Flow Analysis for Verifying Properties of Concurrent Software Systems

MATTHEW B. DWYER
University of Nebraska Lincoln
LORI A. CLARKE and JAMIESON M. COBLEIGH
University of Massachusetts Amherst
and
GLEB NAUMOVICH
Polytechnic University

This article describes FLAVERS, a finite-state verification approach that analyzes whether concurrent systems satisfy user-defined, behavioral properties. FLAVERS automatically creates a compact, event-based model of the system that supports efficient dataflow analysis. FLAVERS achieves this efficiency at the cost of precision. Analysts, however, can improve the precision of analysis results by selectively and judiciously incorporating additional semantic information into an analysis.

We report on an empirical study of the performance of the FLAVERS/Ada toolset applied to a collection of multitasking Ada systems. This study indicates that sufficient precision for proving system properties can usually be achieved and that the cost for such analysis typically grows as a low-order polynomial in the size of the system.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

This work was partially supported by the National Science Foundation under Award Nos. CCR-9703094, CCR-9708184, CCR-0093174, CCR-0205575, and CCR-0306607; an NSF/DOD Capacity Building Grant, Contract F496200110243; the US Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH0100CR231; the US Army Research Office under Award Nos. DAAD190110564 and DAAD190310133; and IBM Faculty Partnership Awards. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency, the US Army Research Office, the US Department of Defense, or IBM.

Authors' addresses: M. B. Dwyer, Department of Computer Science and Engineering, 256 Avery Hall, University of Nebraska, Lincoln, NE 68588-0115; email: dwyer@cse.unl.edu; L. A. Clarke and J. M. Cobleigh, Computer Science Department, University of Massachusetts, 140 Governor's Drive, Amherst, MA 01003-9264; email: {clarke, jcobleig}@cs.umass.edu; G. Naumovich, Polytechnic University, 5 MetroTech Center, Brooklyn, NY 11201-3840; email: gleb@poly.edu.

General Terms: Verification

Additional Key Words and Phrases: Dataflow analysis, finite-state verification, model checking

1. INTRODUCTION

Concurrency is commonly employed to help software systems meet demanding performance or responsiveness requirements. Unfortunately, concurrency often exacerbates the already difficult problem of validating software behavior. In this article, we describe FLAVERS (FLow Analysis for VERification of Systems), an approach that helps address this problem by verifying whether user-specified properties are valid for all possible executions of a system.

The behavior of a concurrent system may depend on the order in which instructions from different tasks are scheduled relative to one another. Two runs of a system on identical inputs may produce different results if the order of instruction execution differs for these two runs. Consequently, errors may only manifest themselves under a few possible task schedules. To help detect serious problems in concurrent systems, therefore, analysts need tools that can exhaustively consider all possible task schedules and analyze the behavior of the system on those schedules.

To address this concern, researchers have been developing a number of *finite-state verification* approaches. These approaches typically compare a finite model of a system to a property specification. Finite-state verification approaches are not as general as theorem-proving-based verification approaches [Hoare 1969; Dijkstra 1976] in terms of the kinds of properties that can be proved but, unlike automated theorem provers, they are guaranteed to terminate and require much less mathematical sophistication to use.

FLAVERS is a finite-state verification approach that uses dataflow analysis techniques to verify event-based, behavioral properties of concurrent systems. With FLAVERS, analysts define a set of events that they wish to reason about and specify properties of concurrent systems as patterns of those events. FLAVERS then automatically creates a concise graph model of the system that explicitly represents intertask communication, synchronization, and event orderings, without enumerating the state space. At the core of FLAVERS is a polynomial-time, conservative flow-analysis algorithm, which determines whether the system satisfies a given property and, if not, provides example traces through the model that violate that property.

As we demonstrate in this article, it is possible to perform reasonably efficient analyses with FLAVERS. The utility of such analyses should be judged not only by efficiency, but also by the precision of the results. To overcome the traditional imprecision of dataflow analysis, we have developed an approach for incrementally improving precision. With this approach, analysts can examine previous analysis results, decide what information is needed to improve precision, and then introduce constraints that represent this information. Moreover, automated support is provided for creating some common types of constraints.

FLAVERS is applicable to a wide range of concurrency models, as well as sequential systems. In addition to analyzing software implementations, it can

be used to analyze high-level architectural descriptions or low-level designs, as long as event-flow information is available. We have conducted a number of case studies [Chamillard et al. 1996; Naumovich et al. 1996; Dwyer 1997; Naumovich et al. 1997] and, in all cases, FLAVERS discovered the event-based errors that were known to exist. In addition, in some cases, FLAVERS discovered previously unknown errors. Industrial users have also reported finding errors using FLAVERS. Many of those errors were simple and were discovered in the process of formalizing properties. However, the more complicated errors were found during verification [Bouwens et al. 1996; Science Applications International Corporation 1997].

In this article, we discuss analysis of systems with explicit tasking and rendezvous communication and illustrate our approach using concurrent Ada programs. A toolset, called FLAVERS/Ada, supports most of the constructs in Ada, but as with most static concurrency analysis approaches, does not handle dynamic constructs, such as dynamic memory or task allocation. As is typical for static approaches, the analyst must select a particular system configuration when dealing with dynamic constructs. This and other limitations of FLAVERS are discussed in the conclusion.

Other finite-state verification approaches have primarily focused on hardware descriptions (e.g., McMillan [1993]) or system modeling notations (e.g., Holzmann [1997]). Recently there has been considerable interest in applying finite-state verification to software [Ball and Rajamani 2001; Holzmann 2000; Visser et al. 2000]. Although promising, these reachability-based approaches are severely limited in the size of the systems that they can evaluate. We believe that FLAVERS makes a number of contributions to flow-analysis and concurrency-analysis research that address some of the limitations of these approaches, including:

—a polynomial-time, dataflow analysis algorithm for proving user-specified properties of a system,
—a concise, event-based model of concurrent systems that is automatically derived from a system representation,
—an incremental process though which analyses of increasing precision can be constructed, and
—a demonstration, through empirical evaluation, that flow analysis can be effectively used to verify concurrent software systems.

Section 2 provides a brief, high-level overview of FLAVERS. Section 3 describes the property notations used by FLAVERS, and Section 4 provides a detailed description of the system model. The state-propagation algorithm and its attributes are given in Section 5, followed by a section that describes how constraints are represented and how the flow-analysis algorithm is extended to handle constraints. After having described the FLAVERS analysis approach, Section 7 presents some experimental results gathered from applying the FLAVERS/Ada toolset to the analysis of a collection of concurrent Ada applications. Section 8 discusses related work, and Section 9 summarizes the

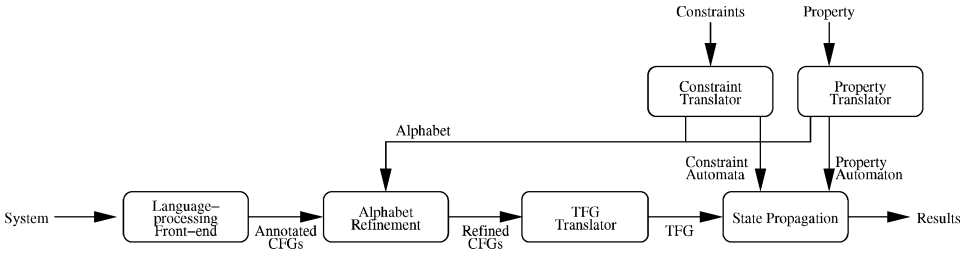

Fig. 1. Architecture of FLAVERS.

contributions of this approach and discusses limitations and directions for future research.

2. OVERVIEW

With finite-state verification approaches, the analyst usually has to experiment with several alternatives before finding an appropriate model of the system that will lead to an effective analysis result. FLAVERS explicitly supports an incremental approach to developing the system model. Using this approach, the analyst provides the software system to be analyzed, the property to be verified, and any initial constraints, such as environmental restrictions, that the analyst knows should be respected. As shown in Figure 1, FLAVERS automatically translates the system, property, and constraints into appropriate internal representations, and then employs *state propagation*, a dataflow analysis technique, to determine if the system, as restricted by the constraints, is consistent with the property.

The analysis may return *conclusive* results, indicating that the property is consistent with the system. Alternatively, the analysis may return *inconclusive* results; this can happen because a fault in the system (or in the property) causes some executable behavior to be inconsistent with the property or because the analysis is too imprecise. FLAVERS generates counterexamples when the analysis returns an inconclusive result. These counterexamples are examined by the analyst and, if a fault is apparent, it can be fixed and the analysis can be rerun. In cases where the fault is not obvious or where imprecision in the model is the cause of the inconclusive analysis result, incorporating further constraints can be helpful. This last step of incorporating additional constraints can be performed multiple times, thereby providing an incremental approach for increasing precision.

2.1 Specifying Properties

FLAVERS supports the verification of event-based properties. When specifying properties, analysts may wish to reason about the execution of a system from a variety of perspectives. For example, if analysts are interested in determining whether some statement in a system uses a variable whose value is uninitialized, then the events of interest are the definitions and uses of that variable. Or, if analysts are interested in determining that whenever a read or write operation on a file is called, the file has been opened and will subsequently be

closed, then the events of interest correspond to file operations. To address these kinds of variations in focus, analysts define a set of events of interest, where an *event* is an observable, indivisible system action. Indivisibility of events is defined with respect to other events. Thus, computation of a complex expression or even a subprogram could be defined as a single event, if it is not possible for intervening events to occur. A set of events used in a FLAVERS analysis is referred to as an *alphabet* and denoted by Σ.

FLAVERS allows analysts to specify properties that describe either desirable or undesirable event sequences, since it is sometimes more convenient to describe a violation of a requirement than to describe the requirement itself. With FLAVERS, analysts can represent a property as a *quantified regular expression* (QRE) [Olender and Osterweil 1990] or as a *finite-state automaton* (FSA). From such a description the toolset constructs a *property automaton* that compactly represents the set of event sequences.

2.2 Modeling System Executions

To reason about the set of possible system executions, we need a semantically well-founded model of those executions. This model need not represent all the details of the executions, but it must contain sufficient information to support the desired conservative analyses. FLAVERS builds a *trace-flow graph* (TFG) model that compactly represents all sequences of events that correspond to potential system executions for the events of interest. As shown in Figure 1, the events of interest are obtained from both the property and constraints and determine the alphabet for the analysis problem. Each task in the system to be analyzed is first translated into an annotated *control-flow graph* (CFG), and then refined based on this alphabet to form a *refined CFG* (RCFG). After *alphabet refinement*, the RCFGs are combined to form the TFG. The TFG is optimized so that each node in the TFG represents important flow of control information, the execution of some event of interest, or both. The TFG conservatively approximates the executable sequences of events, but may also include some *infeasible sequences*, sequences that occur in the model but do not correspond to any actual executions of the system.

2.3 State-Propagation Analysis

FLAVERS casts the question of whether system behavior is consistent with a property as a state-propagation, flow-analysis problem [Olender and Osterweil 1992]. We have adapted this approach to apply to concurrent systems and have extended it to incorporate constraints. Conceptually, a sequence of events associated with a path in the TFG would determine a sequence of transitions in the property automaton. The state-propagation algorithm determines all the property states that could be associated with a node in the TFG for all possible paths to that node. This dataflow analysis algorithm avoids enumerating all potential sequences of system events by collapsing them into equivalence classes. The result is an analysis that computes a conservative answer to the analysis question and whose complexity is polynomial in the number of TFG nodes. Conclusive analysis results provide the same assurance as any other

formal verification method. Inconclusive analysis results provide information about the source of the faults or about imprecision in the system model. It is important to note that FLAVERS may produce conclusive analysis results even when some infeasible sequences remain in the model. Constraints are only necessary when there are infeasible sequences that cause the property to be violated; elimination of these sequences leads to conclusive results if the actual system does not violate the property.

3. SPECIFYING SYSTEM BEHAVIOR

Finite-state verification compares a model of a system's executable behavior to a specification of its intended behavior. Although it may be possible to specify the intended behavior of a system completely using very expressive formalisms, such as Z [Spivey 1992], the resulting specifications may be as difficult to construct and reason about as the implementation itself. Furthermore, there are no effective means to automate the checking of such specifications against an implementation. In contrast, for finite-state verification, analysts typically write small specifications that capture focused system properties. This eases the burden on the specifier and allows the analysis, in turn, to be tailored to each property. For a property to be verified by FLAVERS, it must either be expressed directly as, or translated into, a deterministic FSA. This formalism enables FLAVERS to check event-based properties on finite executions.[1] FLAVERS supports both a QRE textual specification and a diagrammatic representation of a quantified FSA.

A QRE consists of an alphabet, a quantifier, and the regular expression to be satisfied. Figure 2 gives the formal syntax for the QRE notation used in FLAVERS. The QRE alphabet contains events that may be used in the regular expression. The meaning of each event symbol is established by the mapping to the appropriate system action, where an occurrence of the symbol represents the execution of the associated action in the system, as described in Section 4.

Quantifiers are included as a convenience. It is often conceptually simpler for an analyst to describe a violation of a property than to describe all of the legal behaviors. The `no` quantifier indicates that no event sequence corresponding to an execution of the system should lie in the language of the specified regular expression; properties using this quantifier specify *undesirable behaviors*. The `all` quantifier indicates that all event sequences corresponding to executions of the system should lie in the language of the specified regular expression; properties using this quantifier specify *desirable behaviors*. Since regular expressions are closed under complement, the expressive power of `all` properties and `no` properties are equivalent.

The `show` clause of a QRE is a statement of the property to be proved and consists of a quantifier and a regular expression defined over the specified alphabet. These regular expressions have the standard meaning given in detail in Olender and Osterweil [1990]. Note that "." is used to refer to any event in the alphabet, "∼" is set complement, ";" is the concatenation operator, "^pos" is

[1]FLAVERS has also been extended to support properties on infinite executions using an extended specification notation [Naumovich and Clarke 2000]. We do not cover this extension in this article.

```
qre              :   alphabet_clause show_clause

alphabet_clause  :   "for" "events" alphabet

show_clause      :   "show" quantifier "executions"
                       "satisfy" expression

alphabet         :   "{" idlist "}"

quantifier       :   "all" | "no"

idlist           :   id
                 |   id "," idlist

id               :   [a-zA-Z][a-zA-Z0-9_:=<>!]*

pos              :   [1-9][0-9]*

expression       :   "."
                 |   id
                 |   "∼" "[" idlist "]"
                 |   "[" idlist "]"
                 |   expression ";" expression
                 |   expression "|" expression
                 |   "(" expression ")"
                 |   expression "^" pos
                 |   expression "*"
                 |   expression "+"
                 |   expression "?"
```

Fig. 2. QRE syntax.

used to represent a positive number of repetitions, and "?" means zero or one repetitions.

As an example of a QRE, consider a desirable property of any automated teller machine (ATM) stating that "a valid PIN must be entered before a transaction can take place." Assume the events are `valid` and `invalid`, to indicate when a valid or invalid PIN is entered, respectively, and `transaction`, to indicate when a transaction takes place. The property could then be written as:

```
for events {valid, invalid, transaction}
show all executions satisfy invalid*; (valid; transaction?)?
```

which states that any number of invalid PINs may be entered, but a transaction cannot take place until after a valid PIN is entered. For simplicity, this property only deals with the entry of PINs with respect to a single transaction, thus no events from the alphabet of the property are allowed after the transaction takes place. A variety of other properties specified using QREs are described in Section 7.

FLAVERS also provides support for analysts to directly construct an automaton representation of the property. This form of a property consists of an alphabet and a quantifier, both identical to those used in QREs, and a graphical depiction of the FSA. As is the usual convention, the start state has an incoming

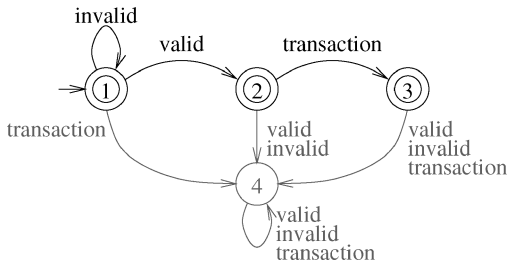

Fig. 3. FSA of the ATM property.

arrow with no source and accept states are represented by concentric circles. The automaton form for the above ATM property is illustrated in Figure 3.

The property, whether specified with a QRE or FSA, is translated into a property automaton, an equivalent, canonical form that is a minimal, deterministic, and total FSA. By total we mean that each state must indicate an outgoing transition for each event in the alphabet. Making an automaton total is done by adding transitions for each missing event from a state to the *trap state*, a nonaccepting state with only self-loop outgoing transitions. The trap state and the transitions leading to it are shown in gray in all pictures of property automata. Standard algorithms are used to make the representation minimal and deterministic [Aho et al. 1986]. Note that there exist expressions for which this translation requires time exponential in the size of the expression but, in our experience, QRE properties tend to be small, and so this cost tends to be small compared to the total cost of a FLAVERS analysis. The canonical representation that we use is not required, but simplifies the dataflow algorithm.

Definition 3.1. A **property automaton** is a deterministic FSA $(\Sigma, S, \delta, A, s, s_{trap})$, where:

Σ is the alphabet of the property,
$S = \{s_0, s_1, \ldots, s_k\}$ is the set of property automaton states that represent equivalence classes of strings over Σ,
$\delta : S \times \Sigma \rightarrow S$ is the total state transition function,
$A \subseteq S$ is the set of accepting states,
$s \in S$ is the unique start state, and
$s_{trap} \in S$ is the unique trap state.

We use $\mathcal{L}(P)$ to denote the set of all strings accepted by a property automaton P. QREs and their corresponding automata representations are able to succinctly express a wide variety of commonly occurring behavioral properties used in finite-state verification [Dwyer et al. 1999].

4. TRACE-FLOW GRAPH MODEL OF SYSTEM EXECUTION

We are interested in reasoning about sequences of events that can occur during execution. Therefore, it is important that our model of the system be a conservative representation of all the possible sequences of these events that could actually occur. A program may manipulate data that can range over an infinite or very large set of values, however. Thus, in practice, the

```
task body customer is
begin
  atm.insert_card;
  loop
    atm.enter_PIN;
    select
      accept valid;
      atm.transaction;
      exit;
    or
      accept invalid;
    or
      accept eat_card;
      exit;
    end select;
  end loop;
end customer;
```

```
task body atm is
  count : integer range 0 .. 2;
begin
  count := 0;
  accept insert_card;
  loop
    accept enter_PIN;
    if(validPIN) then
      customer.valid;
      accept transaction;
      exit;
    else
      if(count = 2) then
        customer.eat_card;
        exit;
      else
        customer.invalid;
        count := count + 1;
      end if;
    end if;
  end loop;
end atm;
```

Fig. 4. Ada tasks for the ATM example.

precise execution behavior of a system cannot be efficiently captured by a finite-state model. Instead, our finite-state model abstracts some system behaviors, while remaining conservative. This model is constructed in a way that preserves all the relevant event sequences, but may not model all the possible values of variables. Therefore, it is possible that additional event sequences may be represented. If this occurs, the analysis will be conservative, but may return spurious results. In Section 6, we describe a mechanism by which users can refine the precision of the model, based on the knowledge they have obtained from past analyses. Thus, in considering the trade-off between creating a model that could produce conclusive results, but will often be too large to support cost-effective analyses, and one that will usually be tractable, but may lead to inconclusive results, we err on the side of the smaller model. In this section, we describe the TFG model, demonstrate that this model is easy to construct, and show that it supports conservative state-propagation analysis.

To illustrate the model, we use the small, concurrent Ada program shown in Figure 4. In Ada, a system consists of a set of threads of control, called *tasks*, that may run in parallel. The basic construct for communication and synchronization between tasks is the *rendezvous*, a form of synchronous communication. A task may call on a named *entry* in another task; execution of the calling task is then blocked until the called task accepts the call and the two tasks complete the rendezvous, possibly passing information in both directions. A call that has not yet been accepted is *pending*. A task declaring a particular entry `e` may accept a pending call from another task on that entry by executing an `accept e` statement. If no calls on this entry are pending, the accepting task is blocked until a call on entry `e` is made by another task. Both the calling task and the accepting task continue execution after their rendezvous is completed.

In Figure 4, the system contains two tasks, `customer` and `atm`, that simulate the operation of an ATM. The customer begins by inserting a card, which is represented by the entry call `atm.insert_card`. After a rendezvous with the ATM, the customer enters the PIN, represented by the entry call `atm.enter_pin`. After this rendezvous, the customer is notified by the ATM whether the PIN is valid, invalid, or, in the case where there have been too many invalid PIN entries, that the card is invalid and will be "eaten" by the machine. If the PIN is found to be valid, then the customer can perform a transaction and remove the card from the machine. In this example, `validPIN` represents a function call that contains no events of interest and thus is treated as an atomic action.[2]

4.1 Trace-Flow Graph Construction

A TFG captures all potential event orderings that might occur during execution of the system. The TFG consists of:

—a collection of RCFGs, one for each task in the system,
—additional nodes and edges to represent intertask synchronization and communication actions, and
—additional edges to represent intertask event orderings.

We consider each TFG component, in turn, and build up the definition of a TFG in three steps.

4.1.1 *Refined Control-Flow Graphs.* The RCFG is derived from a typical CFG model [Aho et al. 1986], which we assume is a conservative approximation of the possible sequences of statements that a task could execute [Marlowe and Ryder 1990]. In this section, we define CFGs whose nodes are labeled with events, and then introduce a procedure for converting them into RCFGs.

Definition 4.1. A **CFG** G is a labeled directed graph $(N, n_{initial}, n_{final}, E, label)$, where:

N is the set of CFG nodes,
$n_{initial} \in N$ is the unique initial node of G,
$n_{final} \in N$ is the unique final node of G,
$E \subseteq N \times N$ is the set of directed edges, and
$label : N \rightarrow \Sigma_S \cup \{\tau\}$ is a labeling function that maps a node to its associated event, where Σ_S is the set of all events for a given system and τ is a special, "empty" event.

Tasks can call subprograms, which can in turn call other subprograms. We inline the subprogram's CFG for each called subprogram.[3] Although, in the worst case, this inlining can result in an explosive growth in the size of the

[2]To save space, we do not show the PIN parameter, since it is not needed in the subsequent analyses.
[3]The assumption that inlining is performed before construction of RCFGs is made only for simplicity of the presentation. In our implementation, inlining is performed after CFGs are refined. FLAVERS does not currently treat recursive subprograms, although techniques for replacing tail recursion with iteration could be incorporated.

