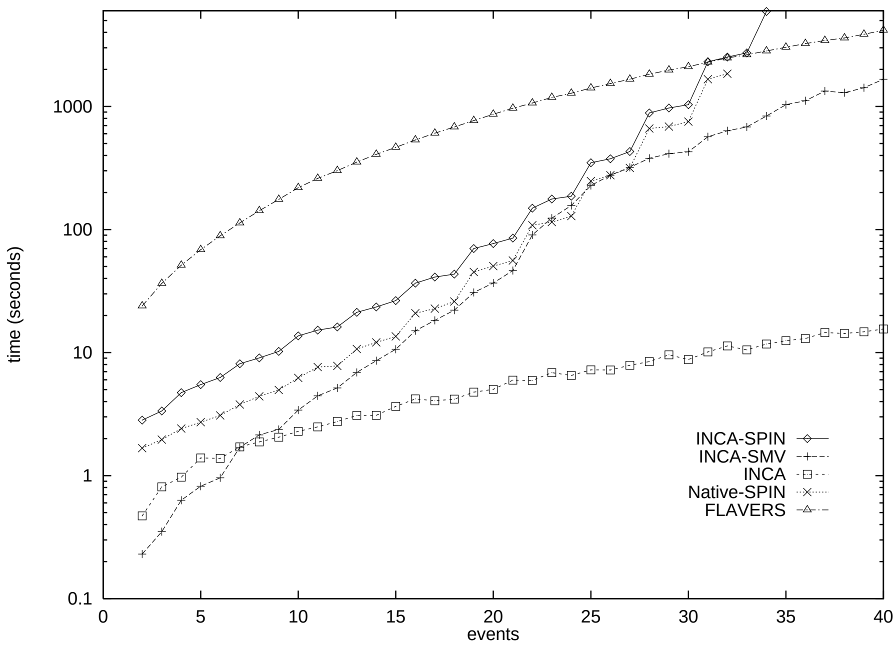

Figure 5: Tool Performance for Property 5 with Decomposed Dispatcher

figures are for analysis using SPIN of Promela code generated by our INCA-based translator. The times shown include the time to generate a Büchi automaton from the LTL formula specifying the property to be checked, the time to generate the C source code for the analzyer for the particular problem, the time to compile that code, and the time to run the analyzer. The data labeled native-SPIN are for analysis using SPIN of Promela code translated by hand from the Ada source code, and the times include the same steps as for INCA-SPIN. The data labeled INCA-SMV are for analysis using SMV of the TRANS relation generated by our INCA-based translator to check the CTL formula representing the property to be checked. The data labeled FLAVERS include the time to build annotated control flow graphs from the Ada source code, the time to build the Trace Flow Graph from these control flow graphs, the time to build the property automata from the Quantified Regular Expression representing the property to be checked, the time to build the feasibility constraints, and the time for state propagation over the Trace Flow Graph to check the property.

5.1 Discussion

In this section, we note some of the most salient features of the data.

Perhaps the most striking aspect of the data is the difference between the performance of INCA and the tools using the INCA-based translators (INCA-SPIN and INCA-SMV) on the original version and on the decomposed version. These tools were unable to handle more than 6 events for the original version, but can handle more than 30 (more than 60 in the case of INCA) for the decomposed version. For the original systems, the performance of native-SPIN (SPIN applied to the hand-translated Promela code) is significantly better than that of INCA, FLAVERS, or the two tools using input produced by the INCA-based translators (although it appears that FLAVERS may do better on the largest systems.) On the decomposed systems, on the other hand, the times for native-SPIN, INCA-SPIN, and INCA-SMV are relatively close, while INCA is very much faster and can do larger systems. (The jumps between sets of three consecutive points seen in some of the data are an artifact of the way we chose which events an artist would register for as the number of events increased.)

It is clear that INCA breaks down on systems where a single task has a large number of states, and that the translations based on INCA's FSAs are unfair to SPIN and SMV for such systems. On the other hand, the translations seem to be reasonable when no task is too large. On the decomposed system, for instance, the native-SPIN and INCA-SPIN times seem to differ only by a small factor and INCA-SPIN can actually handle slightly larger systems. Furthermore, the number of states of the translated Promela models is slightly smaller than the native Promela models we created.

We note that the SPIN times include the time to compile the pan.c program generated from the Promela code. These compilation times can be as much as one-third of the total time for the smaller decomposed examples, and an even bigger fraction on the smaller examples of the original version.