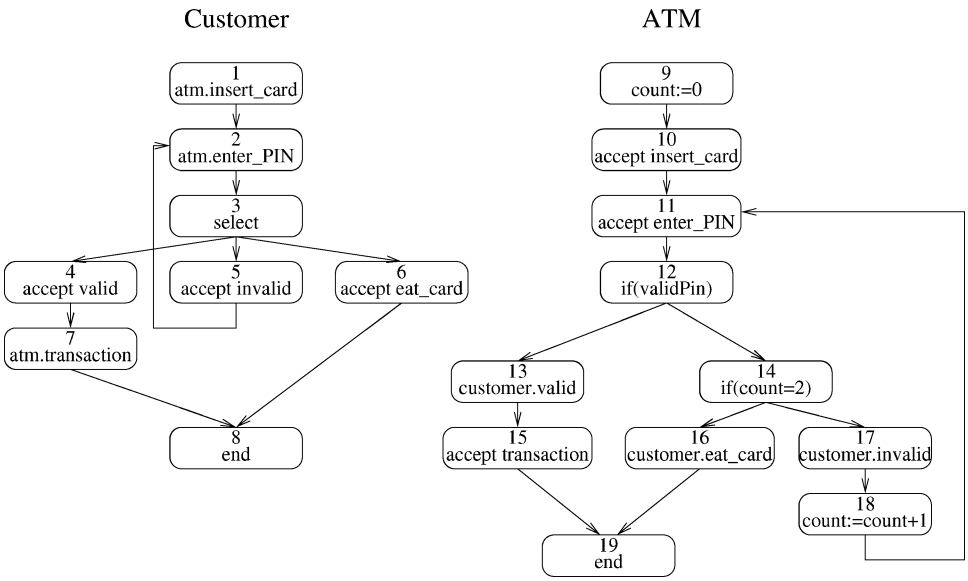

Fig. 5. CFGs for the tasks in Figure 4.

graph, we have found that typically this growth is modest when combined with CFG alphabet refinement, described below, since this refinement tends to remove the vast majority of nodes. This significant reduction is not evident in the small example given here, but is illustrated for larger examples in Section 7.

For expediency, an analyst might choose to form a CFG for a set of analyses. Thus, not all events in the system alphabet Σ_S may be needed for a given FLAVERS analysis. Let $\Sigma_I \subseteq \Sigma_S$ represent the actual set of events of interest. This set would contain at least all the events in the property. Also, note that CFGs are affected by the granularity of system events. For example, where the canonical procedure for creating CFGs creates a node corresponding to a basic block [Aho et al. 1986], our procedure for creating annotated CFGs may create multiple nodes for a basic block, where each node corresponds to a single event in this block. At present, the FLAVERS/Ada toolset recognizes some types of events automatically, such as task communications, but other types of events are indicated by the use of stylized comments placed in the source code.

A CFG conservatively represents each sequence of system events that may be observed on an execution of a task, with at least one path exhibiting precisely the same sequence of events. Figure 5 shows CFGs for the two tasks from Figure 4. For clarity, the nodes of the CFGs are labeled with system events that represent their associated source statement.

Since a rendezvous can only occur in a concurrent system and CFGs are representations of sequential code, the call and accept parts of a rendezvous are each represented by events. These constituent parts are later replaced by the appropriate rendezvous event during the construction of the TFG. Let $\Sigma_C \subseteq \Sigma_S$ contain the set of all intertask communication events, the labels that correspond to either entry calls or `accept` statements.

Input: A CFG $G = (N^G, n^G_{initial}, n^G_{final}, E^G, label^G)$, an alphabet of events of interest Σ_I, and an alphabet of intertask communication events Σ_C
Output: A RCFG $G' = (N^{G'}, n^{G'}_{initial}, n^{G'}_{final}, E^{G'}, label^{G'})$

begin
  copy G to G'
  let $Retained = \{n \in N^{G'} \mid label^{G'}(n) \in \Sigma_I \cup \Sigma_C \vee n = n^{G'}_{initial} \vee n = n^{G'}_{final}\}$
  // remove nodes not labeled with events of interest or intertask communication events
  forall nodes $n \in (N^{G'} \setminus Retained)$:
    let $Preds = \{p \mid (p, n) \in E^{G'} \wedge p \neq n\}$
    let $Succs = \{s \mid (n, s) \in E^{G'} \wedge s \neq n\}$
    forall pairs (p, s), where $p \in Preds, s \in Succs$, add edge (p, s) to $E^{G'}$
    remove n and all of its incident edges from G'
  // assign a τ label to nodes not labeled with events of interest or intertask
  // communication events
  forall nodes $n \in N^{G'}$:
    if $label^{G'}(n) \notin \Sigma_I \cup \Sigma_C$:
      assign $label^{G'}(n) = \tau$
end

Fig. 6. CFG alphabet-refinement algorithm.

We want the model of the system to be as small as possible, yet still be conservative. We therefore transform each CFG into a RCFG, using an *alphabet-refinement* algorithm that eliminates any nodes that are not labeled with an event from Σ_I, the events of interest, Σ_C, the intertask communication events, or $n_{initial}$ and n_{final}, the initial and final nodes. The alphabet-refinement algorithm[4] is given in Figure 6. This algorithm also relabels with τ the initial and final nodes of the RCFG, unless they are labeled with an event from $\Sigma_I \cup \Sigma_C$. Thus, after this algorithm terminates, each node in the RCFG is labeled with τ, or with an event from $\Sigma_I \cup \Sigma_C$. Note there is a weak bisimulation relationship [Milner 1989] between a CFG and its RCFG.

To refine the CFGs in Figure 5 to be employed in the verification of the property in Figure 3, the following alphabets are used:

Σ_I = {`invalid`, `valid`, `transaction`}
Σ_C = {`atm.insert_card`, `atm.enter_PIN`, `accept valid`, `accept invalid`, `accept eat_card`, `atm.transaction`, `accept insert_card`, `accept enter_PIN`, `customer.valid`, `accept transaction`, `customer.eat_card`, `customer.invalid`}.

The resulting RCFGs are shown in Figure 7. Note that, in this example, none of the events in Σ_I occur on any of the nodes of the CFGs or RCFGs since these events correspond to rendezvous events. These events are added to nodes of the TFG when the TFG is created.

[4]In our implementation of this algorithm we do not remove a node if its removal would increase the number of edges in the RCFG. For example, a node with three predecessors and two successors would not be removed because after this node and its adjacent five edges are removed, six new edges would have to be added to the RCFG.

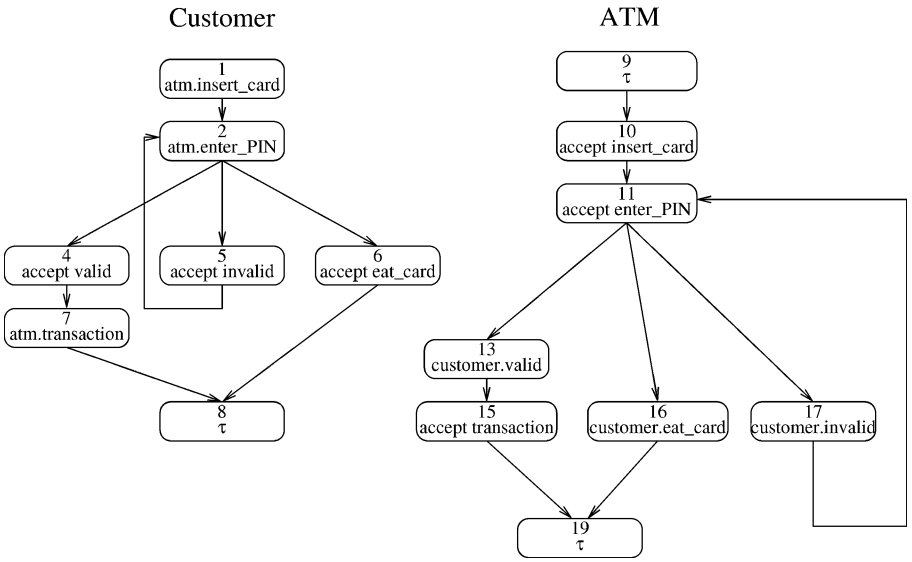

Fig. 7. RCFGs, for the CFGs in Figure 5, refined for the alphabet of the property in Figure 3.

To define the safety of RCFGs, we introduce the projection operator $\sigma|_{S'}$, which takes a sequence $\sigma = s_0, s_1, \ldots, s_n$ of elements defined over a domain, $s_i \in S$, and a subset of that domain, $S' \subseteq S$, and returns a sequence $\sigma' = s_{j_0}, s_{j_1}, \ldots, s_{j_m}$ that retains only those elements in σ that are in S', meaning $s_i \in S' \iff (\exists j_k \mid j_k = i)$. Furthermore, the order of elements in the projected sequence is preserved, meaning $j_k < j_{k+1}$.

We define a *path* through a CFG and other graphs as starting at the initial node, but not necessarily reaching the final node. Formally, we say that a sequence of nodes $n_0, \ldots, n_k$ is a path through graph G if $n_0 = n^G_{initial}$ and $\forall i, 1 \leq i \leq k, (n_{i-1}, n_i) \in E^G$.

THEOREM 4.2 (RCFG CONSERVATIVENESS). *Let G be a CFG and G' the corresponding RCFG obtained using the alphabet-refinement algorithm. Let $R \subseteq N^G$ be the set of nodes of G that were retained in G'. For any sequence of nodes $\pi \in (N^G)^*$, let $Map(\pi) \in (N^{G'})^*$ be the sequence of nodes in G' that corresponds to the nodes in the projection $\pi|_R$. If π is a path in G, then $Map(\pi)$ is a path in G'.*

PROOF. The proof of this theorem appears in Appendix A.1. □

Once the RCFGs are formed for all the tasks in the system, the first step of the TFG construction is to include the nodes and edges from these RCFGs. We say that these nodes and edges are *local* to their respective RCFG, and thus distinguish the local nodes and edges from the communication nodes and edges that are added by the next step to represent intertask communication. We refer to the set of edges that connect TFG nodes that correspond to nodes in the same RCFG as E_{local}.

4.1.2 *Task Synchronization.* The TFG models all potential synchronization actions by adding a *communication node* between two matching

synchronous interactions. A communication node, c, is created for each pair of local nodes, n and m, from different RCFGs, G_i and G_j: $n \in N^{G_i}, m \in N^{G_j}, i \neq j$, where n and m are labeled with events that represent matching communication statements (i.e., one is labeled with an `accept` statement, and the other is labeled with an entry call on the same rendezvous). An edge to c is created from each of n and m. In addition, each edge from E_{local} from n or m to a local node s is replaced by an edge from c to s. If the label on n is not in Σ_I, we relabel node n with τ; a similar relabeling is done on node m. If the rendezvous corresponds to an event e in Σ_I, node c is labeled with e, otherwise it is labeled τ. Since a communication node c represents a synchronization between two tasks, we consider node c to be part of both tasks involved in the rendezvous. We use N_{com} to denote the set of communication nodes in the TFG and E_{com} to denote the set of control edges that connect communication nodes to other nodes in the TFG.

In addition to communication nodes, we also create a unique *initial node* and a unique *final node* for the TFG. The initial node of the TFG represents the synchronization of tasks at the start of execution. It does not have incoming edges, but has edges to the initial nodes of each of the task RCFGs. The final node of the TFG represents termination of the system execution. An edge is created from the final node of each task's RCFG to the final node of the TFG. The final node of the TFG has no outgoing edges.

Figure 8 shows the TFG for the example in Figure 4. In this TFG representation, communication nodes are shown as diamonds and the initial and final nodes are shown as triangles. For example, RCFG nodes 13 and 4 in Figure 7 represent a call on entry `valid` and an accept of this entry, respectively. As shown in Figure 8, node 24 is the communication node that is constructed to represent the task interaction on entry `valid`. Since this rendezvous event is in Σ_I, it is labeled `valid`. Node 21 represents the rendezvous `insert_card`, which is labeled τ since `insert_card` is not in Σ_I. Note that, unlike in the RCFG for task `Customer` in Figure 7, local nodes 1 and 2 are no longer connected by a control edge. This represents the fact that control in task `Customer` can flow from node 1 to node 2 only after completion of the task synchronization represented by the communication node 21.

Different types of task interactions require different patterns of communication nodes. For example, Ada `accept` statements may have *bodies* that consist of executable code. When an entry call is accepted by such an `accept` statement, the calling task blocks, while the `accept` body is executed. To represent such interactions, we need two communication nodes, one to represent the start of the entry call and the other to represent the end of that call.

4.1.3 *Task Interleavings.* The TFG must explicitly represent all possible orderings of events that could occur in concurrently executing tasks. These orderings must be considered during the FLAVERS analysis, and so we add edges (m, n) to the TFG to represent the possibility that execution of code corresponding to node m *may immediately precede* (MIP) execution of code corresponding to node n, where m and n belong to disjoint sets of tasks. Note that we assume an interleaving model of concurrent execution, which means that if two events `a` and `b` in the system may happen in parallel, both sequences `ab` and `ba` must be

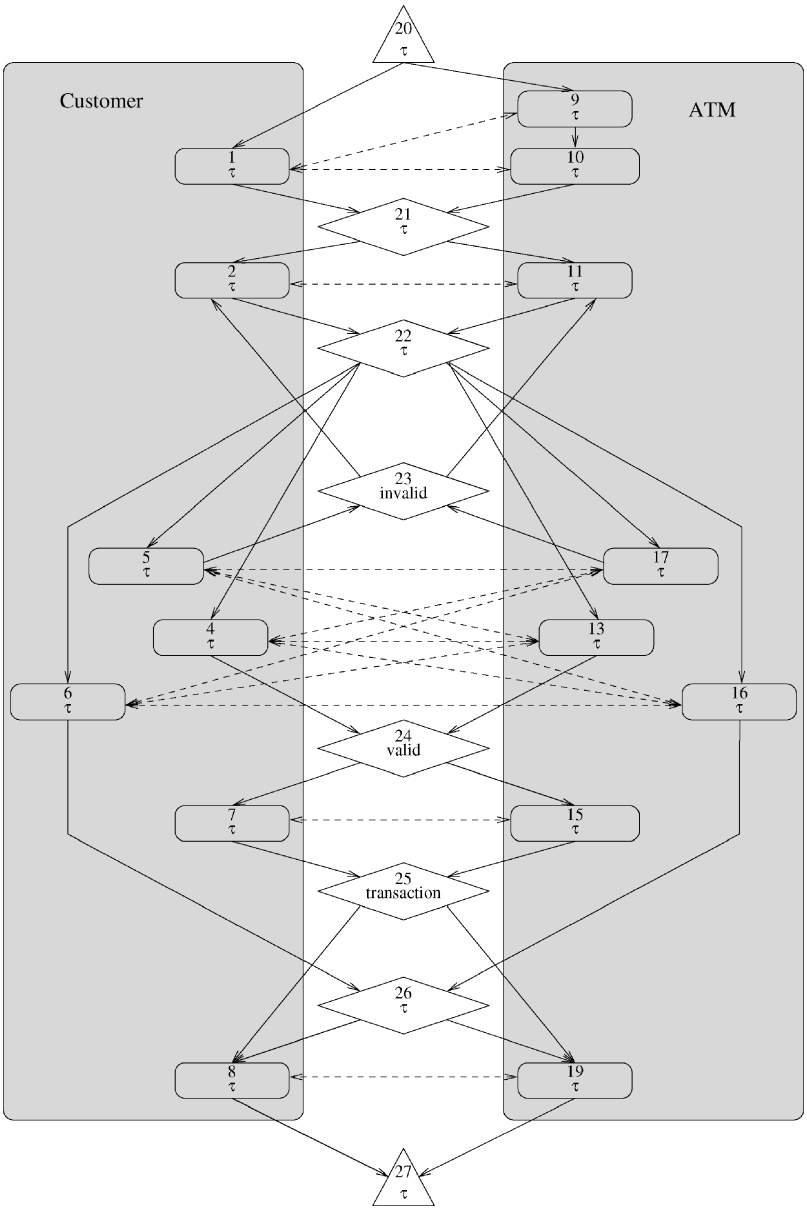

Fig. 8. TFG, for the example in Figure 4, using the RCFGs in Figure 7.

represented in the model. In Figure 8, MIP edges are shown as dashed arrows (a bidirectional dashed arrow connecting two nodes m and n represents two MIP edges, (m, n) and (n, m)). We denote the set of all MIP edges of the TFG by E_{mip}.

A naïve approach to computing MIP edges for a TFG is to create an edge for all pairs of nodes m and n such that the sets of tasks of m and n are disjoint.

Although the resulting set of edges would conservatively capture the MIP relation, many of these MIP edges would connect nodes that, in reality, could not immediately precede each other. For example, in the TFG in Figure 8, node 11 may not immediately precede node 1, because for task `ATM` to execute node 11, a synchronization with task `Customer` on entry `insert_card` must have happened (node 21), which, in turn, requires that `Customer` completes the execution of node 1. A *spurious* MIP edge, such as (11, 1), leads to imprecision in the analysis, since it introduces paths in the graph that do not correspond to real system executions. In addition, processing these extra edges increases analysis cost.

Although, in general, the problem of precisely determining which code regions may immediately precede each other is undecidable, it is possible to compute a conservative approximation efficiently. We compute a conservative approximation of the minimal set of necessary MIP edges by using the *may happen in parallel* (MHP) analysis [Naumovich and Avrunin 1998]. The MHP analysis uses a nonsymmetric dataflow analysis algorithm to compute a conservative estimate of pairs of TFG nodes representing regions of code that may happen in parallel with each other. The worst-case complexity of the MHP analysis is cubic in the number of TFG nodes. Based on the MHP information and the task interaction semantics, we compute a set of MIP edges.

4.2 Formalization of the TFG

We construct the TFG from the local nodes of each individual task RCFG with additional communication, initial, and final nodes. The edges of the TFG preserve the local control-flow edges that do not involve task synchronization, but also include edges to represent task synchronization and interleaving explicitly.

Definition 4.3. Assume that a system is composed of k tasks, where each task T^i has a RCFG $(N^i, n^i_{initial}, n^i_{final}, E^i, label^i)$, where N^i is the set of nodes and $n^i_{initial}$ and n^i_{final} represent the unique initial and final node, respectively, $E^i \subseteq N^i \times N^i$ is the set of edges, and $label^i$ relates a node in N^i to an event in Σ_I or τ. A **TFG** is a labeled directed graph $(N, n_{initial}, n_{final}, E, \Sigma_I, label)$ where:

$N = \{n_{initial}, n_{final}\} \cup N_{com} \cup \bigcup_{1 \le i \le k} N^i$ is the set of TFG nodes, where N_{com} is the set of communication nodes,
$n_{initial}$ is the unique initial node,
n_{final} is the unique final node,
$E = E_{com} \cup E_{mip} \cup E'_{local} \cup \bigcup_{1 \le i \le k} \{(n_{initial}, n^i_{initial})\} \cup \bigcup_{1 \le i \le k} \{(n^i_{final}, n_{final})\}$ is the set of TFG edges, where E_{com} is the set of edges between local nodes and communication nodes, E_{mip} is the set of edges representing the interleaving among tasks, and $E'_{local} = \bigcup_{1 \le i \le k} (E^i \setminus R^i)$, where R^i is the set of RCFG edges removed when adding communication nodes,
Σ_I is the set of events of interest defined by the analyst, and
$label : N \to \Sigma_I \cup \{\tau\}$ maps a TFG node to its associated event, where

$$label(n) = \begin{cases} label^i(n) & \text{if } n \in N^i \\ e & \text{if } n \in N_{com},\ e \in \Sigma_I, \text{ and } e \text{ is the event associated with the rendezvous represented by } n \\ \tau & \text{otherwise.} \end{cases}$$

For the TFG, we define two functions to return the successor and predecessor nodes for a given node n:

$$Preds(n) = \{p \mid (p, n) \in E\}$$
$$Succs(n) = \{s \mid (n, s) \in E\}.$$

A path in the TFG is assigned meaning via the concatenation of non-τ labels on nodes in the path. Let Π be the set of all paths through the TFG that start with the initial node. We introduce function $Labels : N^* \rightarrow {\Sigma_I}^*$, which maps a sequence of nodes to a sequence of events in the set Σ_I such that:

$$Labels(n_0, n_1, \ldots, n_k) = label(n_0), label(n_1), \ldots, label(n_k)|_{\Sigma_I}.$$

Informally, for a path π, $Labels(\pi)$ is the sequence of events of interest on the nodes in the path. We use $\mathcal{L}(G)$ to denote the set of all event sequences in the TFG:

$$\mathcal{L}(G) = \{Labels(\pi) \mid \pi \in \Pi\}.$$

Let $\mathcal{L}(S)$ be the set of all prefixes of event sequences, projected on alphabet Σ_I, of all actual executions of the system S, still assuming an interleaving model of concurrent execution.

THEOREM 4.4 (TFG CONSERVATIVENESS). *For each actual execution of the system S, the TFG contains a path, starting in the initial node, that exhibits the same set of events projected on Σ_I as this execution projected on Σ_I. Thus, $\mathcal{L}(S) \subseteq \mathcal{L}(G)$.*

PROOF. A formal proof of this theorem appears in Appendix A.2. □

Note that the reverse of the statement of this theorem is not true. There may exist paths in the TFG that do not correspond to actual executions of the system. For example, in Figure 8, the path 20, 1, 21, 2 does not correspond to an actual execution of the system, because the synchronization represented at node 21 cannot happen until nodes 9 and 10, in task ATM, execute. We address this source of imprecision through the use of constraints, described in Section 6.

THEOREM 4.5 (SIZE OF TFG). *The number of nodes in a TFG, $|N|$, is $\mathcal{O}(S^2)$, where S is the sum of the number of nodes in the task RCFGs. The number of edges in a TFG is $\mathcal{O}(S^4)$.*

PROOF. Let C be the number of entry call nodes and A the number of accept statement nodes in the task RCFGs. The number of nodes in the TFG is $\mathcal{O}(S + A*C+2)$. This is the sum of the number of RCFG nodes, communication nodes, and 2 for the initial and final nodes. In the worst case, if $C \approx A \approx S/2$, the bound on $|N|$ becomes $\mathcal{O}(S^2)$.

A TFG has at most two directed edges between any pair of nodes. Therefore, the number of TFG edges is, in the worst case, $\mathcal{O}(|N|^2) = \mathcal{O}(S^4)$. □

The product of the number of entry calls and accept statements is more accurately computed on a per-entry basis, and then summed over the set of entries; this value can be significantly less than the product of all entry call and accept statements. In addition, communication statements are typically a small

percentage of the statements in a given system. Consequently, we have found that the worst-case quadratic blow-up in the number of TFG nodes, compared to the total number of RCFG nodes, is not common in practice. In the experimental study reported here and in an experimental study of 159 mostly small Ada programs [Naumovich and Avrunin 1998], it was found that the nodes labeled with events from Σ_I tend to be sparse in the graph and that the number of TFG nodes tends to be linear in the number of RCFG nodes and, thus, linear in the number of statements in the program.

THEOREM 4.6 (COMPLEXITY OF THE TFG CONSTRUCTION ALGORITHM). *The time required to construct a TFG from a collection of CFGs is $\mathcal{O}(S^6)$, where S is the sum of the number of nodes in the task RCFGs.*

PROOF. The TFG construction consists of several steps. First, RCFGs are constructed, which takes $\mathcal{O}(S^2)$ time. Next, the initial, the final, and the communication nodes and then the MIP edges are added. Since there are at most $\mathcal{O}(S^2)$ communication nodes, adding communication nodes takes $\mathcal{O}(S^2)$ time. MHP analysis is $\mathcal{O}(N^3)$, where N is the number of TFG nodes and, therefore, is $\mathcal{O}(S^6)$. The procedure for adding MIP edges is linear in the number of MHP pairs, which is $\mathcal{O}(N^2) = \mathcal{O}(S^4)$. Thus, the complexity of constructing the TFG is dominated by the complexity of the MHP analysis, which is $\mathcal{O}(S^6)$. □

In our experiments, the time to construct the TFG was at most cubic in the size of the system, as described in Section 7.

4.3 Reducing the Size of the TFG

Clearly, the size of the TFG has a major impact on the cost of the analysis. CFG refinement tends to reduce the size of the graph significantly. In addition, there are several other optimizations that can be employed.

One such optimization is partial-order reduction, which may remove some of the MIP edges from the graph, using an algorithm that takes into account partial orders of events from different tasks [Naumovich et al. 1999]. This algorithm is conservative in the sense that a MIP edge is only removed if it can be shown that for each possible event sequence in the original TFG, there is an event sequence in the reduced TFG that has an identical effect on the property. Note that although the initial computation of MIP edges based on the MHP analysis does not depend on the alphabet, this partial-order reduction does. In our evaluation of this optimization, the partial-order reduction, on average, removed 25% of the MIP edges and improved the analysis time by about 20% on the examples where this optimization was applicable [Naumovich et al. 1999].

We can use the MHP analysis to further reduce the size of the TFG since, in some cases, it can determine that a communication node represents a synchronization that can never happen. Let nodes m and n correspond to an entry call and an `accept` on the same rendezvous. As described in Section 4.1.2, the TFG construction algorithm creates a communication node c with m and n as predecessors. A necessary condition for an intertask communication is the possibility that the entry call and the accept statement execute simultaneously. Therefore,

if the MHP analysis determines that m and n may not happen in parallel, it means that the rendezvous represented by c can never occur. We can remove node c from the TFG. After removing all nonexecutable communication nodes and their incident edges, we can do a pass through the TFG removing all the nodes that cannot be reached using edges in $E \setminus E_{mip}$.

We perform additional conservative structural optimizations during TFG construction, but do not include them in the formal description since they complicate the discussion about conservativeness and complexity bounds. The TFG construction algorithm, as presented, may add unnecessary τ nodes to the TFG. For example, in the TFG in Figure 8, node 9 was retained because it was the initial node in the `ATM` task, and node 10 was retained because it corresponded to an `accept` on the `insert_card` rendezvous. Since nodes 9 and 10 are τ nodes, they have no effect on the state of the property, and one of these nodes can be removed from the TFG without changing the set of non-τ event sequences in the TFG.

In addition, notice that node 22 has three successors in task `Customer` that are τ nodes, nodes 4, 5, and 6. We can replace these three nodes by a single node with an out-edge to every node that has an out-edge from node 4, 5, or 6. A similar optimization could be made to collapse nodes 13, 16, and 17 into a single node. Neither of these optimizations affect the set of non-τ event sequences associated with a TFG.

Figure 9 shows the resulting TFG after applying these optimizations to the TFG shown in Figure 8. In this TFG, node 10 has been removed; nodes 4, 5, and 6 have been replaced with a single node, numbered 4; and nodes 13, 16, and 17 have been replaced with a single node, numbered 13. There are some additional annotations on the nodes that can be ignored for now.

5. STATE-PROPAGATION ANALYSIS

FLAVERS uses state propagation to compare the executable behavior of a system with event sequences defined in a property. Howden [1986], and later Olender and Osterweil [1992], developed state-propagation algorithms for checking properties of sequential systems. Our work builds on these results. Here, we extend the state-propagation analysis for sequential systems [Olender and Osterweil 1992] to concurrent systems modeled as a TFG.

Let $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$ be a property automaton, and let $G = (N^G, n^G_{initial}, n^G_{final}, E^G, \Sigma^G, label^G)$ be a TFG. Recall that $\mathcal{L}(P)$ is the language of the property, the set of all event sequences accepted by P. Let $\mathcal{L}(G^{\text{term}}) \subseteq \mathcal{L}(G)$ be the set of all event sequences found on paths that start at the initial node of G and end at the final node of G. We check consistency between the property and system by checking either language containment, $\mathcal{L}(G^{\text{term}}) \subseteq \mathcal{L}(P)$, if the property specifies desirable behaviors, or empty language intersection, $\mathcal{L}(G^{\text{term}}) \cap \mathcal{L}(P) = \emptyset$, if the property specifies undesirable behaviors of the system.

Intuitively, the objective of state-propagation analysis is to check each path π through the TFG against the property automaton P. If P represents desirable behaviors of the system, then for each path π, the corresponding event sequence

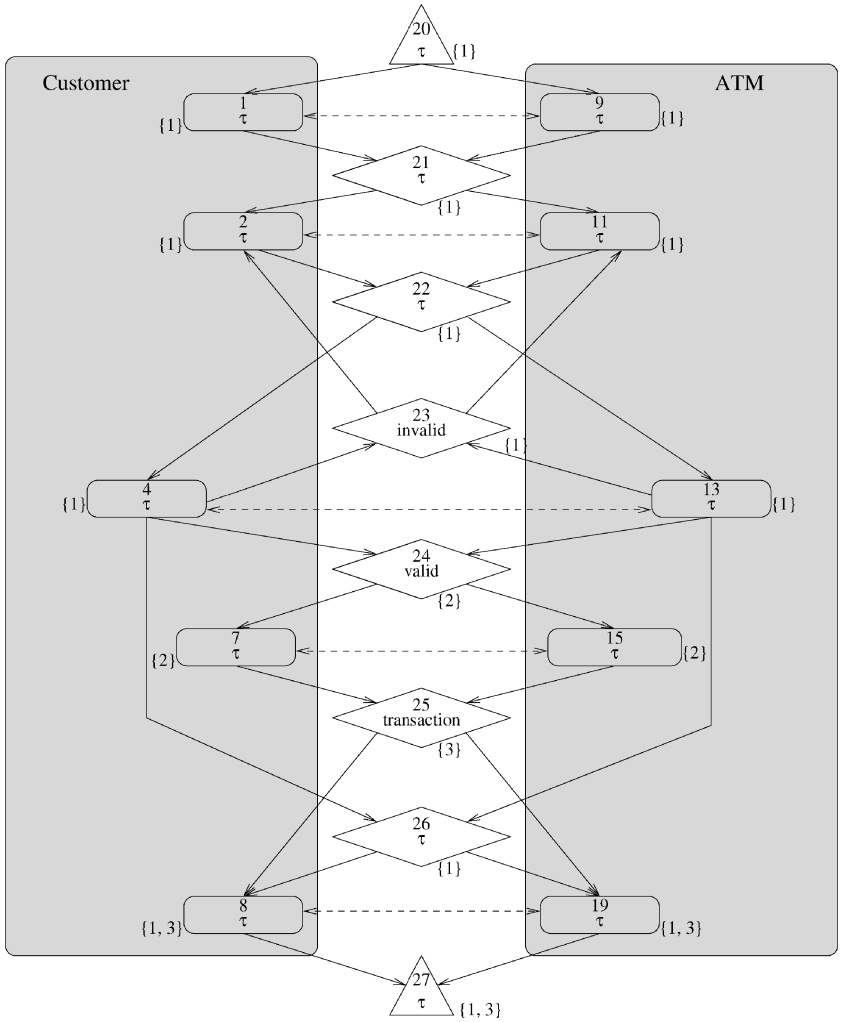

Fig. 9. Optimized TFG, annotated with states obtained from state propagation of the property in Figure 3.

$Labels(\pi)|_{\Sigma^P}$ has to be accepting for P to hold for the TFG. If P represents undesirable behaviors of the system, then for each TFG path π, the corresponding event sequence $Labels(\pi)|_{\Sigma^P}$ must not be accepting for P to hold. Since the number of paths in a TFG may be infinite, we use dataflow analysis to collapse paths into equivalence sets with respect to a property P.

We define state-propagation analysis as an instance of a dataflow framework comprised of a semi-lattice of values and a set of functions defined over those values [Hecht 1977]. The values in the analysis are organized as a join semi-lattice L^P over the power-set 2^{S^P} of P's states. The bottom value $\perp$ for this lattice is $\emptyset$, and the top value $\top$ is S^P. The join operation is set union, $\cup$, and the ordering relation is subset, $\subseteq$.

For each node n in the TFG, we define a *propagation function* $F^n : 2^{S^P} \rightarrow 2^{S^P}$ that captures the effect that this node has on the property automaton:

$$\forall n \in N^G, \forall S \subseteq S^P, F^n(S) = \{s' \in S^P \mid s \in S \wedge s' = f^P(n, s)\},$$

where

$$f^P(n, s) = \begin{cases} \delta^P(s, label^G(n)) & \text{if } label^G(n) \in \Sigma^P \\ s & \text{if } label^G(n) \notin \Sigma^P. \end{cases}$$

Given a flow graph that represents all executable sequences of system events, we can instantiate a dataflow framework for it and be sure that its solution contains a conservative set of property automaton states at each flow-graph node [Marlowe and Ryder 1990]. Informally, this means that the set of states associated with a particular node in the TFG includes all the property states that could be reached along an event sequence corresponding to a path through the TFG from the initial node up to that node. Since the TFG is a conservative representation of all sequences of the events of interest that could occur during execution, the states associated with each node are also a conservative representation of all the property states that could be associated with the system location that is represented by this node.

The solution at the final TFG node can be compared to the set of accepting states of P. If P specifies desirable behaviors, it holds if the set of states computed for the final node of the TFG contains only accepting states. If P specifies undesirable behaviors, it holds if the set of states computed for the final node of the TFG contains only nonaccepting states.

THEOREM 5.1 (STATE-PROPAGATION CONSERVATIVENESS). *If state-propagation analysis over a TFG determines that a property holds, then this property holds for the actual system. Formally, for any TFG node n, if there exists an actual program execution on which state r of the property is associated with n, then after the state-propagation algorithm terminates,* $r \in States[n]$.

PROOF. This proof can be found in Appendix A.3. □

Formulation of our analysis as a monotone dataflow framework provides an algorithm for its solution. Unfortunately, a straightforward implementation of the dataflow framework as an instance of Hecht's iterative worklist algorithm [Hecht 1977] is not very efficient, as it has worst-case complexity of $\mathcal{O}(|N^G||E^G||S^P|^2)$.

We can improve the worst-case complexity of the state-propagation algorithm significantly by optimizing the propagation of states among the TFG nodes. In particular, we would like to ensure that for any two nodes n and p, where $p \in Preds(n)$, if a state s is associated with node p, then n's transition function F^n will be applied to s only once. We achieve this by associating two sets of states with each TFG node, set $States[n]$, which contains all states associated with node n, and set $IN[n]$, which contains all states that have been propagated to n from n's predecessors. On each iteration of the algorithm, we update both sets for a single node n. For each predecessor p of n, we add to $IN[n]$ only those states from $States[p]$ that have not been placed in $States[n]$ on a prior iteration,

and apply the transition function of n to only those states, placing the resulting states in $States[n]$. Figure 10 shows the FLAVERS state-propagation algorithm incorporating this optimization.

THEOREM 5.2 (STATE-PROPAGATION COMPLEXITY). *Given a TFG, with nodes N^G, and a property automaton, with states S^P, the state-propagation algorithm terminates in $\mathcal{O}(|S^P||N^G|^2)$ time.*

PROOF. This proof is given in Appendix A.4 □

5.1 Examples

To illustrate the state-propagation algorithm, we use the optimized TFG in Figure 9, and the property automaton in Figure 3. The result of state propagation is shown in Figure 9 by annotating each TFG node with the set of states that would be associated with it at the end of state propagation.

State propagation begins by associating the start state of the property automaton, state 1, with the initial node of the TFG, node 20. The *Flow* sets of the two out-edges of the initial node, (20, 1) and (20, 9), are initialized to the set containing the start state of the property automaton, {1}. The successors of node 20, nodes 1 and 9, are then added to the worklist. Suppose node 1 is removed from the worklist first. State propagation next computes the set of property automaton states that need to be propagated over node 1. This is the union of the *Flow* sets of the in-edges to node 1, minus the property automaton states that have already reached node 1. This is $(Flow[(20, 1)] \cup Flow[(9, 1)]) \setminus IN[1] = (\{1\} \cup \emptyset) \setminus \emptyset = \{1\}$. The *IN* set of node 1 is next updated to reflect this, so $IN[1] = \{1\}$. Since the property automaton states associated with the in-edges of node 1 have been processed, the *Flow* sets of these edges are reset to the empty set. Since node 1 is a τ node, propagating state 1 of the property automaton across it does not change the state of the property automaton. Since state 1 is not in $States[1]$, state 1 needs to be propagated to the successors of node 1, so $Flow[(1, 9)] = \{1\}$, $Flow[(1, 21)] = \{1\}$, and nodes 9 and 21 are added to the worklist. Since state 1 has been processed on node 1, it is added to $States[1]$, so $States[1] = \{1\}$.

Now, nodes 9 and 21 are on the worklist. Suppose node 9 is removed from the worklist next. The processing of node 9 is similar to the processing of node 1, and results in state 1 being associated with node 9, $Flow[(9, 1)] = \{1\}$, $Flow[(9, 21)] = \{1\}$, and nodes 1 and 21 being added to the worklist. When node 1 gets removed from the worklist, the set of states that needs to be propagated over node 1 will be computed again, which is $(Flow[(20, 1)] \cup Flow[(9, 1)]) \setminus IN[1] = (\emptyset \cup \{1\}) \setminus \{1\} = \emptyset$, meaning no new states need to be propagated over node 1. Thus, the use of *Flow* sets has prevented state propagation from processing states that have already been processed on a node.

At this point, only node 21 is on the worklist. State propagation will process nodes 21, 2, 11, 22, 4, and 13, in a manner similar to the processing of nodes 1 and 9, and will associate state 1 with each of these nodes. After processing node 4 (and 13), state 24 will be on the worklist, with $Flow[(4, 24)] = \{1\}$ and $Flow[(13, 24)] = \{1\}$. This results in state 1 being propagated over node 24. Since node 24 is labeled with `valid`, which causes the property automaton

Input:
A TFG $G = (N^G, n^G_{initial}, n^G_{final}, E^G, \Sigma^G, label^G)$ and a property automaton $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$

Auxiliary data structures:
—*States* is an array of length $|N^G|$ that holds sets of states for each node
—W is a worklist represented as a set of nodes, where at all times $W \subseteq N^G$
—*IN* is an array of length $|N^G|$ that holds sets of incoming states for each node
—*Flow* is an array of length $|E^G|$ that holds sets of states for each edge

Initialization:

$$\forall n \in N^G, States[n] = \begin{cases} \{s^P\} & \text{if } n = n^G_{initial} \\ \emptyset & \text{otherwise} \end{cases}$$

$$W = Succs(n^G_{initial})$$

$$\forall n \in N^G, IN[n] = \emptyset$$

$$\forall (n,r) \in E^G, Flow[(n,r)] = \begin{cases} \{s^P\} & \text{if } n = n^G_{initial} \\ \emptyset & \text{otherwise} \end{cases}$$

Main Loop:
We evaluate the following statements repeatedly until $W = \emptyset$:

Remove a node n from W	/* Remove a node from the worklist */
$V = \bigcup_{p \in Preds(n)} Flow[(p,n)]$	/* Compute the set of all new property states that need to be propagated through n. The union of the *Flow* sets of n's predecessors are the property states that are new to n's predecessor nodes */
$V = V \setminus IN[n]$	/* Remove the states that have already been propagated through n */
$IN[n] = IN[n] \cup V$	/* Update the *IN* set of n so it contains the new property states */
$\forall p \in Preds(n), Flow[(p,n)] = \emptyset$	/* Clear the *Flow* sets of n's in-edges */
$U = F^n(V) \setminus States[n]$	/* Propagate the states in U through n, removing property states that have already been encountered */
if $U \neq \emptyset$ then	/* If there are any new property states, i.e. U is not empty */
$\forall r \in Succs(n)$, $Flow[(n,r)] = Flow[(n,r)] \cup U$	/* Add the new property states to the *Flow* sets of n's out edges */
add nodes in $Succs(n)$ to W	/* Add n's successor nodes to the worklist */
$States[n] = States[n] \cup U$	/* Update the set of states encountered on n to contain the property states in U */

Determining the analysis results:
if P represents desirable behaviors then
  if $States[n^G_{final}] \subseteq A^P$ then
    report conclusive result: the property holds
  else
    report inconclusive result: the property may not hold
else if P represents undesirable behaviors then
  if $States[n^G_{final}] \cap A^P = \emptyset$ then
    report conclusive result: the property holds
  else
    report inconclusive result: the property may not hold

Fig. 10. State-propagation algorithm.

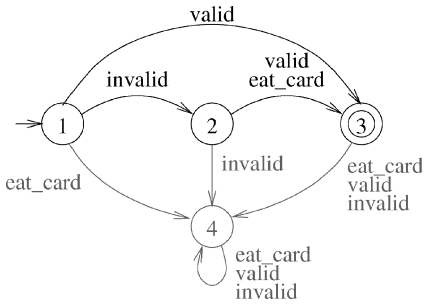

Fig. 11. Property stating that a card is eaten only after an invalid PIN has been entered two times in a row.

to transition from state 1 to state 2, state 2 will be added to *Flow*[(24, 7)], *Flow*[(24, 15)], and *States*[24]. State propagation will continue until a fixed point is reached, at which time states 1 and 3 of the property automaton will be in *States*[27]. Since node 27 is the final node of the TFG, the property being checked is an `all` property, and states 1 and 3 are both accepting states of the property automaton, FLAVERS would conclude that the property holds on all executions.

Figure 11 shows an `all` property automaton stating that "the ATM will eat a card only if an invalid PIN is entered two times in a row." To check this property, FLAVERS produces a new TFG that contains events relevant to this property. This TFG, including the annotations that would result from state propagation, is shown in Figure 12.

In this case, state propagation reports an inconclusive result. This is because nonaccepting state 4 of the property automaton in Figure 11 has been propagated to the final node, node 27. FLAVERS would then present the user with a counterexample path through this TFG, for example 20, 9, 21, 11, 22, 13, 26, 19, 27. This path corresponds to the event sequence `eat_card`. This path is spurious, because along this path the variable `count` is first set to 0, and then the true branch of the statement `if(count=2)` is taken. This occurs because the value of the variable `count` has not been modeled in the analysis. In the next section, we describe how to model variables when they are deemed to be important to the analysis.

Although not shown here, FLAVERS provides support for selecting short paths that violate the property [Tan et al. 2004] and for visualizing paths. Path visualization is particularly important when dealing with medium-size or larger sequential programs, or concurrent programs of just about any size.

6. CONSTRAINTS

FLAVERS, like all practical flow-analysis approaches, is based on abstractions of system control and data information that are encoded into the analysis. The term sensitivity is often used to describe whether a kind of information is (partially) preserved by the abstractions used in a flow analysis. Flow-sensitive and context-sensitive analyses honor, respectively, the sequencing of statements and proper nesting of procedure invocations during system execution.

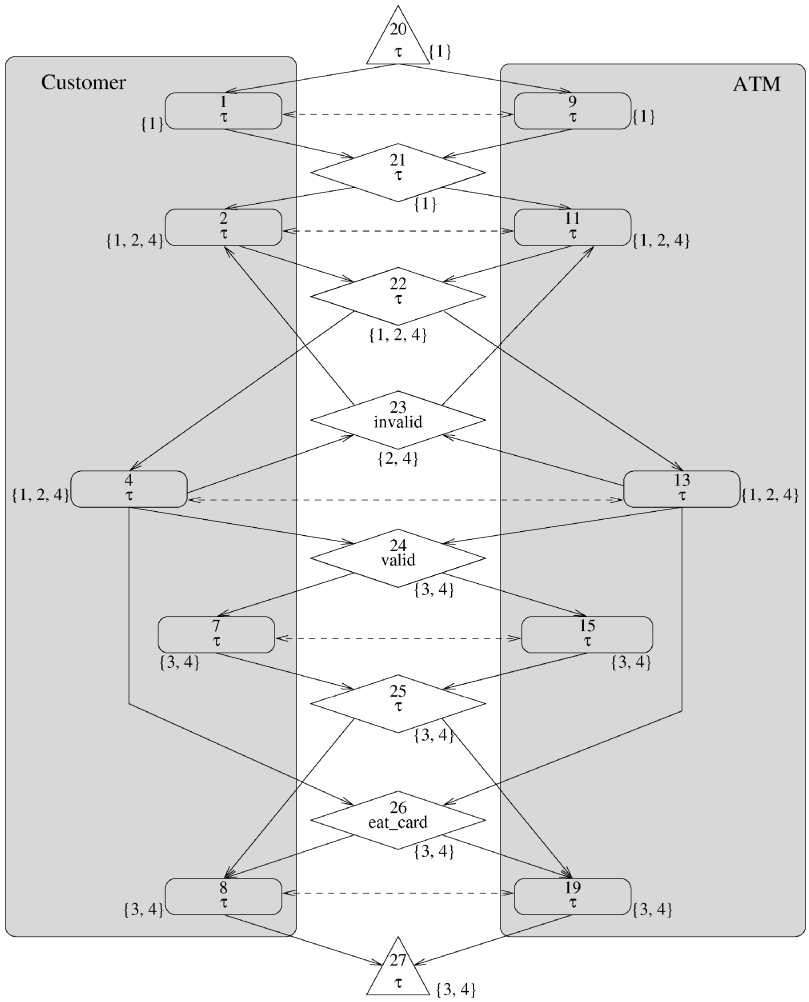

Fig. 12. TFG, for the example in Figure 4, refined for alphabet {`valid`, `invalid`, `eat_card`} and annotated with states obtained from state propagation of the property in Figure 11.