The FLAVERS times include the time to build the annotated CFGs, which can be as much as 3/4 of the total time. The FLAVERS tool is designed to work directly from Ada source code. The times shown for the other tools do not include the time necessary to translate from Ada into their native input languages, although they do include the time to create internal representations from those native input languages. The times shown for FLAVERS are therefore perhaps not strictly comparable to those shown for the other tools, although simply subtracting the time to build the CFGs would not give comparable times, either. Furthermore, the FLAVERS times shown represent the analysis time using a minimal set of feasibility constraints of two standard types that was needed to verify the property. This set was determined by exhaustively testing sets of possible feasibility constraints, and may or may not represent the best possible FLAVERS time—it is possible that using additional constraints would result in a lower time by eliminating more paths through the Trace Flow Graph. The problem of quickly determining a useful set of feasibility constraints for FLAVERS is one that is not well-understood as yet, but is important for the use of FLAVERS in finite-state verification.

We were surprised to observe that, with the exception of FLAVERS and checking for deadlock with SPIN, the performance of the tools showed very little variation across the properties. (Since SPIN includes a built-in check for deadlock, its improved performance when checking for deadlock was not unexpected.) For FLAVERS, there was considerable variation between properties. In some cases (e.g., property 9) this appears to be due to the size of the regular expression used to state the property, but in others (e.g., property 4), we do not understand the precise source of the variation. It should be noted however, that SPIN, INCA, and SMV were applied to the same program model for each property, while the FLAVERS feasibility constraints were chosen separately for each property. Tuning the model to the particular property, for instance by using program slicing to remove variables not important for that property, might lead to greater variation across properties for the other tools as well.

6 CONCLUSION

In this paper, we have reported on a case study comparing several finite-state verification techniques for concurrent systems. We applied these techniques to check a number of properties of a Chiron user interface system and successfully used these tools to detect a deadlock that had been observed but not explained by the Chiron developers. We have also accumulated a large body of examples and data that are available on the web and contribute to a common body of information that can be used by other researchers comparing tools and by developers of finite-state verification tools. In the course of our study, we have obtained clear evidence of the surprising sensitivity of finite-state verification tools to the details of the particular model of the system being analyzed. This raises a number of important questions for the transfer of finite-state verification technology to industrial practice, and points out how cautious researchers must be about using “equivalent” models or translating from one specification formalism to another.

ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation under Grants CCR-9407182, CCR-9708184, and CCR-9703094, by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract F30602-94-C-0137, and by the National Aeronautics and Space Administration and Defense Advanced Research Projects Agency under NASA grant NAG-2-1209. We are grateful to Rajesh Prabhu and Gleb Naumovich for their assistance in formulating the properties to be checked, to Kari Nies for helping us understand and run Chiron, to Jamieson Cobleigh for his assistance in using FLAVERS, and to Lori Clarke and Lee Osterweil for many helpful discussions about this project.

References

[1] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng.*, 19(1):24–40, Jan. 1993.

[2] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. TR 96-84, Department of Computer Science, University of Massachusetts, 1996. Revised May 1997.

[3] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, 24(7):498–520, July 1998.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, April 1986.

[5] J. C. Corbett. The S-expression design language (SEDL). ICS-TR-93-02, Information and Computer Science Department, University of Hawaii at Manoa, 1993.

[6] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.

[7] 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.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design*, pages 522–525, Oct. 1992.

[9] Y. Dong, X. Du, Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, O. Sokolosky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, *LNCS* volume 1579, pages 111–222, 1999.

[10] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Trans. Softw. Eng. Meth.*, 3(4):340–380, Oct. 1994.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 411–420, Los Angeles, may 1999.

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

[13] M. B. Dwyer and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. TR UM-CS-1999-052, Department of Computer Science, University of Massachusetts, Amherst, MA, Aug. 1999.

[14] M. B. Dwyer, J. C. Corbett, and C. S. Păsăreanu. Translating Ada programs for model checking: A tutorial. KSU CIS TR 98-12, Department of Computing and Information Sciences, Kansas State University, 1998.

[15] K. Forester, C. MacFarlane, M. Cameron, and G. Bolcer. Chiron-1 user manual. Arcadia Document UCI-93-07, University of California, Irvine, Sept. 1993.

[16] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[17] G. J. Holzmann. The engineering of a model checker: The GNU i-protocol case study revisited. In *Theoretical and Practical Aspects of SPIN Model Checking*, *LNCS* volume 1680, pages 232–244. Springer Verlag, Sept. 1999.

[18] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, July 1996.

[19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[20] S. Masticola. Static detection of deadlocks in polynomial time. LCSR-TR-208, Department of Computer Science, Rutgers University, May 1993.

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

[22] Specification patterns web site. `http://www.cis.ksu.edu/santos/spec-patterns/`.

[23] M. Young, R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Trans. Softw. Eng. Meth.*, 4(1):65–106, Jan. 1995.