Synchronization-sensitive analyses honor the semantics of synchronization statements during execution of a concurrent system [Ramalingam 2000]. Path-sensitive analyses improve on flow-sensitive analyses by incorporating some path-feasibility information. Typically, path sensitivity is achieved by defining a value-sensitive analysis that captures partial information about program variables (e.g., Holley and Rosen [1981]). Increased sensitivity in an analysis leads to increased precision in analysis results. Unfortunately, enforcing even a few forms of sensitivity, such as flow, context, and synchronization sensitivity, throughout the program in a single analysis makes most interesting analysis problems undecidable [Ramalingam 2000]. The key to efficient and effective flow analysis is in targeting the sensitivity of the analysis to parts of the program that are related to the analysis question.

Figure 12 illustrates the need for increased sensitivity in a FLAVERS analysis. For the example program, a property that is concerned with the number of times an invalid PIN has been entered would need to monitor the possible values of variable `count`. The CFG, and consequently the RCFG and TFG, would normally not include such information. As a result, if the state-propagation analysis determines that a property is violated, it is possible that the violation occurs only on infeasible paths. Since excessive spurious violations hinder usability of a tool such as FLAVERS, it is important to develop techniques for improving the analysis precision without sacrificing too much efficiency.

In this section, we describe a flexible mechanism for enforcing different forms of sensitivity in a FLAVERS analysis. FLAVERS allows information that is not explicitly represented in the TFG to be added to an analysis in the form of *constraints*. Constraints are selectively introduced in situations where it is determined that the analysis results are too imprecise without this additional information. If a constraint is determined to be violated along some TFG path, then the effect of that path on the analysis is discarded. Constraints can encode information, such as statement sequencing, variable values, and branch decisions, that can be used to remove infeasible paths and, thus, to increase the sensitivity of a FLAVERS analysis.

In FLAVERS, constraints are represented as FSAs whose transitions are based on events associated with TFG nodes. Each constraint has a single *violation state* that represents the fact that an event sequence leading to that state is inconsistent with the constraint. Intuitively, if on a path through the TFG a constraint automaton enters a violation state, this path is recognized as spurious and removed from consideration. Similar to the state-propagation analysis defined for property automata in Section 5, we define a combined state-propagation analysis to form a qualified flow analysis [Holley and Rosen 1981].

Definition 6.1. A **constraint automaton** is a deterministic FSA $(\Sigma, S, \delta, A, s, v)$, where:

Σ is the alphabet of the constraint,
S is the set of states that represent equivalence classes of strings over Σ,
$\delta : S \times \Sigma \rightarrow S$ is the total state transition function,
$A \subseteq S$ is the set of accepting states,
$s \in S$ is the unique start state, and
$v \in S$ is the unique violation state.

There are several different types of constraints, including variable, task, environmental, and interface. Since constraints can be added for a subset of program features (e.g., a subset of program variables), FLAVERS provides the ability to define partially sensitive flow analyses that are optimized to retain only the information that is necessary to achieve a desired level of precision. Possible FLAVERS analyses range from an unconstrained analysis, which is flow insensitive due to the fact that MIP edges introduce intertask paths that violate control flow within a task, to a fully task and variable constrained analysis, which is flow, synchronization, scalar-value, and path sensitive.

Although variable and task automata are the only kinds of constraints for which FLAVERS currently provides automated support, the concept is very

general. A wide range of finite-state abstractions could be encoded as constraints and incorporated into FLAVERS using state-propagation analysis. We discuss this further in Section 8. User interaction profiles [Bouwens et al. 1996], models of the networking environment [Naumovich et al. 1996], and models of the software environment [Dwyer 1997] have each been incorporated into FLAVERS analyses using constraints.

In the remainder of this section, we describe variable automaton constraints and task automaton constraints in more detail. We then discuss our technique for combining multiple constraints into a single analysis.

6.1 Variable Automaton Constraints

Flow graphs do not typically model system variables. For many systems, however, accurate analysis depends on modeling some data values. As the example in Figure 12 illustrates, we are often interested in modeling variables that are used in conditional statements or in guards that control task communication statements. Many of these variables are defined over small finite domains and modified in a disciplined way. Examples include Boolean variables to which only constant values are assigned and bounded integer counter variables to which only increment and decrement operations are applied. A *variable automaton* (VA) is a type of constraint that encodes information about the value of a variable. VA transitions represent modifications to the values of the variables and the results of conditional tests of the values of those variables.

To reduce imprecision when checking the property in Figure 12, we build a VA for the counter variable `count`, as shown in Figure 13. In this VA, selected operations on a variable are interpreted precisely, for example, assignment of the constant zero and increment and decrement operations for values between zero and two. All other operations are safely abstracted to the situation where the variable's value is nonzero or unknown. Assigning zero to the value of the variable is represented by the event `count=0`. If the value to be assigned cannot be statically determined or is outside the range of values represented by the VA, then this is represented by the event `count=unknown`. Conditional results are represented by the possible outcomes (`is_count=0`, `is_count!=0`, and `is_count>1`). There are transitions to the violation state when the result of a conditional is inconsistent with the value of the variable described by the VA state. For example, the test `is_count=0` is false if the value of `count` is known to be 1, so there is a transition from 1 to the violation state on this event. The self-transition out of the violation state is labeled with a * to indicate that this transition is labeled with every possible event in the alphabet. The structure for other counter VAs is similar, although each will vary in size depending on the range of permissible values.

To use this VA, events modeling the behavior of `count` need to be added to the TFG, as shown in Figure 14. In this TFG, node 9 represents the initialization of variable `count` to 0, node 18 represents the increment of this variable, and nodes 14a and 14b represent the comparison of this variable to 2. Nodes 14a and 14b ensure that the true branch (false branch) is taken only when the condition is true (false).

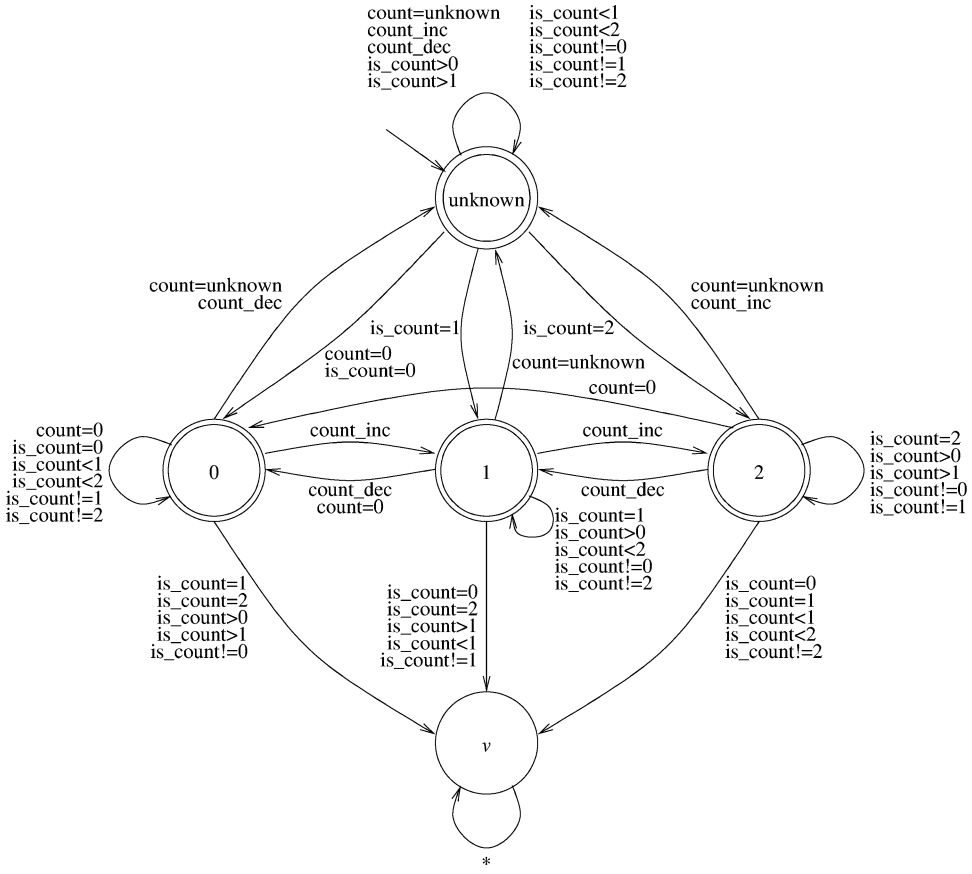

Fig. 13. VA for variable `count`.

Consider the path 20, 9, 21, 11, 22, 14a, 26, 19, 27 in the TFG in Figure 14 that corresponds to the infeasible path in the TFG in Figure 12 that we considered earlier. This path corresponds to the sequence of events `count=0`, `is_count=2`, `eat_card`. The subsequence `count=0`, `is_count=2` that contains events from the alphabet of the VA for `count` causes this VA to enter its violation state, which means that this path is infeasible, and, as explained in Section 6.3, this path will be discarded by FLAVERS during state propagation using this VA.

To enforce the conditions encoded in a VA during state propagation, the TFG alphabet must include the events in the VA alphabet and the TFG must be extended to identify system statements that cause variable state transitions. In many cases this could be done automatically, although the FLAVERS/Ada toolset currently requires that the events associated with a VA be added through the use of stylized comments in the source code, as mentioned previously.

The FLAVERS toolset allows users to select a VA definition, then automatically specializes that definition for a specific variable by instantiating a generic VA template and replacing the variable names in the template with the system variable name. Counter, enumerated, and Boolean VAs are constructed in

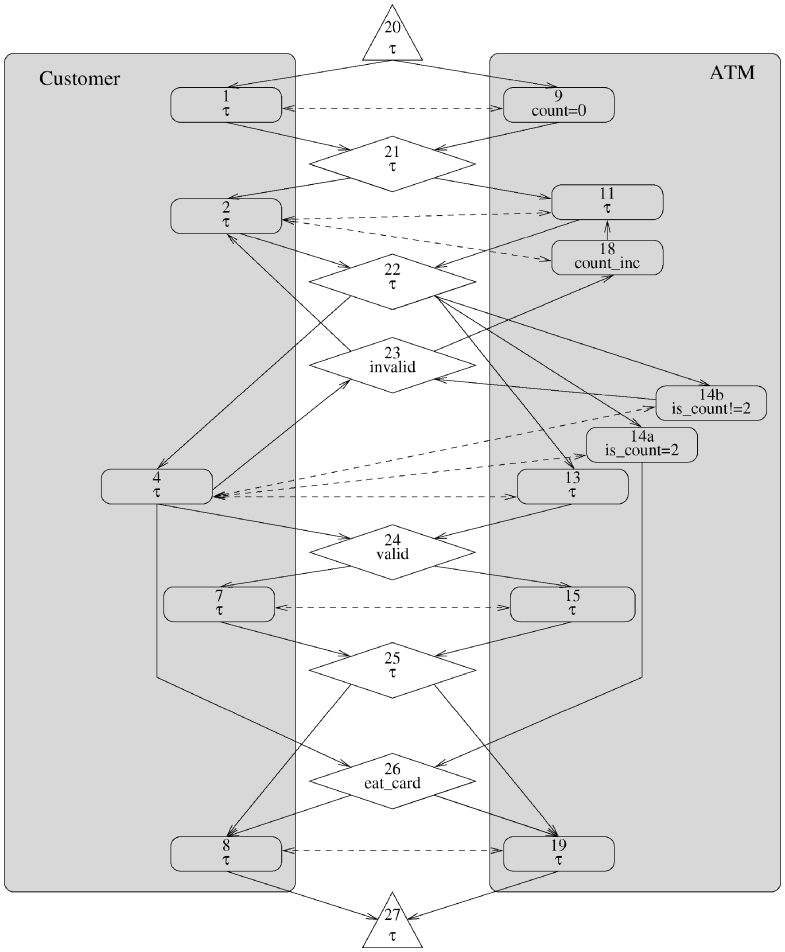

Fig. 14. TFG, for the example in Figure 4, refined for alphabet {`invalid`, `valid`, `eat_card`, `count=0`, `count_inc`, `is_count=2`, `is_count!=2`}.

$\mathcal{O}(k^2)$ steps, where k is the number of different possible values the variable can assume.

6.2 Task Automaton Constraints

Although MIP edges in the TFG are used to model the possible interleaving of events from different tasks succinctly, they have the negative side-effect of introducing paths that may violate event orderings that are encoded as control-flow edges in the TFG. For example, in Figure 14 the analysis may propagate dataflow information through the TFG nodes in the following order: 20, 9, 21, 2, 18, 2, 18, 11, 22, 14a, 26, 19, 27. This path violates the control flow in task `ATM` in several ways, for example, node 18 appears on this path before node 23 does, although the control flow through task `ATM` indicates that node 23 must precede node 18. Therefore, this path is infeasible and its removal would improve the

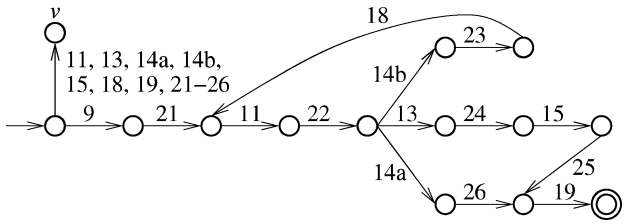

Fig. 15. TA for task ATM, built for the TFG in Figure 14. (For the violation state, all incoming transitions, except for those coming from the start state, are omitted.)

analysis precision, since the property in Figure 11 is violated on this path. Note that the VA for variable count is not violated on this path.

Infeasible paths that violate the control flow of a task are removed by encoding the control flow of that task as an FSA, called a *task automaton* (TA). During state propagation, this TA constrains the analysis to consider only those TFG paths that include events from that task in an order that is consistent with its intratask control flow.

To construct the TA for a task, we build the node-edge dual of the task's subflow-graph. Although we could build this from the RCFG directly, we use the TFG so any refinements that may have been applied to the TFG will be reflected in the task's subflow-graph as well.

We use function *Tasks* to return a set of tasks to which a TFG node belongs. For any local node, *Tasks* returns a set containing a single task and for any communication node, *Tasks* returns a set containing two tasks whose rendezvous is represented by this communication node. A TA is constructed for task T_i from the subflow-graph of the TFG with the set of nodes $\{n \mid n \in N \wedge T_i \in Tasks(n)\}$ and the set of edges in this subflow-graph that are incident upon these nodes. The initial and final nodes of the subflow-graph determine the start and accept states of the TA, respectively.

Figure 15 shows the TA for task ATM from the TFG in Figure 14. Each transition in this TA corresponds to executing code represented by a TFG node, and each state represents the task state before or after such execution. For clarity, we only show those transitions to the violation state that leave the start state of this automaton. For any node from task ATM, there is a transition based on this node from every state in the TA. All of the transitions that are not shown lead to the violation state.

Consider again the TFG path 20, 9, 21, 2, 18, 2, 18, 11, 22, 14a, 26, 19, 27. Since this path leads to the violation state of the TA in Figure 15 (specifically, a transition to the violation state is taken after transitions based on nodes 9, 21, and 18 are taken), FLAVERS will ignore this infeasible path during state propagation.

Note that, as described here, the definition of TAs is different from that of property automata and VAs in that the transitions in the TAs are based on the identities of TFG nodes instead of the labels of these nodes. Conceptually, we can label the TFG nodes with these unique node identifiers to be able to use the previously defined FSA model. In our implementation of FLAVERS, we redefine all FSA transitions in terms of TFG node identifiers for efficiency.

Construction of a TA requires $\mathcal{O}(|N_i|^2)$ steps, where $N_i \in N$ is the set of nodes in T_i. In the worst case, the subflow-graph for the task is fully-connected, therefore, we need to construct N_i transitions for each of the N_i states of the TA. Thus the entire algorithm is $\mathcal{O}(|N_i|^2)$. Constructing TAs is very fast in practice.

6.3 Combining the Property and Constraints in the State-propagation Analysis

To incorporate constraints into FLAVERS analyses, the state-propagation analysis is solved in conjunction with the analyses for each constraint. One obvious way to combine flow analyses for the property and all constraints is to form the product of the property and constraint automata, unifying all of the violation states, and then propagate states of this product automaton through the TFG. Here, we describe a more efficient approach that forms reachable product automata states on-the-fly [Naumovich et al. 1998].

Let $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$ be the property automaton and $\forall i, 1 \leq i \leq k, C_i = (\Sigma^{C_i}, S^{C_i}, \delta^{C_i}, A^{C_i}, s^{C_i}, v^{C_i})$ be the constraint automata used by the analysis. To handle these automata simultaneously, we define tuples of length $k+1$ that are associated with TFG nodes and represent the values of the property and constraints in these nodes. Formally, a *tuple* T is a collection of one state from each automaton in the problem:

$$T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k}), \text{ where } t^P \in S^P \text{ and } \forall i : 1 \leq i \leq k, t^{C_i} \in S^{C_i}.$$

Let *Tuples* be the set of all possible tuples:

$$\mathit{Tuples} = \left\{ (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k}) \mid t^P \in S^P \wedge \bigwedge_{i=1}^{k} (t^{C_i} \in S^{C_i}) \right\}.$$

The *initial tuple* is the tuple $T_0 = (s^P, s^{C_1}, s^{C_2}, \ldots, s^{C_k})$.

We associate a function f^n over tuples with each TFG node n:

$$\forall T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k}) \in \mathit{Tuples} : f^n(T) = (\overline{t^P}, \overline{t^{C_1}}, \overline{t^{C_2}}, \ldots, \overline{t^{C_k}}),$$

where

$$\overline{t^P} = \begin{cases} \delta^P(t^P, \mathit{label}(n)) & \text{if } \mathit{label}(n) \in \Sigma^P \\ t^P & \text{if } \mathit{label}(n) \notin \Sigma^P \end{cases}$$

$$\forall 1 \leq i \leq k, \overline{t^{C_i}} = \begin{cases} \delta^{C_i}(t^{C_i}, \mathit{label}(n)) & \text{if } \mathit{label}(n) \in \Sigma^{C_i} \\ t^{C_i} & \text{if } \mathit{label}(n) \notin \Sigma^{C_i} \end{cases}.$$

As in the state-propagation algorithm for property automata in Section 5, we generalize f^n to a function F^n over sets of tuples for each TFG node, making sure that tuples containing violation states of constraints are not produced. For every $n \in N$ and for every $X \subseteq \mathit{Tuples}$, we define $F^n(X)$ as follows:

$$F^n(X) = \left\{ f^n(T) = f^n(t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k}) \mid T \in X \wedge \bigwedge_{i=1}^{k} (t^{C_i} \neq v^{C_i}) \right\}.$$

The lattice elements for this dataflow problem are sets of tuples, and the join operation is set union. Once the solution of the dataflow problem over the TFG

converges to a fixed point, we look only at the final node of the TFG to determine whether the property holds. For each tuple associated with n_{final}, we check the constraint automata states to see whether all of these are accepting states. If any constraint automaton is left in a nonaccepting state, we remove the entire tuple from n_{final}. We then look at the property states on the remaining tuples as was done in the state-propagation algorithm in Figure 10.

This tuple-based propagation approach is reminiscent of K-tuple frameworks from Masticola et al. [1995]. An important distinction is that each component of a tuple in K-tuple frameworks corresponds to a special kind of edge in the graph. In our approach, an event associated with a TFG node can be present in alphabets of several automata, and thus components of tuples are not directly tied to disjoint sets of information in the flow graph.

The proof of conservativeness of state-propagation analysis using constraints appears in Appendix A.3.

THEOREM 6.2 (STATE-PROPAGATION COMPLEXITY WITH CONSTRAINTS). *Given a TFG G with nodes N^G, a property automaton P with states S^P, and constraint automata $C_1, \ldots, C_k$ with states $S^{C_1}, \ldots, S^{C_k}$, state propagation using the tuple-based approach has worst-case complexity that is $\mathcal{O}(|S^P||S^{C_1}| \cdots |S^{C_k}||N^G|^2)$.*

PROOF. This is proved in Naumovich et al. [1998]. Intuitively, if the property automaton and constraint automata are treated as FSAs, their cross-product could be computed, yielding a single FSA with at most $|S^P| \cdot |S^{C_1}| \cdots |S^{C_k}|$ states. This FSA could be modified into a property automaton and used in the state-propagation algorithm from Figure 10, with worst-case complexity $\mathcal{O}(|S^P||S^{C_1}| \cdots |S^{C_k}||N^G|^2)$. □

Using the VA for `count` and the TA for the task `ATM`, FLAVERS will still report that the property does not hold and present the analyst with a counterexample path. For inconclusive results, there are usually several counterexamples that could be shown, and the order that these are generated in depends on how the worklist is implemented in the state-propagation algorithm [Cobleigh et al. 2001]. For instance, FLAVERS might return the path 20, 1, 9, 21, 2, 11, 22, 4, 14b, 23, 2, 18, 11, 22, 4, 14b, 23, 2, 18, 11, 22, 4, 14a, 26, 8, 19, 27. This path corresponds to the event sequence `count=0`, `is_count!=2`, `invalid`, `count_inc`, `is_count!=2`, `invalid`, `count_inc`, `is_count=2`, `eat_card` and, indeed, is an example of an executable path that violates the property. For this example, the analyst would soon conclude that the property shown in Figure 11 does not hold on this program because of an off-by-one error. In the program, the variable `count` is initialized to zero and the card is eaten when the variable is equal to two, which occurs after an invalid PIN has been entered three times. By changing the if statement that corresponds to CFG node 14 in Figure 5 to read `if(count=1)`, then state propagation, using the VA for `count` and the TA for the task `ATM`, reports that the property in Figure 11 holds.

Alternatively, the counterexample path 20, 9, 21, 11, 22, 14b, 23, 18, 11, 22, 14b, 23, 18, 11, 22, 14a, 26, 19, 27 might be generated. This path results in the same event sequence, but is infeasible because it does not follow the control flow of task `Customer`. Often when observing an infeasible counterexample, the

analyst's attention is directed close enough to the source of the problem that the fault is still discovered. When this is not the case, the source of the infeasibility can be used to help select a constraint that will eliminate this infeasible path from further consideration. For the infeasible counterexample given above, the analyst would add a TA for task `Customer`.

7. EVALUATION OF FLAVERS

The complexity bounds of FLAVERS are polynomial in the number of program statements, but exponential in the number of constraints. These are worst-case bounds, however, and do not indicate the expected cost of analysis on typical applications. To develop an understanding of the feasibility of FLAVERS, we applied it to the analysis of a variety of problems, where a problem consists of a program, property, and set of constraints. In this section, we describe our methodology for applying FLAVERS to a set of problems, present the results of our empirical evaluation, and discuss a number of observations.

7.1 Methodology

Our empirical evaluation was performed using an implementation of FLAVERS that is targeted for Ada tasking programs. The FLAVERS/Ada toolset has been programmed in three different languages. The tools that construct CFGs from the Ada source code of the program being analyzed are built on top of Arcadia infrastructure components [Taylor et al. 1988] and are written in Ada. Tools written in Java convert these CFGs into a TFG and construct the constraint and property automata that are needed. Finally, the MHP and state-propagation analyses are written in C for efficiency.

We evaluated four scalable Ada tasking programs. Three of these, the Readers/Writers, Gas Station, and several variants of the Dining Philosopher program, have been used frequently as examples in the concurrency analysis literature (e.g., Avrunin et al. [1991]; Chamillard et al. [1996]; Corbett and Avrunin [1994]; Duri et al. [1993]; Dwyer [1995]; Masticola and Ryder [1993]; Young et al. [1995]). They were selected because they represent variations on topologies and synchronization structures that appear to be commonly used in concurrent programs. Readers/Writers and Gas Station represent variations on centralized resource management strategies, and the different Dining Philosopher programs represent different approaches to distributed resource management strategies. In addition, for each of these programs, correctness properties are easily identified. The fourth example, the Chiron program, is derived from a real system [Keller et al. 1991] and has been used as an example for comparing finite-state verification approaches [Avrunin et al. 1999]. For the Chiron program, we developed several of our own properties by reverse engineering the program. The goal was to come up with properties that developers would actually want to check for this program.

Since the version of FLAVERS/Ada that we used is designed to work with programs that terminate, the code for some of these programs needed to be modified slightly to ensure this. Additionally, to make use of VAs and to specify some properties, we included stylized comments in the source code, as

previously described, to indicate the relevant events to be associated with the source statements. The source code for all these programs can be found at `http://laser.cs.umass.edu/verification-examples/`.

These programs can be scaled by replicating some of the tasks. As is typically the case with finite-state verification techniques, properties involving such tasks are stated in terms of only some of these tasks, such as customer 1 and customer 2. Informal arguments are then made that if the property holds for an arbitrary selection of such tasks, it would hold for any such selection.

There are two important concerns in this evaluation: cost and precision. For evaluating the former, we consider the cost of constructing the analysis artifacts as well as the cost of performing the analysis. To isolate concerns about cost from concerns about precision, we chose properties that are known to be valid for the selected programs and measured only the performance of the analyses that yielded conclusive results.

For evaluating precision, we report on the constraints that were included to achieve a conclusive result. To find this set of constraints, we first tried to verify each property without any constraints. If the result was inconclusive, we examined the counterexample path returned by FLAVERS and added one or more constraints to remove the infeasible counterexample path from consideration and repeated the verification. We needed to repeat this process a small number of times, almost always less than three, to find a set of constraints sufficient for proving each property conclusively. Once we were able to prove each property conclusively, we began removing constraints until we found a minimal set of constraints, in the sense that removing one of the constraints from this minimal set would result in an inconclusive analysis result. The resulting set is not necessarily an optimal solution, where optimal means that it took the least amount of execution time (or space) to produce conclusive results. A more rigorous approach would be needed to determine the optimal set, since adding a constraint may reduce the execution time (or space). So even after a set of constraints sufficient to obtain conclusive results is found, additional constraints would need to be considered to determine an optimal set. Our goal was not to find the best performance, but to determine if there is a reasonable process that analysts might follow to find a minimal set of constraints. In following such a process, analysts would most likely not add additional constraints after finding a set that produced conclusive results.

To gain a sense of how the analysis would scale as the size of the system increases, we attempted to scale programs up to 200 replications. Note that each replication may involve adding one or more tasks to the system. For example, for the Dining Philosopher program both a philosopher and a fork must be added for each replication. Before scaling a program, we first considered a small, yet reasonable, instantiation of that program and then found a minimal configuration of constraints that led to conclusive results. We then used this configuration as we scaled the size of the program. Of course, there is no guarantee that this configuration will continue to produce conclusive results for a property (or the correspondingly scaled version of a property) when a program is scaled to a larger size. This was the case, however, for almost all the programs and properties that we considered; exceptions are noted below.

Although these experiments give the cost of running FLAVERS on a system with a minimal set of constraints, they do not give any indication of the analyst's time, including the time to develop the properties, to examine counterexamples to determine if they are feasible or infeasible, and to determine which constraints should be added to remove infeasible paths from consideration. Such an evaluation is beyond the scope of this article, but we provide anecdotal evidence when discussing the results of the evaluation.

Our primary measure of analysis time is the sum of user and system time as measured by `/usr/bin/time` on a Sun Enterprise 3500 with two 336MHz processors and 2GB of memory, running Solaris 2.6. Although this is a multiuser system, for all experiments we had exclusive access to the machine to reduce variance in the times. The Ada components of the FLAVERS/Ada tools were compiled using the Verdix Ada Development System version 6.2.3c, with optimizations disabled (to avoid known compiler bugs). The Java components were run using the Sun JDK version 1.3.0, with HotSpot. The C components were compiled with the Free Software Foundation's gcc version 2.95.2, using the -O2 flag for optimization.

When computing the execution time, we do not include the time for performing language processing, which would include syntactic and semantic analysis and the creation of an abstract syntax tree and annotated CFG. In our toolset, language processing is built upon an obsolete frontend and is not representative of expected frontend costs. Moreover, these costs are well understood and not of interest here. We include the cost of TFG construction, alphabet refinement, partial-order reduction, and state propagation.

To estimate the actual functional dependence between execution time and the size of the program, we fit both polynomial and exponential curves to the timing data. To evaluate the fit, we look at the mean square residuals, MS_{Res}, which is the amount of variability *not* explained by the model per-degree-of-freedom. When comparing fits, a smaller value for MS_{Res} represents a better fit. We also give the mean square regression, MS_{Reg}, which is the amount of variability explained by the model per-degree-of-freedom. MS_{Reg} is used to show the difference in scale between it and MS_{Res}. In our examples, MS_{Reg} is often orders of magnitude larger than MS_{Res}, indicating that the curves we are fitting to the data explain a large amount of the variance in the data. We supplement the fit data with plots that compare the fit data with the actual data.

To evaluate the space costs, we consider the size of the analysis artifacts compared to the size of the program. In our evaluation, LOC gives the number of lines of code for the Ada program, not counting blank lines and lines that contain only comments; Nodes and Edges are the number of nodes and edges of the TFG; Tuples is the number of different tuples created during state propagation; and Node-Tuples is the sum, over all nodes in the TFG, of the number of tuples associated with each node. Since the number of node-tuples corresponds to the number of states FLAVERS explored while performing state propagation, determining how the number of node-tuples grows with respect to program size gives a good indication of how memory usage by FLAVERS scales with respect to program size. To estimate the functional dependence between space and program size, we fit curves to the different size measure data.

The evaluation described here provides valuable insight into the FLAVERS approach. Since the sample size is small and not known to be representative, however, we do not claim that the results can be generalized to all Ada tasking programs.

7.2 Detailed Empirical Results

In this subsection, we describe each program and then discuss each property considered, the constraints used to produce conclusive results, and any interesting issues that arose. For each program, we then evaluate the performance of FLAVERS. The complete size and timing data for these evaluations are presented in Appendix B and Appendix C, respectively.

7.2.1 *The Readers/Writers Program.* The Readers/Writers program is a standard example that implements a means of thread-safe access to shared data. This program consists of a central data server, called the controller task, and a collection of reader and writer client tasks. It is scalable in the number of client tasks. In the program, readers only attempt to read and writers only attempt to write the shared data. The controller, however, can support clients that both read and write. The controller enforces exclusive-write semantics; if a writer is active, then no other writer or reader can be active. These semantics are encoded using local variables `WriterPresent` and `ActiveReaders`.

The state space of this program grows exponentially with the number of client tasks, where a state of the program records the states of the controller, readers, and writers. To evaluate how the cost of the FLAVERS analysis scales for this program, we began with two readers and two writers, and then increased the size of the program up to 200 readers and 200 writers. Assuming that a client task can only be in one of two states, there are 2^{r+w} reachable states for this program, where r is the number of readers, and w is the number of writers. Thus, if we can demonstrate polynomial growth for accurate analysis, we will have improved on naïve reachability analysis.

We specify properties in terms of the events `rw.control.start_write`, `rw.control.stop_write`, `rw.control.start_read`, and `rw.control.stop_read`, which are the fully-qualified Ada names of the controller entries, corresponding to a writer task starting and stopping writing and a reader task starting and stopping reading, respectively. We use the abbreviations `wstart`, `wstop`, `rstart`, and `rstop`, respectively, to make the QREs more readable.

Exclusive Read Write Property. The Exclusive Read Write Property checks that when a writer is active, no other reader or writer can become active. The QRE for this property is:

```
for events {wstart, wstop, rstart, rstop}
show all executions satisfy
(~[wstart]*; wstart; ~[wstart, rstart, wstop]*; wstop)*;
(wstart; ~[wstart, rstart, wstop]*)?
```

The first line of the regular expression portion of the QRE states that between the time a writer starts writing and the time it stops, no other writer or reader

can start. Note that in this property we explicitly list all the events that cannot occur after a writer starts, instead of just listing the one event that could occur, namely `rstop`. We prefer this style since the regular expression portion does not have to be changed if additional events are added to the property alphabet, as often happens when adding VA constraints. The second line represents the case where a writer starts writing and does not stop. Although this is behavior that would be unexpected in the example, we are trying to capture the most general form of the property. This property was proved conclusively using two constraints: a TA for the controller and a VA for the Boolean `WriterPresent` variable that keeps track of whether a writer task is currently writing. We used FLAVERS to prove this property for versions of this example with up to 200 readers and 200 writers. The analysis of the largest version required 29 seconds.

No Write While Reading Property. Although the **exclusive read write** property excludes the possibility of a write or a read being initiated while a writer is active, it does not preclude a reader from being active when a writer starts. A separate property can be used to check for this. Since multiple readers can be active simultaneously, this property needs to count the number of active readers. As a result, this property needs to be scaled with the number of readers. For the 2 readers case, the QRE for this property is:

```
for events {wstart, rstart, rstop}
show all executions satisfy
(
  (rstart;
    (rstart; rstop)*;
  rstop)*;
wstart*)*
```

For the 3 readers case, the QRE for this property is:

```
for events {wstart, rstart, rstop}
show all executions satisfy
(
  (rstart;
    (rstart;
      (rstart; rstop)*;
    rstop)*;
  rstop)*;
wstart*)*
```

This QRE keeps track of the number of readers that have started but not yet stopped and ensures that a writer can only start while there are no active readers. To prove this property conclusively, FLAVERS needed 2 constraints, a TA for the controller and a VA for the integer counter variable `ActiveReaders` that keeps track of the number of active readers. Since the variable `ActiveReaders` can grow up to the number of readers in the example, the size of this automaton needs to be scaled with the size of the program. FLAVERS was able to prove

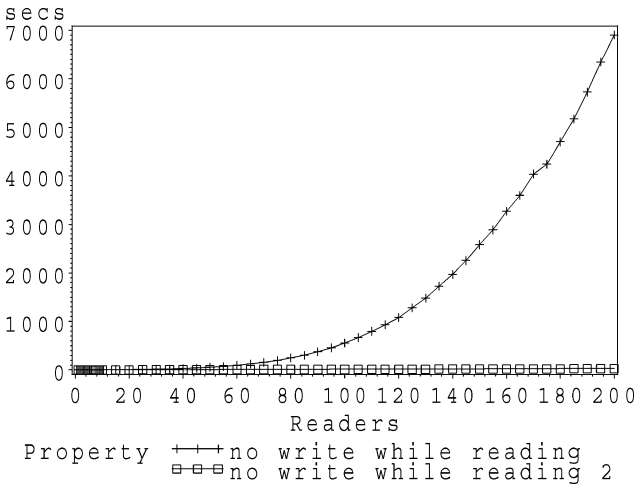

Fig. 16. **no write while reading** comparison.

that this property holds with 200 readers and 200 writers, but it required 6,901 seconds, a significant increase over the cost of proving the **exclusive read write** property, which did not require the property or constraints to scale as the program size was scaled.

We can take advantage of variable `ActiveReaders` to derive the following alternative expression of the property, **no write while reading 2**. The QRE for this property ensures that a write can only start when the value of the variable `ActiveReaders` is zero:

```
for events {r=0, is_r=0, r_inc, r_dec, wstart}
show all executions satisfy
(~[wstart, r=0, is_r=0] |
  ([r=0, is_r=0]; ([r=0, is_r=0] | wstart)*;
   ~[wstart, r=0, is_r=0])
)*; ([r=0, is_r=0]; ([r=0, is_r=0] | wstart)*)?
```

In this QRE, `r=0` represents an assignment of 0 to the variable `ActiveReaders`, `is_r=0` represents a check that the value is 0, `r_inc` and `r_dec` represent the incrementing and decrementing of this variable. In the program, these are the only events that access or change the value of `ActiveReaders`. The property checks that either `r=0` or `is_r=0` must precede a writer starting without an intervening change in the variable's value. This property does consider some event sequences as violations that are legal, for example, `r=0`, `r_inc`, `r_dec`, `wstart`. Still, if this property is conclusive, then we can conclude that no writer can start while a reader is reading. To prove this property conclusively, FLAVERS only needed a TA for the controller and proved this property on an example with 200 readers and 200 writers in about 30 seconds. Figure 16 shows the timing comparisons of the two formulations of this property.

Write First Property. Another property we consider is that data must be in the buffer before a read is attempted. The QRE for this property, **write**

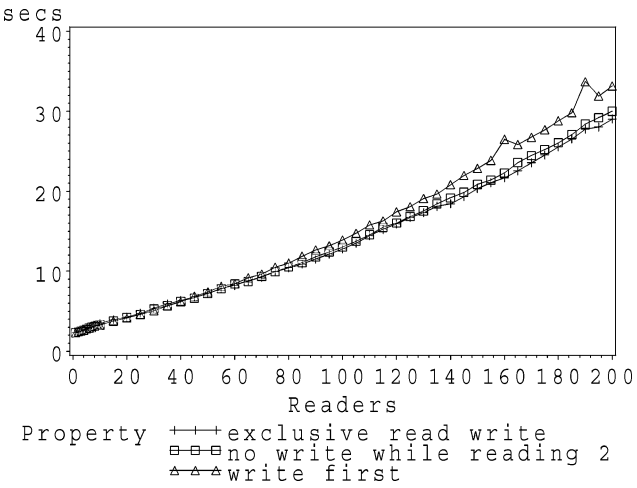

Fig. 17. Reader/Writer total times.

first, is:

```
for events {wstart, rstart} show all executions satisfy
(~[wstart, rstart]*; wstart; .*) | ~[rstart]*
```

The first clause of the regular expression ensures that a reader does not start until after a writer starts. The second clause says that if no writer starts throughout a program run, then no reader can start on this run. To prove this property conclusively FLAVERS used a TA for the controller and was able to handle an example with 200 readers and 200 writers in about 33 seconds.

Performance. Figure 17 shows execution time versus the number of readers in the example for all of the properties except for **no write while reading**, which was shown in Figure 16.

To estimate the actual functional dependence between execution time and the number of readers and writers, we fit different curves to the time needed to verify the four properties. Since **no write while reading** required significantly more time than the others, we only provide data for this property. The results of these fittings are shown in Table I. Each column gives the data for the coefficients of the best-fit curve of the given form.[5] As is expected, for the three polynomial fits MS_{Res} decreases as the degree of the polynomial increases. Note, MS_{Res} for the exponential line is larger than MS_{Res} for the cubic polynomial. When comparing fits, a smaller value for MS_{Res} represents a better fit. One thing that is important to notice in this and all future tables is that the mean squared residuals are orders of magnitude smaller than the mean squared regressions, meaning the curves we are attempting to fit to the data do a good job of explaining the data. Figure 18 plots the cubic polynomial against the exponential, and the cubic fit looks quite accurate, as expected from the fit data.

[5]We also looked at curves of the form $e^{c_1 x}$, but in all cases this curve was a very poor fit so we do not include it here.

Table I. Curve Fitting for **no write while reading**

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2x^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$8.9763 * 10^5$	$4.7369 * 10^4$	$1.5484 * 10^3$	$3.2612 * 10^4$
MS_{Reg}	$1.3452 * 10^8$	$8.6837 * 10^7$	$5.8579 * 10^7$	$1.3043 * 10^8$
c_0	$-9.1272 * 10^2$	$2.7825 * 10^2$	-6.3646	4.3524
c_1	$2.6152 * 10^1$	$-2.4050 * 10^1$	2.4618	0.0227
c_2		$2.7208 * 10^{-1}$	$-9.3607 * 10^{-2}$	
c_3			$1.2660 * 10^{-3}$	

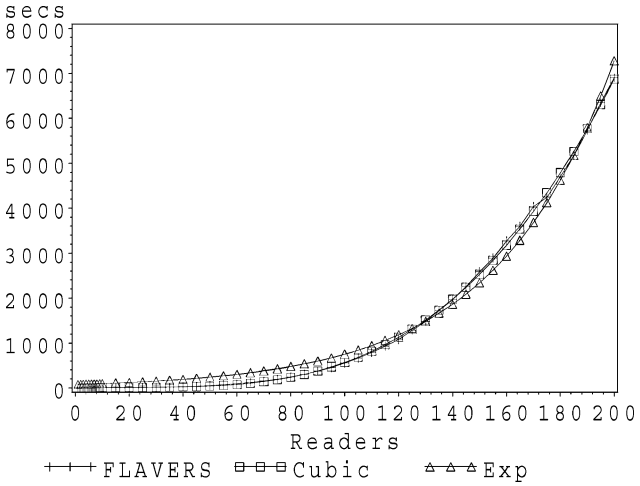

Fig. 18. Fit comparison for **no write while reading**.

For property **no write while reading**, Table II gives the size information for some of the artifacts created during the analysis, for a sampling of program sizes. For this property, the numbers in these columns fit exactly to a polynomial, as shown in the last row of Table II. The growth of the number of lines of code, nodes, and edges is linear in the number of readers, whereas the growth of the number of tuples and node-tuples is quadratic. The fact that the number of node-tuples grows quadratically is particularly important because the number of node-tuples corresponds closely with the amount of memory used for running an analysis. Thus, FLAVERS is showing polynomial growth both in terms of space and time on this property.

7.2.2 *The Gas Station Program.* The Gas Station program is a simulation of an automated self-serve gas station [Helmbold and Luckham 1985]. The gas station consists of a collection of server tasks, or pumps, an operator task, and a collection of client tasks, or customers. It is similar to the Readers/Writers program in that both have a server with client tasks. In the Readers/Writers program, the server is the control task. In the Gas Station program, the server is a scalable collection of cooperating tasks. Thus, the Gas Station program is scalable in two different dimensions, size of the server and number of clients.

Table II. Size Information for **no write while reading**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	25	50	16	47
50	1,153	711	1,912	7,905	23,959
100	2,253	1,411	3,812	30,805	92,909
150	3,353	2,111	5,712	68,705	206,859
200	4,453	2,811	7,612	121,605	365,809
n	$22n + 53$	$14n + 11$	$38n + 12$	$3n^2 + 8n + 5$	$9n^2 + 29n + 9$

The server subsystem consists of an operator task that accepts payments and gives change to customers, and a number of pump tasks that independently start and stop the pumping of gas. The operator interacts with the pumps by enabling them to pump gas after payment has been received and by getting information about how much gas was pumped. The client tasks pay for gas, pump it, and get their change. Instead of an explicit queue to represent the clients that are waiting to pump gas, the customers block on a rendezvous until the pump is available.

We looked at programs with 1, 2, and 3 pumps and attempted to scale the number of customers, up to 200, for all properties. We did not look at systems with more than 4 pumps because, as mentioned, the language processing tools are built on an obsolete frontend and the cost for running them became onerous.

Mutual Exclusion Property. The Gas Station program should only allow one customer to use a pump at a time. We check this property for customers 1 and 2, on pump 1. Since all the customer and pump tasks are identical and none reference task ids, we argue that verifying this property for these specific customers and pump is equivalent to verifying it for any arbitrary pair of customers and any particular pump. This property is specified in terms of the events indicating that customers 1 and 2 start and stop pumping using pump 1: `cust_1_start_pump_1`, `cust_1_stop_pump_1`, `cust_2_start_pump_1`, and `cust_2_stop_pump_1`. We abbreviate these by `c1_start`, `c1_stop`, `c2_start`, and `c2_stop` to make the QREs more readable. The QRE for the **mutual exclusion** property is:

```
for events {c1_start, c2_start, c1_stop, c2_stop}
show all executions satisfy
( (c1_start ; c1_stop) | (c2_start ; c2_stop) )*
```

This QRE ensures that either customer 1 starts pumping and then stops pumping, or customer 2 starts pumping and then stops pumping. To prove this property, FLAVERS needs a TA for pump 1 and for each customer. Thus, there are $c + 1$ constraints, where c is the number of customers. Because of the number of constraints, the analysis of this property scaled very poorly. As shown in Figure 19, FLAVERS could only prove this property on an example with 1 pump and 7 customers, with 2 pumps and 5 customers, and with 3 pumps and 4 customers; larger sizes ran out of memory.

As with the Reader/Writer program, we looked for an alternative way to express this property. By focusing just on an individual customer and pump,

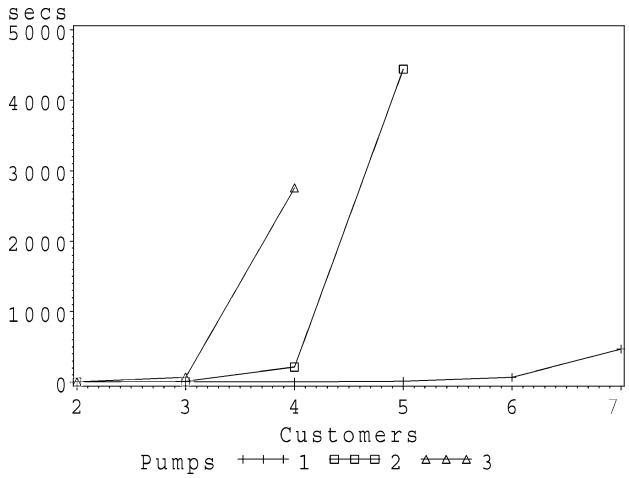

Fig. 19. **mutual exclusion** times.

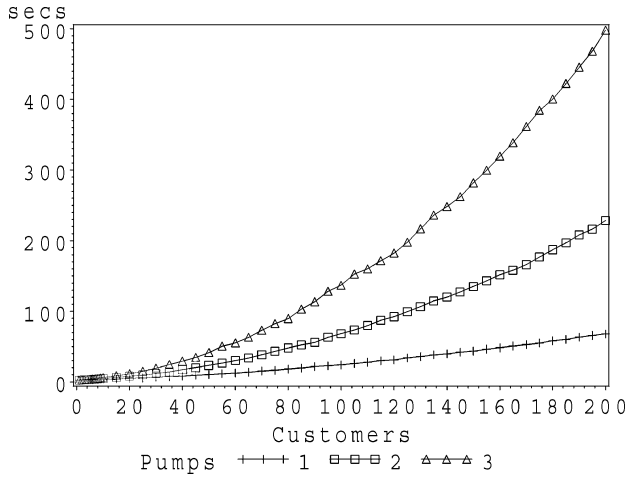

Fig. 20. **customer 1 start/stop** times.

say customer 1 and pump 1, we can create two simpler properties **customer 1 start/stop** and **pump 1 start/stop** that can be used together to prove the original **mutual exclusion** property.

The **customer 1 start/stop** property states that customer 1 starts pumping and then stops pumping. The QRE for this is:

```
for events {c1_start, c1_stop} show all executions satisfy
(c1_start ; c1_stop)*
```

To prove this property conclusively, FLAVERS did not need any constraints. FLAVERS was able to prove this property with 200 customers and 3 pumps as shown in Figure 20. Since all customers tasks are identical, we can use the proof that customer 1 obeys the **customer 1 start/stop** property to argue that customer i also starts pumping and then stops pumping.

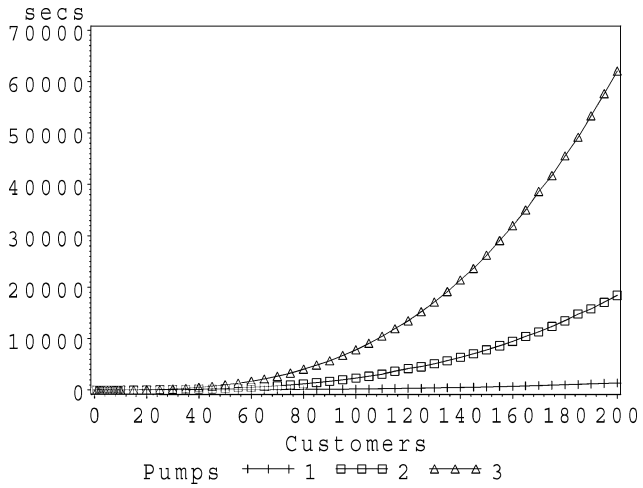

Fig. 21. **pump 1 start/stop** times.

The QRE for the second property, **pump 1 start/stop**, is:

```
for events {c1_start, c2_start,..., cn_start,
            c1_stop, c2_stop,..., cn_stop}
show all executions satisfy
( [c1_start, c2_start,..., cn_start];
  [c1_stop, c2_stop,..., cn_stop] )*
```

This property states that a start event for some customer is always followed by a stop event for some customer, although not necessarily the same customer. To prove this property conclusively, FLAVERS needs a TA for pump 1. As shown in Figure 21, FLAVERS proved this property on the gas station example with 3 pumps and 200 customers.

Now we need to show that proving these two properties is sufficient to prove the **mutual exclusion** property. The alphabets of properties **customer 1 start/stop**, **pump 1 start/stop**, and **mutual exclusion** are not the same. Note, however, that for any property, we can add an event to its alphabet and not change its semantics by adding self-loop transitions for that new event to each of the property states. Let P_{ME} represent the property **mutual exclusion**, P_{C1} the property **customer 1 start/stop**, P_{P1} the property **pump 1 start/stop**, and P_{C2} the **customer 2 start/stop** property, all extended so that their alphabets are the same.

Now, let ρ be a path that represents a feasible execution of the Gas Station program. Since P_{C1}, P_{C2}, and P_{P1} were all proven conclusively, then $\rho \in \mathcal{L}(P_{C1})$, $\rho \in \mathcal{L}(P_{C2})$, and $\rho \in \mathcal{L}(P_{P1})$. Therefore, $\rho \in \mathcal{L}(P_{C1}) \cap \mathcal{L}(P_{C2}) \cap \mathcal{L}(P_{P1})$. Now, it can be shown that $\mathcal{L}(P_{C1}) \cap \mathcal{L}(P_{C2}) \cap \mathcal{L}(P_{P1}) \subseteq \mathcal{L}(P_{ME})$. Thus, since every feasible path is contained in the language of P_{ME}, we can conclude that the **mutual exclusion** property holds for the Gas Station program.

Correct Change Property. Another desirable property is that if a customer prepays on a given pump, then that customer receives the change for that

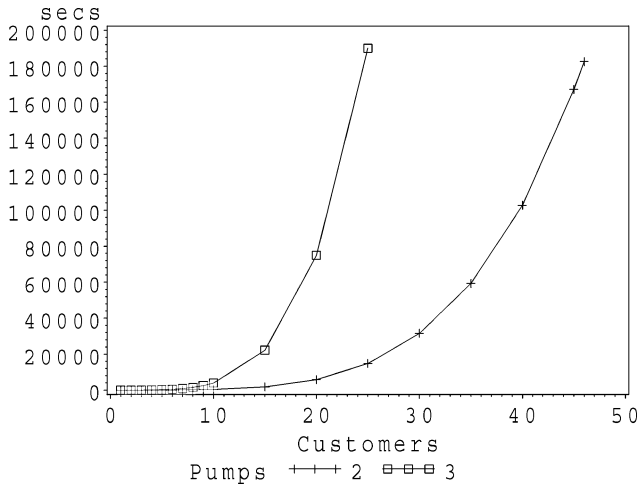

Fig. 22. **correct change** times.

Table III. Size Information for **correct change** with 3 Pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	140	187	567
2	316	131	738	2,638	30,946
5	673	296	5,268	25,117	1,395,485
10	1,268	571	21,938	112,892	12,494,400
15	1,863	846	50,008	263,667	43,692,665
20	2,458	1,121	89,478	477,442	105,385,280
25	3,053	1,396	140,348	754,217	207,967,245
$n > 2$	$119n + 78$	$55n + 21$	$228n^2 + -86n - 2$	$1260n^2 + -1345n + 342$	$13860n^3 - 13813n^2 + 1478n + 920$

pump. The following QRE expresses this property for customer 1 and pump 1:

```
for events {c1_change_p1, c1_change_p2,..., c1_change_pn,
            c1_prepay_p1}
show all executions satisfy
( ( c1_prepay_p1 ; c1_change_p1 ) |
  [c1_change_p2,..., c1_change_pn] )*
```

This QRE ensures that if customer 1 prepays on pump 1, then customer 1 receives change on pump 1. To prove the property conclusively, FLAVERS needs TAs for the operator, customer 1, and pump 1. Figure 22 shows FLAVERS' performance on this property. In this case, FLAVERS was only able to prove the property with 2 pumps and 46 customers, and with 3 pumps and 25 customers; with more customers FLAVERS ran out of memory. Table III shows the size information for this property with 3 pumps.

Performance. Of the gas station properties where FLAVERS could handle programs that scale to 200 customers, it takes the most time to verify the **pump 1 start/stop** property, thus we show the curve fittings for this property. Table IV shows the results of these fittings. With this property, MS_{Res} is better

Table IV. Curve Fitting for **pump 1 start/stop** with 3 Pumps

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$5.9292 * 10^7$	$1.6652 * 10^6$	$5.4350 * 10^3$	$3.1968 * 10^6$
MS_{Reg}	$1.2483 * 10^{10}$	$7.5678 * 10^9$	$5.0701 * 10^9$	$1.2010 * 10^{10}$
c_0	$-8.0699 * 10^3$	$1.7321 * 10^3$	$1.9707 * 10^1$	7.0802
c_1	$2.5193 * 10^2$	$-1.6125 * 10^2$	-1.7397	0.0201
c_2		2.2393	$3.9162 * 10^{-2}$	
c_3			$7.6166 * 10^{-3}$	

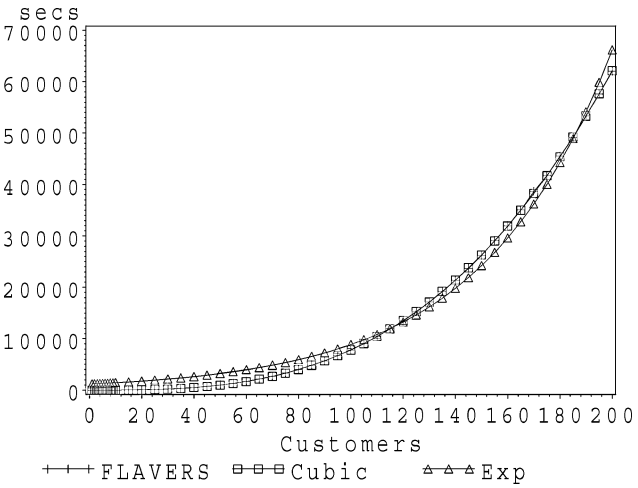

Fig. 23. Fit comparison for **pump 1 start/stop** with 3 pumps.

for the exponential curve than the linear polynomial, but the cubic fit is best, as can also be seen in the plot shown in Figure 23.

Table V gives size information for a sampling of program sizes. For this property, these sizes fit exactly to a polynomial, as shown in the last row of the table, when the number of customers is 2 or more. For these problems, the number of LOC, nodes, and tuples grew linearly with the number of customers, and the number of edges and node-tuples grew quadratically.

There were two gas station properties where FLAVERS was not able to scale up to 200 customers, the **mutual exclusion** property and the **correct change** property. On the **mutual exclusion** property, the number of constraints needed to be increased as the number of customers increased, resulting in FLAVERS running out of memory. Unfortunately, there are not enough data points for this property to do meaningful curve fitting. Still, by verifying three other properties, we were able to prove that the **mutual exclusion** property holds on a system with 200 customers. For the **correct change** property, the number of constraints did not need to be increased with the number of customers, but FLAVERS was still unable to verify this property on a system with 200 customers. In this case, the number of node-tuples grew as a cubic polynomial of the number of customers, but the cubic coefficient was large, so the memory

Table V. Size Information for **pump 1 start/stop** with 3 Pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	136	10	292
2	316	131	878	17	1,580
50	6,028	2,771	669,518	353	761,180
100	11,978	5,521	2,689,018	703	3,027,330
150	17,928	8,271	6,058,518	1,053	6,798,480
200	23,878	11,021	10,778,018	1,403	12,074,630
$n > 1$	$119n + 78$	$55n + 21$	$270n^2 - 110n + 18$	$7n + 3$	$301n^2 + 173n + 30$

requirements grew quickly. We are unsure as to why the number of node-tuples grew so quickly for the **correct change** property.

7.2.3 *The Dining Philosophers Program.* The Dining Philosophers program consists of equal numbers of philosophers and forks. These are organized into a ring with alternating philosopher and fork tasks. Each philosopher has access to two forks, left and right, and thus each fork is shared by two philosophers. A philosopher attempts to gain access to both of its incident forks. A philosopher that has both forks will proceed to eat. Between the time a philosopher puts down the forks and makes an attempt to pick up the forks again, the philosopher is thinking. This program has a potential deadlock where each philosopher has picked up the left (right) fork and is waiting for the right (left) fork to become available.

We verified properties for four different versions of the Dining Philosopher program: standard, dictionary, fork manager, and host. The standard version prevents the potential deadlock by having each philosopher pick up the left fork, and then the right fork, except for one philosopher, who reverses this order. In the standard version, each philosopher and fork is a task, so a program with n philosophers has $2n$ tasks. The dictionary version adds a token, the dictionary, that prevents its holder from picking up a fork and so it also has $2n$ tasks when there are n philosophers. When the left neighbor is thinking, the philosopher can pass the dictionary to that neighbor. The fork manager version uses a manager task to control access to the forks and ensures that philosophers pick up both forks in an atomic action. In this version, the manager task controls access to all of the forks, and so there is only one fork task, and thus $n + 1$ tasks when there are n philosphers. The host version uses a control task to ensure that at most $n - 1$ philosophers can be attempting to pick up the forks at the same time. In this version, each philosopher and each fork is a task, and there is one additional task for the control task, and so the host version with n philosophers has $2n + 1$ tasks. We checked the dining philosopher properties described below on each version of the Dining Philosopher program, scaling the number of philosopher and fork tasks up to 200 each.

Mutual Exclusion Property. To check the property that "adjacent philosophers cannot eat concurrently," we need to choose two representative adjacent philosophers, say philosophers 1 and 2. In the QRE representation of the property, we use `p1start` and `p1stop` to represent the events of philosopher 1 starting eating, and stopping eating and use `p2start` and `p2stop` for the same actions

Table VI. Constraints for the Dining Philosopher Programs

Example	**mutual exclusion**	**no fork 1 up twice**
Standard	TA: phil_1, phil_2, fork_2	TA: fork_1
Fork Manager	TA: manager, phil_1, phil_2; VA: fork_2	TA: manager
Dictionary	TA: phil_1, phil_2, fork_2	TA: fork_1
Host	TA: phil_1, phil_2, fork_2	TA: fork_1

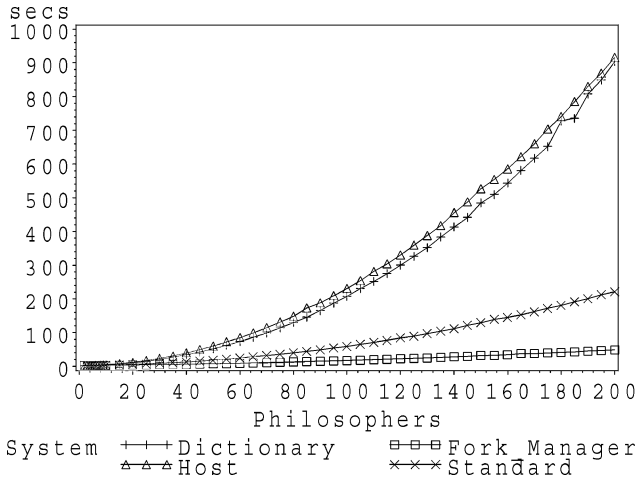

Fig. 24. **mutual exclusion** total times.

of philosopher 2. The QRE for the **mutual exclusion** property is:

```
for events {p1start, p1stop, p2start, p2stop}
show all executions satisfy
( ( p1start ; p1stop ) | ( p2start ; p2stop ) )*
```

This QRE is similar to the **mutual exclusion** property for the Gas Station program, in that it ensures that once philosopher 1 (or 2) starts eating, then philosopher 2 (or 1) cannot eat. FLAVERS is able to prove this property on all four versions of the Dining Philosopher program using the constraints shown in Table VI. The times necessary to verify this property are shown in Figure 24.

No Fork Up Twice Property. To check that a fork cannot be picked up twice in a row without first being put down, we again need to represent this property in terms of an arbitrary fork instance. The events `up` and `down` represent the fork being picked up and put down, respectively. The QRE to check this property on fork 1 is:

```
for events {up, down} show all executions satisfy
(~[up]*; up; ~[up,down]*; down)*; ~[up]*; (up; ~[up,down]*)?
```

The first clause of the regular expression part of this property ensures that once an `up` for fork 1 occurs, no `up`s will be seen until a `down` for this fork occurs. The last clause deals with the case where the execution might terminate with an `up` that is not followed by a `down`. It ensures that two `up`s are not seen

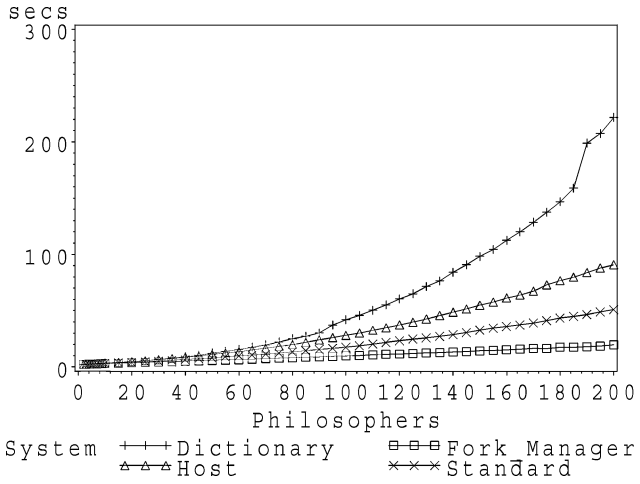

Fig. 25. **no fork up twice** times.

Table VII. Curve Fitting for **mutual exclusion** on the Host Version

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$5.8657 * 10^3$	5.038	5.069	$1.8156 * 10^3$
MS_{Reg}	$3.4409 * 10^6$	$1.8523 * 10^6$	$1.2349 * 10^6$	$3.5543 * 10^6$
c_0	$-1.0317 * 10^2$	2.2300	2.6553	3.8761
c_1	4.2639	$2.5399 * 10^{-2}$	$-1.1591 * 10^{-2}$	0.0152
c_2		$2.2734 * 10^{-2}$	$2.3237 * 10^{-2}$	
c_3			$-1.731 * 10^{-6}$	

consecutively if this situation occurs. FLAVERS is able to prove this property on all four versions using the constraints shown in Table VI. Figure 25 shows FLAVERS' performance on this set of problems.

Performance. Of the four versions of the Dining Philosopher program, FLAVERS takes the most time to verify the **mutual exclusion** property on the host version. Table VII shows the results of curve fitting for timing results for this version. For this property, the quadratic appears to be the best fit, as can be seen in the plot shown in Figure 26.

Table VIII gives the size information for a sampling of program sizes. For the **mutual exclusion** property and a program with at least three philosophers, these sizes fit exactly to a polynomial, as shown in the last row of the table. For this property, the number of tuples was a constant; the number of LOC, edges, and node-tuples grew linearly with the number of philosophers, and the number of edges grew quadratically.

7.2.4 *The Chiron Problem.* Chiron is a user-interface framework developed at the University of California at Irvine [Keller et al. 1991]. Since it was not developed as an analysis test case, it may be more representative of "typical" concurrent Ada programs than some of the other programs we considered.

In an instantiation of Chiron, there are one or more artist tasks that draw on the screen. There are also a number of events, for example button pushes,

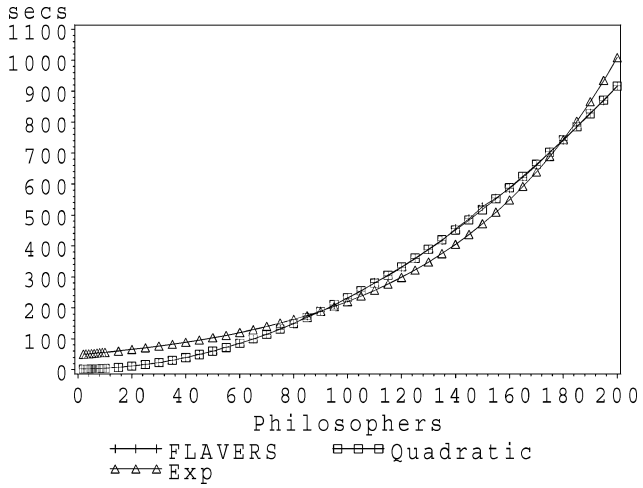

Fig. 26. Fit comparison for **mutual exclusion** on the Host version.

Table VIII. Size Information for **mutual exclusion** on the Host Version

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	87	44	132	87	445
3	114	62	305	87	2,118
50	1,383	908	81,662	87	66,884
100	2,733	1,808	323,412	87	135,784
150	4,083	2,708	725,162	87	204,684
200	5,433	3,608	1,286,912	87	273,584
$n > 2$	$27n + 33$	$18n + 8$	$32n^2 + 35n - 88$	87	$1378n - 2016$

that may be of interest to artists. To listen for events, artists register with a central server called the dispatcher. The dispatcher maintains a list of artists registered for each event, and when it receives an event, it passes it on to all artists registered for that event. An alternative to a centralized dispatcher is to have dedicated dispatchers for each event. This alternative "decomposed dispatcher" version of Chiron was created and also considered in our evaluation.

We checked properties of the Chiron program with a centralized dispatcher and two artists, and scaled up the number of events for which artists could register. The two artists were not identical—only `artist1` registered for one third of the events, only `artist2` registered for another third, and both artists registered for the remaining one third of the events.

We checked nine properties on this Chiron program. The description of the properties and the constraints needed to prove each are shown in Table IX. In Chiron, the dispatcher uses an array for each event to keep track of which artists are registered for that event. A VA called "event$i$ list size" keeps track of the number of spaces in the array that are used, that is, the number of artists that are registered for event i. A VA called "event$i$ list slot j" keeps track of the data that is stored in the jth position of the array that stores the artists that are registered for event i.

Table IX. Chiron Properties

Property	Description	Constraints
p01	`artist1` never registers for `event1` if it is already registered for this event, and `artist1` never unregisters for `event1` if it is not registered for this event.	TA: artist1
p02	If `artist1` is registered for `event1` and the dispatcher receives `event1`, then the dispatcher will not accept another event before passing `event1` to `artist1`.	TA: dispatcher VA: event1 list size VA: event1 list slot 1 VA: event1 list slot 2
p03	The dispatcher does not notify any artists of `event1` until it receives an `event1`.	TA: dispatcher
p04	Having received `event1`, the dispatcher never notifies artists of `event2`.	TA: dispatcher
p05	If no artists are registered for `event1`, the dispatcher does not notify any artist of `event1`.	TA: artist1 TA: artist2
p06	The dispatcher never gives `event1` to `artist1` if `artist1` is not registered for `event1`.	TA: artist1
p07	If `artist1` registers for `event1` before `artist2` does, then when the dispatcher receives `event1` it will first notify `artist1` and then `artist2` of this event.	TA: dispatcher VA: event1 list size VA: event1 list slot 1
p08	The size of the list used to store the IDs of artists registered for `event1` never exceeds the number of existing artists.	TA: dispatcher VA: event1 list size VA: event1 list slot 1 VA: event1 list slot 2
p09	The program does not terminate while there is an artist that is registered for an event.	TA: artist1 TA: artist2

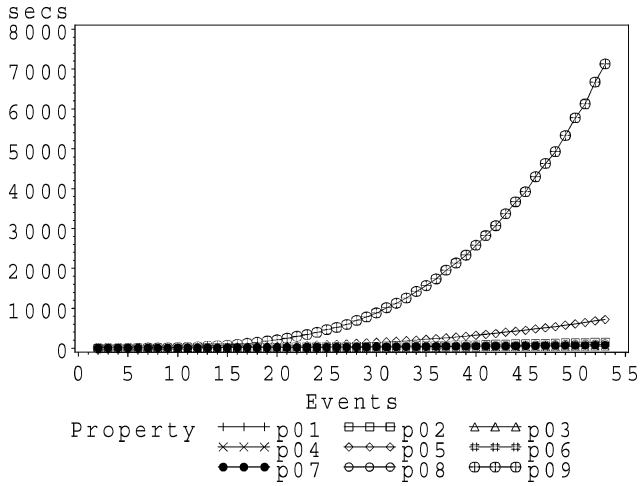

Fig. 27. Chiron total times.

Timing information for all of the Chiron properties are shown in Figure 27. Unfortunately, we could only scale the program up to 53 events, because of limitations with the language processing tools that we used. FLAVERS verified all of the nine properties, up to 53 events.

Table X. Curve Fitting for **p09**

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$8.7971 * 10^5$	$4.8461 * 10^4$	$4.221 * 10^2$	$2.9107 * 10^4$
MS_{Reg}	$1.6812 * 10^8$	$1.0487 * 10^8$	$7.0696 * 10^7$	$1.7514 * 10^8$
c_0	$-1.6561 * 10^3$	$7.0291 * 10^2$	$-7.2032 * 10^1$	4.2851
c_1	$1.1981 * 10^2$	$-1.2453 * 10^2$	$2.5074 * 10^1$	0.0875
c_2		4.4426	−2.1808	
c_3			$8.0284 * 10^{-2}$	

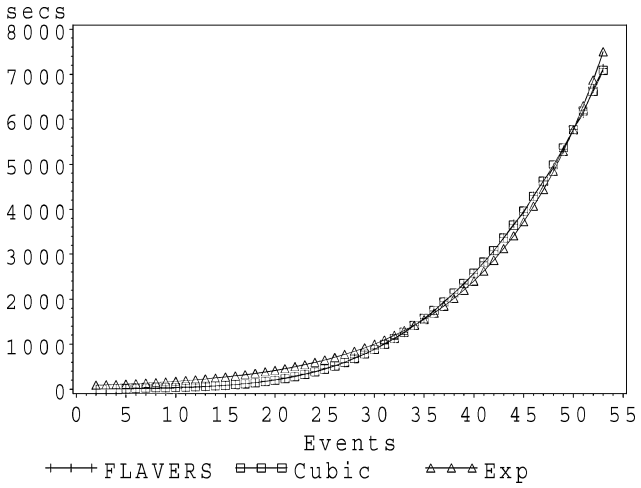

Fig. 28. Fit comparison for **p09**.

Performance. Of the Chiron properties, it takes the most time to verify property **p09**, and thus, in Table X, we show the curve fittings for only this property. For this property, MS_{Res} is best for the cubic polynomial. This can be seen in the plot shown in Figure 28.

Table XI gives the size information for a sampling of program sizes. Unlike the previous properties, these sizes do not fit exactly into polynomials, even when some of the smaller-sized programs are not considered. Thus, we performed curve fitting on the size metrics shown in Table XI. We found that the best-fit curve for the number of LOC and nodes is linear, for the number of edges and tuples it is quadratic, and for the number of node-tuples it is cubic. Since the amount of space used for an analysis is closely related to the number of node-tuples, we show the curve-fitting data for this metric in Table XII and Figure 29.

7.2.5 *Tool Comparison.* The study reported in Avrunin et al. [1999] used the Chiron program to compare different finite-state verification techniques across a range of properties and for different sizes of systems. The tools considered included both explicit state and symbolic model checkers, SPIN [Holzmann 1997] and SMV [McMillan 1993], respectively; an integer programming based analysis, INCA [Corbett and Avrunin 1995]; and FLAVERS.

Table XI. Size Information for **p09**

Events	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	374	559	5,984
10	2,393	314	2,876	9,101	250,542
20	4,690	582	8,894	33,451	641,572
30	6,987	850	17,782	73,126	5,132,994
40	9,286	1,124	30,476	130,001	11,957,892
50	11,583	1,392	46,134	200,751	22,799,552
53	12,274	1,473	51,442	225,214	27,030,782

Table XII. Curve Fitting for **p09** Node-Tuples

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$9.1562 * 10^{12}$	$2.1581 * 10^{11}$	$2.7322 * 10^{9}$	$6.3260 * 10^{11}$
MS_{Reg}	$2.7918 * 10^{15}$	$1.6195 * 10^{15}$	$1.0832 * 10^{15}$	$3.0010 * 10^{15}$
c_0	$-6.1078 * 10^{6}$	$1.6260 * 10^{6}$	$-6.1071 * 10^{3}$	$1.3259 * 10^{1}$
c_1	$4.8821 * 10^{5}$	$-3.1284 * 10^{5}$	$2.2593 * 10^{3}$	0.0739
c_2		$1.4565 * 10^{4}$	$6.1458 * 10^{2}$	
c_3			$1.6909 * 10^{2}$	

To give a sense of the variation in performance, we show the comparative run-times of the tools on the original centralized dispatcher and the decomposed dispatcher versions of Chiron, with two artists and increasing numbers of events. For the original Chiron program, we show the comparison for property **p07** in Figure 30, where FLAVERS exhibited its best comparative performance, and for property **p09** in Figure 31, where FLAVERS exhibited its worst comparative performance. In both cases, FLAVERS appears to scale significantly better than the other verification techniques. The performance of different verification tools is highly sensitive to program structure. Although FLAVERS performs equally well for both the centralized and decomposed dispatcher versions of Chiron, the performance of other tools improves, in some cases dramatically, for the decomposed dispatcher version as illustrated in Figure 32. Although FLAVERS does not have the best absolute run-time for that property, the rate of growth of its run-time appears better than, or at least quite competitive with, the other tools.

7.3 Model Construction

The times presented in the previous subsection included the cost for buiding the artifacts from the CFGs, and running state propagation. Time usually grew no worse than cubically in the size of the system being analyzed. To determine where the time was spent, we broke the process down into three steps: the cost for building the property automaton, the constraint automata, and the TFG without MIP edges; the cost for computing the MHP information and adding the MIP edges; and the cost for running state propagation.

Figures 33 and 34 show this breakdown for two of the properties of the Gas Station program. The lines in the plot show cumulative times for the process, meaning the lowest line is just the time for building the initial artifacts; the middle line includes both the time for building the artifacts, running the MHP

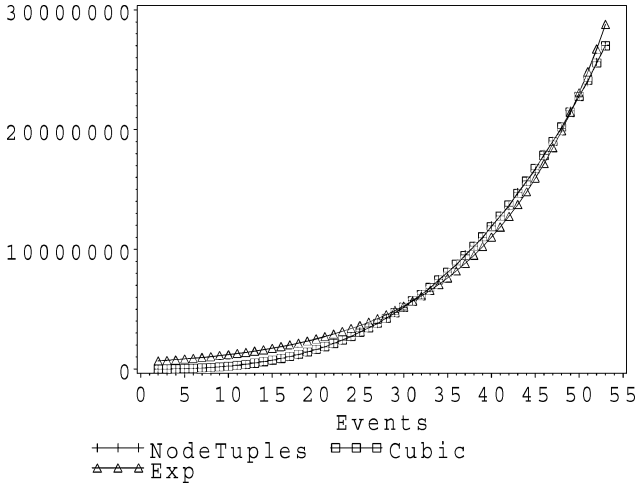

Fig. 29. Fit comparison for **p09** Node-Tuples.

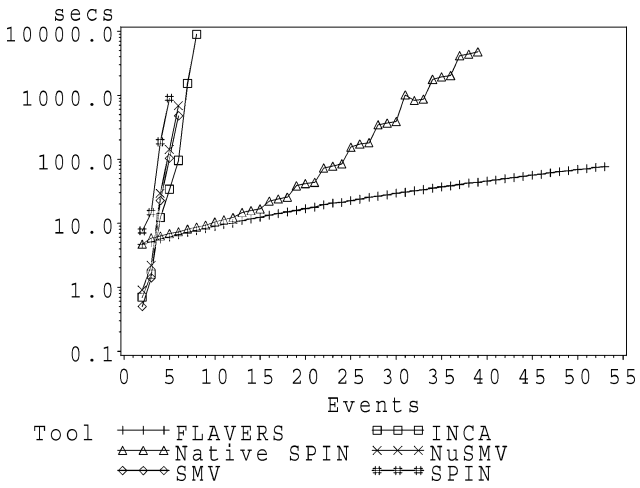

Fig. 30. Tool comparison for **p07** on original Chiron.

algorithm, and adding the MIP edges; and the top line is the time for the entire process, including state propagation.

Across all of the properties presented in the article, the plots tended to look like one of these two plots. Either the three times grew relatively evenly as the system size increased, as in Figure 33, which shows the time for the **customer 1 start/stop** property on the Gas Station with 3 pumps, or the cost of state propagation was the dominant cost for the process, as in Figure 34, which shows the time for the **correct change** property on the Gas Station with 3 pumps.

Since the cost for the overall process was usually no worse than cubic, this means that the cost for each of these steps was no worse than cubic.

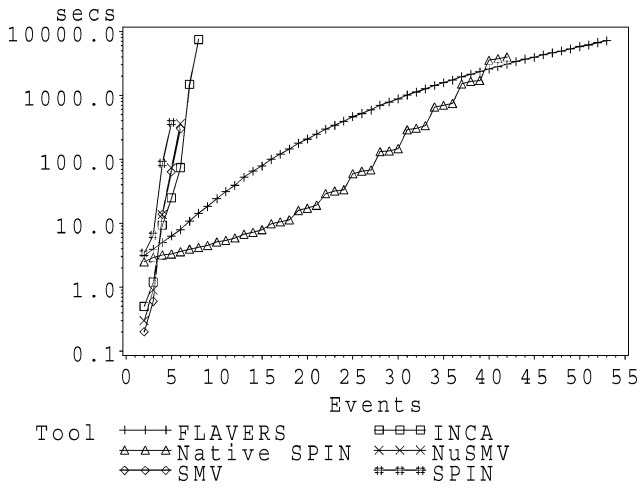

Fig. 31. Tool comparison for **p09** on original Chiron.

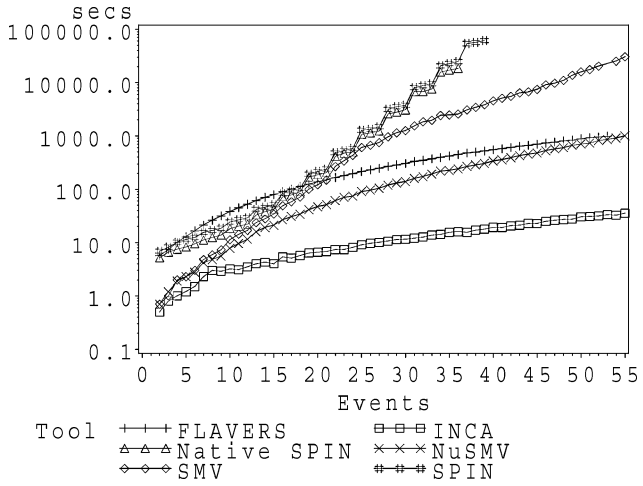

Fig. 32. Tool comparison for **p07** on decomposed dispatcher Chiron.

7.4 Discussion

The empirical results in this section demonstrate the feasibility of FLAVERS analyses. In particular, they demonstrate that there exist relevant properties of nontrivial programs that can be efficiently verified using an analysis approach, where the precision of the model is improved incrementally. It was shown empirically that the computational time and space for a FLAVERS analysis, starting from annotated CFGs through to finding a conclusive result once the set of constraints was known, grew at a rate that was usually no more than cubic.

In this study, we looked at three properties for the Readers/Writers program, two properties for the Gas Station program, two properties for each of four variants of the Dining Philosopher program, and nine properties for the Chiron program, for a total of 22 properties. Of these, 19 were scalable to programs up

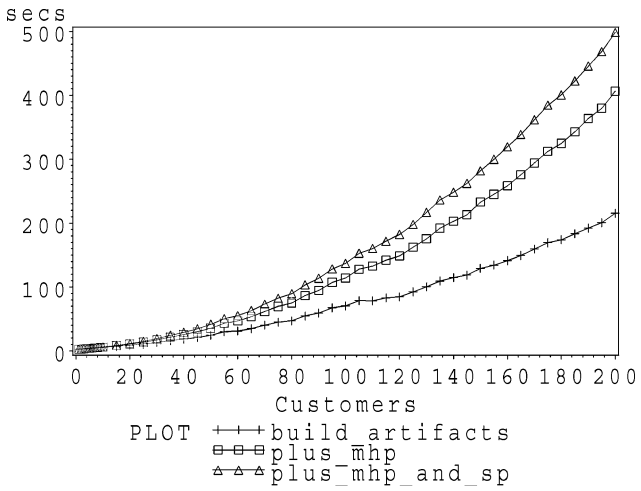

Fig. 33. Cost breakdown for **customer 1 start/stop** with 3 pumps.

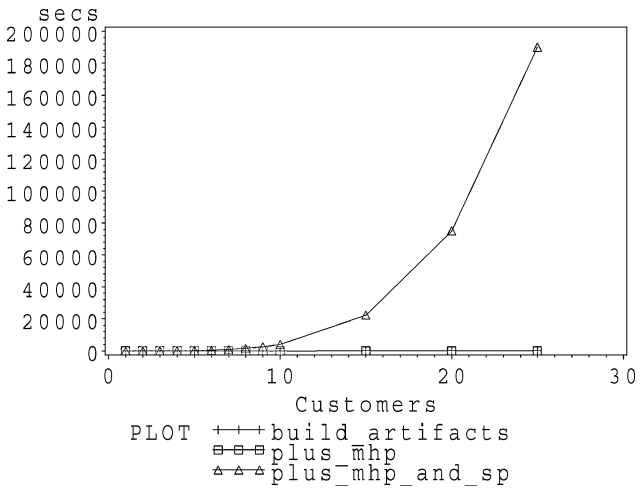

Fig. 34. Cost breakdown for **correct change** with 3 pumps.

to 200 replications (or in the case of Chiron 53), where a replication may have required the addition of a single task, such as a customer in the Gas Station program, or several tasks, such as a philosopher and a fork in the standard Dining Philosopher program. Two of the properties, the **mutual exclusion** property of the Gas Station program and the **no write while reading** property of the Readers/Writers program, had to be reformulated before they could be verified on programs scaled to 200 replications. The property that could not be scaled in our experiments, the **correct change** property of the Gas Station program, appears to need space cubic in the number of customers in the system, but with a large coefficient in front of the cubic term, as shown in Table III.

For all the programs we considered, both the lines of code and the number of tasks in the system grew linearly as we scaled the programs. While FLAVERS'

worst-case performance is $\mathcal{O}(S^6)$, where S is the number of statements in the program, our experimental results provide good evidence that, in practice, we can expect to see performance that is $\mathcal{O}(S^3)$. In practical terms, there was only one property of the 22 properties that could not be scaled.

For most of the properties verified, the number of constraints needed did not change when the program size was scaled up. Exceptions were the **mutual exclusion** and **correct change** properties of the Gas Station program. For these properties, we could not scale the programs as far as we wanted. We also failed to scale the Chiron analyses as far as we wanted, but this was due to limitations in the language-processing tools.

Although the number of constraints remained constant for most problems, the size of some of the constraints needed to be scaled as the program size was scaled. For example, in the **correct change** property of the Gas Station program, the number of states in the TAs for the operator and pump tasks grows linearly with the size of the program. Although FLAVERS did not scale well on this problem, there were other problems where the size of the constraints scaled, and where the analysis did scale well. The number of states in the TA also grows linearly with the number of philosophers for the manager task in the **mutual exclusion** property of the Dining Philosopher program with the Fork Manager. Verifying this property on the Fork Manager version required less time than on the Dictionary and Host versions, where the size of the constraints remained constant as the programs scaled.

For almost all of the scalable programs we considered, the rate of growth of analysis cost was best explained by a cubic polynomial with respect to the number of tasks in the program. This rate of growth included the time for generating all of the artifacts, starting from the CFGs. For this study, we do not consider LOC to be a good metric for measuring the complexity of the program. By using alphabet refinement, FLAVERS builds its model from only the portions of the program relevant to the property being checked. Thus, a large program with sparse events will have a small model. Still, with respect to the number of LOC in the system, the worst-case bound of FLAVERS is a sixth-degree polynomial. For all of the problems we evaluated, the number of LOC grew linearly with the number of tasks in the system. This means FLAVERS' actual performance was cubic with respect to the number of LOC on these systems. Similarly, we saw the number of nodes in the TFG growing linearly with respect to the number of lines of code, not quadratically as indicated by the worst-case bounds.

In FLAVERS, the number of node-tuples in an analysis gives a good indication of the amount of memory needed to perform an analysis. In most of our examples, the number of node-tuples grew quadratically with the size of the system, indicating FLAVERS offers practical performance with regards to space, in addition to time.

We attribute this performance to FLAVERS' approach of incrementally improving the model until it is precise enough to prove the property. It is well known that the cost of unreduced state-space enumeration techniques scales exponentially with the number of tasks [Taylor 1983a]. Based on the performance of other finite-state verification approaches on these or similar programs, it is not the case that we are choosing "easy" programs as examples. FLAVERS

is demonstrating polynomial growth on programs with state spaces that grow exponentially in the number of potential program executions. Unfortunately, we also saw situations where the rate of growth in FLAVERS' analysis time was unacceptable. Recent work has been unable to predict when FLAVERS or other finite-state verification techniques will have such undesirable performance [Avrunin et al. 1999].

Although FLAVERS' performance on the considered problems is often good, these programs are small and mostly drawn from the standard example programs used to test finite-state verification tools. The largest program analyzed was the Chiron example, which is about 15,000 lines long. Chiron is not a contrived program designed to test finite-state verification tools. Since FLAVERS' performance on Chiron was similar to its performance on the other programs, we have reason to believe that we will see comparable performance on other real world programs.

As mentioned previously, although our experiments give the cost of running FLAVERS on a system where a minimal set of constraints is known, they do not give any indication of the overall or human cost of applying FLAVERS to verify a property. In particular, we do not look at the cost of finding a set of constraints sufficient for proving that a property holds. In our experience, it is not too difficult to find such a set of constraints; it usually requires one to three iterations. Moreover, the cost of running FLAVERS for inconclusive problems is usually much smaller than running it on conclusive problems since our actual implementation stops as soon as one property violation is found. Thus, the cost for inconclusive problems is very variable and somewhat serendipitous. Finally note that there are heuristics that appear to be good at predicting the constraints that should be used [Tan et al. 2004] to help reduce the number of iterations needed to find a set of constraints sufficient for either returning a conclusive result or determining that an inconclusive result is a real error and not due to a spurious path.

8. RELATED WORK

As both a flow-analysis and a finite-state verification technique, FLAVERS draws on a long history of previous work. We focus our discussion here on the connections between FLAVERS and flow analysis of concurrent systems, model-checking techniques, and systematic approaches to abstraction.

8.1 Flow Analysis

Program flow analysis was developed to enable optimizing compilers to generate efficient code [Aho et al. 1986]. In flow analysis, specific patterns of behavior are analyzed by computing the fix-point of a relation defined over a graph structure whose nodes explicitly encode control information.

Most flow analyses are targeted at analyzing the system for a single property, such as the set of live variables at each system statement [Aho et al. 1986]. FLAVERS uses an approach that combines multiple flow analyses into a single analysis. Combining multiple dependent flow analyses into a single flow analysis (e.g., Click and Cooper [1995]; Holley and Rosen [1981]) offers the potential

to improve the precision of analysis results, for example, by eliminating some infeasible paths. In FLAVERS, analyses based on constraint automata are combined on-the-fly in such a way that any automaton reaching a violation state causes the entire combined analysis to reach a violation state. The strength of this approach is that it makes it easy to define analyses that are sensitive to selected semantic features of the program, while efficiently abstracting the balance of the program.

The bulk of flow-analysis research has been aimed at sequential software, but flow analysis of concurrent software is becoming increasingly important. Analyses that are both flow-insensitive and context-insensitive can be adapted directly to reason about concurrent software since, by definition, they do not reason about the ordering of statements either within or between tasks. Flow-sensitive analyses typically use flow-graph models that are, essentially, a collection of task flow graphs with additional edges, or labels, to represent intertask synchronization and communication. These include, for example, analyses to detect the potential for statements to execute concurrently [Masticola and Ryder 1993; Naumovich and Avrunin 1998; Naumovich et al. 1999b], data races [Netzer and Miller 1990], and pointer and escape analyses [Salcianu and Rinard 2001]. For more general analyses, however, interleavings need to either be modeled in the graph or incorporated into the analysis algorithm. The flow-graph model used in FLAVERS, described in Section 4, explicitly represents the potential interleaving of pairs of events.

The application of flow-analysis techniques for validation of sequential code [Johnson 1978; Osterweil and Fosdick 1976; Ryder 1974] followed quite quickly after the development of flow analysis for optimization. These approaches used a set of analyses formulated to detect a fixed set of anomalies. The Cecil/Cesar system [Olender and Osterweil 1990, 1992] generalized this approach by providing a single parameterized flow analysis for properties expressed as QREs. FLAVERS builds on this work by extending the analysis to concurrent systems and by incorporating a variety of mechanisms to increase the precision of the results.

More recently, the SLAM [Ball and Rajamani 2001] and BLAST [Henzinger et al. 2002] projects have each adapted an existing context-sensitive flow-analysis algorithm [Reps et al. 1995] to reason about correctness properties of sequential C programs that are specified as automata. Like FLAVERS, these projects have a flexible approach to incorporating selected semantic features of the program into their analysis. Unlike FLAVERS, both the SLAM and BLAST provide automated support for determining how the program should be abstracted. This work is similar to determining what constraints should be used in FLAVERS, a direction of future work for us.

Flow analysis for validation of concurrent programs has been less well studied. Masticola and Ryder developed a system for checking deadlock freedom in Ada tasking systems [Masticola and Ryder 1991; Masticola 1993]. They incorporate an approach for refining the system model that is similar in spirit to the refinements described in Section 4. In their approach, they use flow analyses to refine the system model prior to the final analysis, thereby increasing the precision of the analysis results [Masticola and Ryder 1993]. FLAVERS differs

from this work in that it applies refinements only a single time, provides for more precise analysis through the use of constraints, and supports analysis of a wide class of properties as opposed to just deadlock.

8.2 Model Checking

Model checking refers to a class of techniques for checking conformance of the behaviors encoded in a finite-state transition system to a formula written in a temporal logic [Manna and Pnueli 1991]. Model checking has gained popularity due to its success in reasoning about hardware components and communication protocols [Clarke et al. 1999].

The relationship between model checking and flow analysis is very strong; both involve a fix-point computation of a relation that encodes a specified property. In fact, it has been shown that a wide range of useful flow analyses can be formulated as model-checking problems and vice versa [Müller-Olm et al. 1999; Schmidt 1998]. The application of model-checking techniques to software relies on a preprocessing step for extracting a safe, compact model of a system's behavior. Only very recent approaches automate this step [Ball and Rajamani 2001; Corbett et al. 2000; Demartini et al. 1999; Visser et al. 2000]. Once a model is produced, properties of the model can be checked. In contrast, flow analyses are designed to automatically extract a compact model and then to analyze this model. FLAVERS applies both of these approaches by extracting a safe, compact model through the application of refinements, optionally extracting additional information through the use of constraints, and then performing the property analysis [Dwyer and Clarke 1994].

Explicit-state model checking involves the construction of the reachable state space of a transition system and the calculation of the formulas that are true at each state [Clarke et al. 1999]. Much of the early work on static concurrency analysis took this approach, although each limited their search to specific system states, such as states that violate assertions or describe deadlocks [Taylor 1983b; Young et al. 1995]. Unfortunately, the size of a system's reachable state space grows exponentially with the number of tasks, even when all data are ignored [Taylor 1983a]. One popular form of explicit-state model checking is the automata-based approach employed in the SPIN model checker [Holzmann 1997]. In this approach, the property to be checked is combined with the state space to reduce model checking to state reachability; for liveness properties the check involves a form of cycle detection. FLAVERS can be viewed as performing a kind of explicit-state analysis. In fact, if constraints are used for each system task and system variable, then FLAVERS will construct the full reachable state space encoded in the sets of tuples propagated to the TFG nodes. The strength of FLAVERS is that it provides a very flexible mechanism for defining abstracted state spaces that are very compact and for which analysis is relatively inexpensive.

To increase the applicability of explicit-state analysis, researchers have investigated a variety of techniques including partial-order reductions that detect equivalent states and then limit analysis to a single representative of the equivalence class (e.g., Peled [1998]), and compositional approaches that build

and analyze the state space compositionally, (e.g., Cheung and Kramer [1996]; Cleaveland et al. [1993]; Yeh and Young [1991]). Although none of these techniques have been able to reduce the worst-case complexity of analysis to be subexponential, each technique has been successfully applied to selected systems. FLAVERS incorporates a form of partial-order reduction [Naumovich et al. 1999] and has been used to perform assume-guarantee compositional reasoning [Dwyer 1997].

Information about the execution environment of the system under analysis can help reduce the number of reachable system states. Cheung and Kramer [1996] model behaviors of execution environments with *context constraints* in the context of explicit compositional model checking. In their approach, context constraints are FSAs that are composed with the FSAs modeling components of the system under analysis. These context constraints are quite similar to FLAVERS' constraints, and our use of constraints to model the execution environment is identical to their approach. FLAVERS' constraints are more general for encoding different semantic aspects of a program. For example, VA constraints are derived from the abstract finite semantics of a variable's data type.

There is a rich literature on nonenumerative state space analyses. Some of the most well-studied are the symbolic model checking approaches [McMillan 1993]. These operate by manipulating an encoding of the next-state function using some form of decision diagram, typically an ordered binary decision diagram [Bryant 1992]. Fix-point calculations are performed using this representation to determine the set of states that satisfy a given formula. For certain systems, symbolic encoding can yield reductions in the space and time required for model checking by several orders of magnitude. Unfortunately, it has proven very difficult to predict whether such reductions will be effective for a given system and property to be analyzed. Other nonenumerative techniques can offer similar advantages, for example, deductive methods based on theorem proving, (e.g., Huisman et al. [1999]; Flanagan et al. [2002]), and methods based on integer programming [Corbett and Avrunin 1995].

8.3 State-space Abstraction

Fundamental decidability results force all precise concurrency analyses to employ reductions or abstractions to achieve tractability. Flow analyses encode abstractions that preserve only the property under analysis. Flow analyses expressed as abstract interpretations [Cousot and Cousot 1977] make the nature of the abstraction explicit. Recent work has applied abstract interpretation to the generation of abstracted state spaces that are amenable to model checking.

In predicate abstraction, one provides predicates over a transition system's state variables and a new transition system is derived that has a boolean state variable for each predicate [Graf and Saïdi 1997]. Scaling this approach to programming languages is a challenge, and several partial solutions have been investigated (e.g., Ball and Rajamani [2001]; Dwyer et al. [2001]). FLAVERS can accommodate these kinds of abstractions through the use of constraints. For instance, an integer variable could be modeled by a constraint automaton

with five states: *unknown*, *positive*, *negative*, *zero*, and *violation*. Transitions in the automaton would be defined appropriately for operations on the variable, such as tests of, and assignments to, the variable.

9. CONCLUSION

We have presented FLAVERS, a finite-state verification approach that analyzes whether software systems satisfy user-defined properties. In contrast to other finite-state verification techniques, FLAVERS creates a system model that is relatively small but still conservative with respect to the property that is being evaluated. FLAVERS achieves this reduction in size at the cost of precision. Analysts can improve the precision of the results as needed by selectively and judiciously incorporating additional semantic information into the analysis problem. FLAVERS provides automated support for creating some of the common constraints that are used to represent this additional information.

Our evaluation, although preliminary, indicates that sufficient precision can usually be achieved relatively easily and that the cost for many problems grows as a low-order polynomial in the size of the system. In addition to the evaluation reported here, FLAVERS has been applied in a number of interesting ways. It has been used in an empirical evaluation of concurrency analysis techniques [Chamillard et al. 1996] and to analyze communication protocols [Naumovich et al. 1996], partial software systems [Dwyer 1997], and architectural system descriptions [Naumovich et al. 1997]. It has also been applied to advanced distributed simulation systems [Bouwens et al. 1996] and to evaluate the adherence of such systems to high-level architectural requirements [Science Applications International Corporation 1997]. In these latter two studies, FLAVERS detected previously unknown errors in the systems being analyzed. As is often the case with rigorous verification techniques, many of the errors were found just because of the additional scrutiny associated with formulating the properties and annotating the code. Although more errors were detected while preparing the associated artifacts than while using verification, the errors that FLAVERS did detect were deemed to be more complicated then those caught during preparation. In other words, the errors that FLAVERS detected could not be easily observed by the analyst without such validation tools.

Although this article has described the approach in some detail for Ada systems, the approach is relatively language independent. Each programming language must be carefully translated into the system model so that the ordering of events is conservatively captured in the TFG's flow of control. To date, translators have been successfully developed by others for C++ and Jovial. We are currently exploring some alternative models for Java [Naumovich et al. 1999a]. Interestingly, changes to the state-propagation algorithm to support these different languages were relatively minor. The MHP algorithm, however, seems to require some careful specialization for programming languages with different models of concurrency [Naumovich and Avrunin 1998; Naumovich et al. 1999b].

In this article, we have emphasized how FLAVERS can be used to verify properties of software systems. In addition to being applicable to programs, the FLAVERS approach is applicable to other artifacts that capture the flow

of events through a system. For example, it could be applied to architectural descriptions or detailed designs. When applied to programs, the verification process could be viewed as a complementary activity to testing. Alternatively, it could be used to help with debugging, where a hypothesis about a fault is formulated as a property and FLAVERS is then used to determine if there are any traces through the TFG (and thus consequently through the code) that would cause this fault to occur. Such support is particularly important for distributed systems, since output results may vary depending on the run-time task scheduling.

There are some limitations to the approach used in FLAVERS. The model is less precise than reachability-graph based approaches [Cheung and Kramer 1993; Holzmann 1997], and thus does not seem well suited for the kinds of detailed analysis required for deadlock detection and definition/reference anomaly detection. Moreover, it is restricted to properties that can be represented in a regular language. Being a static analysis approach, it also cannot reason about dynamic configurations. Thus, as was done in our experimental evaluation, specific configurations must be selected. Users can often present compelling arguments that verification over a restricted set of configurations is equivalent to verification over the general set, but this is not always possible or easy. Others have argued, however, that most faults are detected with small sample sizes [Jackson and Vaziri 2000].

The current implementation of FLAVERS/Ada does not support dynamically-allocated data, recursion, exceptions, generics, or some of the low-level representation data types. There has been considerable work on modeling recursion for dataflow analysis [Reps et al. 1995], but using these techniques to model multithreaded systems is not practical [Ramalingam 2000]. By reducing the flow-graph representation based on the property to be checked, FLAVERS may localize the need to model recursion to a few threads, and we are studying the extent to which existing techniques can be applied in this case. The other limitations primarily pose engineering hurdles that could be addressed with appropriate effort.

There are several interesting research directions to be explored. Improving performance is always of concern. We are particularly interested in exploring techniques for decomposing the analysis problem. Because the events of interest tend to be sparsely located throughout the system, the resulting model tends to be small after alphabet refinement and other optimizations are applied. In all the programs that we have examined, using inlining to create the model worked reasonably well since the optimized representation for an invoked component was usually very small, and often was empty. This will not always be the case, however, so we are exploring how counterexamples can be used to automate assume-guarantee reasoning using the L* learning algorithm [Cobleigh et al. 2003].

Counterexample generation also raises some interesting issues. We have studied how different variations of the state-propagation algorithm can impact performance, depending on whether consistent or inconsistent results are expected or if short counterexample traces are desired [Cobleigh et al. 2001]. Path length is only one of the important attributes of counterexamples, however.

Another concern is determining if the counterexample is executable. One interesting approach, used in SLAM [Ball and Rajamani 2001], is to employ symbolic evaluation techniques to help determine path executability. We are currently exploring techniques for guiding the selection of counterexamples using the constraint mechanism, not only for path feasibility, but also as a debugging aid.

Specifying properties is another area for future work. It is surprisingly difficult to correctly capture the intent of a system. Although regular expressions and FSA models seem more natural to practitioners than temporal logics, many subtle options need to be considered when formulating a property. A recent experimental study showed that many finite-state verification properties could be mapped to a small number of property patterns [Dwyer et al. 1999]. Building on that work, a framework for extending these patterns into more extensively parameterized templates has been developed. This framework provides FSA-based templates as well as a natural-language interface for describing properties [Smith et al. 2002]. The hope is that this framework will help practitioners formulate properties in a way that is more natural and intuitive for them, but, at the same time, will provide a rigorous mathematical representation that can be used as a basis for verification.

Finally, model development is an area of future research that we have been actively exploring. One issue is whether the iterative model development approach used by FLAVERS can be supplanted with an approach that extensively exploits compiler optimizations, partial evaluation, and abstraction techniques. The Bandera system is being developed to explore this issue [Corbett et al. 2000]. It is employing a number of techniques, such as program slicing, to automatically create a concise model, optimized for the property and system configuration. This direction is also being explored in the work on Java PathFinder [Visser et al. 2000]. The resulting models could be sufficiently precise that incremental improvement might not be necessary. Alternatively, this concise model might be a better starting point for the incremental refinement approach used by FLAVERS. We are currently developing a version of FLAVERS where the TFG is built from the highly-optimized internal representation produced by Bandera. We are also exploring alternative representations, such as maintaining some of the variable information directly in the TFG representation, as well as doing some dynamic expression evaluation during the analysis process itself.

Although there are a number of interesting research issues to be explored, the FLAVERS toolset demonstrates the feasibility of applying flow-analysis-based verification techniques to software systems. It is extremely important that practical techniques be developed to help reason about software systems. Distributed systems are particularly problematic, and are becoming extremely common. Such systems are more difficult to reason about, to test, and to debug than sequential systems, for which these problems are difficult enough. Although the experimental evaluation reported here is preliminary, and the example programs are relativley modest in size, the experimental results are promising and lead us to believe that finite-state verification techniques, in general, and the FLAVERS approach, in particular, hold considerable promise for helping developers reason about systems.

ELECTRONIC APPENDIXES

Electronic Appendixes B, the size information for all of the property checks presented in the body of this article, and C, the timing information for all of the property checks, are available in the ACM Digital Library.

APPENDIX

This appendix contains proofs of conservativeness of the reduced CFG model, TFG model, and the state-propagation algorithm of FLAVERS, and the proof of complexity of the state-propagation algorithm.

A. PROOFS

A.1 RCFG Conservativeness

The following lemma and theorem prove that the CFG alphabet refinement is conservative. We call nodes in the CFG that are removed by the refinement algorithm, and thus do not appear in the RCFG, *removed* and all other nodes *retained*. In the lemma, we refer to two CFGs, G and G'. We use superscripts to differentiate between different elements of the tuples of G and G'. For example, N^G refers to the set of nodes of G, and $N^{G'}$ refers to the set of nodes of G'. We use this convention to disambiguate the origin of an element when necessary.

LEMMA A.1. *Let G be a CFG and G' the RCFG obtained from G, using the alphabet refinement algorithm. For any pair of retained nodes m and n from N^G with corresponding nodes m' and n' in $N^{G'}$, if there is a path from m to n in G that consists only of removed nodes, then there is an edge (m', n') in $E^{G'}$.*

PROOF. The proof is by induction on the length of the path. If this path is of length 1, that is, $(m, n) \in E^G$, then trivially $(m', n') \in E^{G'}$, since only those edges that are incident to a removed node are removed by the alphabet-refinement algorithm.

Assume that the statement of the theorem holds for paths of length k. Consider any path of length $k+1$: $m, n_1, \ldots, n_k, n$, such that $n_1, \ldots, n_k$ are removed nodes. Let $n_i \in \{n_1, \ldots, n_k\}$ be the first of the nodes $n_1, \ldots, n_k$ removed by the alphabet-refinement algorithm. Then, since n_{i-1} is a predecessor of n_i, and n_{i+1} is a successor of n, an edge (n_{i-1}, n_{i+1}) will be created (temporarily, if $n_{i-1} \neq m$ or $n_{i+1} \neq n$). After this, the length of the path $m, n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_k, n$ is k and, according to the induction hypothesis, eventually an edge between m' and n' is created. □

THEOREM 4.2 (RCFG CONSERVATIVENESS). *Let G be a CFG and G' the corresponding RCFG obtained using the alphabet-refinement algorithm. Let $R \subseteq N^G$ be the set of nodes of G that were retained in G'. For any sequence of nodes $\pi \in (N^G)^*$, define* $\mathrm{Map}(\pi) \in (N^{G'})^*$ *as the sequence of nodes in G' that corresponds to the nodes in the projection $\pi|_R$. If π is a path in G, then* $\mathrm{Map}(\pi)$ *is a path in G'.*

PROOF. We argue by induction on the number of nodes in π. The statement of the theorem trivially holds if π is a path of length 0. Suppose the statement

of the theorem holds for any path of length k. Consider any path of length $k+1$. It can be written as $\pi_k n$, where π_k is a path of length k, and $n \in N^G$ is the final node in the path.

First, suppose $n \notin R$. Then $\pi_k|_R = \pi_k n|_R$. By our inductive hypothesis, we know that $\mathrm{Map}(\pi_k)$ is a path in G', so $\mathrm{Map}(\pi_k n)$ is also a path in G', as desired.

Second, suppose $n \in R$. Let $m \in N^G$ be the last retained node in π_k. Note that path π_k always contains such a node, since any path in G starts with $n^G_{initial}$ and, according to the alphabet refinement algorithm, $n^G_{initial}$ is retained. Since $m, n \in R$ and are connected by a path of removed nodes, then by Lemma A.1, there is an edge $(m', n') \in E^{G'}$. Using the inductive hypothesis, $\mathrm{Map}(\pi_k|_R)$ is a path in G'. Since $(m', n') \in E^{G'}$, $\mathrm{Map}(\pi_k n)$ is also a path in G', as desired. □

A.2 TFG Conservativeness

The following theorem states and proves conservativeness of the TFG model.

THEOREM 4.4 (TFG CONSERVATIVENESS). *For each actual execution of the system S, the TFG contains a path, starting in the initial node, that exhibits the same set of events projected on Σ_I as this execution projected on Σ_I. Thus, $\mathcal{L}(S) \subseteq \mathcal{L}(G)$.*

PROOF. For easy correlation between the system and the TFG, we use TFG nodes to represent a unit of system execution. This could be viewed as introducing a unique event to mark each node in the TFG and tracking these events in the system. Thus, in this proof we say "a node is executed" to mean that the code corresponding to this node is executed. The system executes a communication node when the synchronization between the two tasks represented by this communication node takes place. The proof is by induction on the length of the execution.

Executions of length 1 correspond to execution of a single node, which must be the initial node $n^i_{initial}$ in one of the system tasks T^i. Since the initial node of the TFG is connected to each of the initial nodes of the tasks, the corresponding path in the TFG is $n_{initial}, n^i_{initial}$.

As an inductive hypothesis, suppose that the statement of the theorem holds for executions of length k. Consider any execution of length $k+1$: $n_{initial}, n_1, \ldots, n_k, n_{k+1}$. By the induction hypothesis, there is a path $n_{initial}, n_1, \ldots, n_k$ in the TFG. We consider several cases based on the relationship between n_k and n_{k+1}.

—**n_k and n_{k+1} are in the same task T^i.** There are several possibilities to consider.

(1) First, assume that both n_k and n_{k+1} are local nodes, but neither corresponds to a task synchronization. Since they happen subsequently on an actual execution of the system, the code corresponding to n_k precedes the code corresponding to n_{k+1} in the control flow of T^i. Since, by Theorem 4.2, RCFGs are conservative representations of individual tasks, edge (n_k, n_{k+1}) is in the RCFG for T^i. By the TFG construction, this edge is also in the TFG, and so path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.

(2) Second, suppose n_k is a local node that corresponds to a task synchronization statement of task T^i. The next event in the system in task T^i after a synchronization statement is the actual synchronization with another task. Therefore, n_{k+1} must be a communication node. By the TFG construction, this is indeed the case, and $(n_k, n_{k+1}) \in E_{com}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
(3) Third, suppose that n_k corresponds to a task synchronization, which means that $n_k \in N_{com}$. Then n_{k+1} must be a local node in task T^i that immediately follows the synchronization represented by n_k. By the TFG construction, $(n_k, n_{k+1}) \in E_{com}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
(4) Finally, by the TFG construction, n_k and n_{k+1} cannot both be communication nodes.

—**n_k and n_{k+1} belong to different sets of tasks.** Let n_k belong to task T^i and n_{k+1} belong to task T^j, where $i \neq j$. This case represents a context switch between two tasks, where executions of T^i and T^j are interleaved. The MHP algorithm is conservative [Naumovich and Avrunin 1998] and, therefore, it detects that n_k and n_{k+1} may happen in parallel. Since construction of MIP edges is directly based on the MHP algorithm, $(n_k, n_{k+1}) \in E_{mip}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.

—**n_{k+1} is the final node n_{final}.** In this case, the execution $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ represents a terminal execution of the system, which means that the task RCFGs execute their final nodes, and so n_k is one of these final nodes n^i_{final}. Since the TFG contains an edge from each n^i_{final} to the final node of the TFG, n_{final}, the path corresponding to this execution is in the TFG. □

A.3 State-Propagation Conservativeness

The following theorem proves conservativeness of the state-propagation algorithm of FLAVERS.

THEOREM 5.1 (STATE-PROPAGATION CONSERVATIVENESS). *If state-propagation analysis over a TFG determines that a property holds, then this property holds for the actual system. Formally, for any TFG node n, if there exists an actual program execution on which state r of the property is associated with n, then after the state-propagation algorithm terminates, $r \in States[n]$.*

PROOF. According to Theorem 4.4 about conservativeness of the TFG representation, there is a path $\pi = n_{initial}, n_1, \ldots, n_k$ that corresponds to the actual execution described in the statement of this theorem. Let s_i be the state of the property on node n_i of the path. We use an argument by induction on the length of π.

Suppose first that the length of π is 0, meaning it consists of only $n_{initial}$. Since the initial node represents the state of the program before the execution begins, the only state of the property that can correspond to this state of the program is the start state, s. According to the initialization step of the state-propagation algorithm, $s \in States[n_{initial}]$.

Assume that the statement of the theorem holds for executions that correspond to paths of length k. Consider the case where the length of π is $k+1$. By the induction hypothesis, state s_k that is associated with n_k on path π is in $States[n_k]$. When s_k was added to $States[n_k]$, the state propagation algorithm placed n_{k+1} on the worklist, since edge (n_k, n_{k+1}) is in the TFG. When the algorithm subsequently processes node n_{k+1}, all states associated with each of n_{k+1}'s predecessors, including s_k, are propagated through n_{k+1}. Therefore, state s_{k+1} will be placed in $States[n_{k+1}]$. □

Next, we show that the results of state propagation with constraints are conservative. This will be proved in two parts. The first part shows that state propagation with constraints, where pruning of tuples is not performed when a constraint automaton enters its violation state, is conservative. Next, we show that the pruning of tuples does not change the conservativeness of the results of state propagation. For the first part of the proof, we need to change the definition of $F^n(X)$, the transfer function for a TFG node n and a set of tuples X, from the one given in Subsection 6.3 to one that does not prune tuples:

$$\forall n \in N, \forall X \subseteq Tuples, F^n(X) = \{f^n(T) \mid T \in X\}. \tag{1}$$

COROLLARY A.2 (CONSERVATIVENESS OF STATE PROPAGATION WITH CONSTRAINTS AND WITHOUT PRUNING TUPLES). *Given a TFG, a property automaton P, and a set of constraint automata $C_1, C_2, \ldots, C_k$, for any TFG node n, if there exists an actual program execution on which a tuple $T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k})$ is associated with n, then after the state-propagation algorithm terminates, using the propagation function given in Equation* (1)*, T is associated with n.*

PROOF. The proof of this corollary mimics the proof of Theorem 5.1. □

THEOREM A.3 (CONSERVATIVENESS OF STATE PROPAGATION WITH CONSTRAINTS AND WITH PRUNING TUPLES). *Given a TFG, a property automaton P, and a set of constraint automata $C_1, C_2, \ldots, C_k$, for any TFG node n, if there exists an actual program execution on which a tuple $T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k})$, such that $\forall i, 1 \leq i \leq k, t^{C_i} \neq v^{C_i}$, is associated with n_{final}, then after the state-propagation algorithm terminates, using the propagation functions from Section 6.3, T is associated with n_{final}.*

PROOF. The proof of this theorem follows from that of Corollary A.2 if we can prove that when the transfer function without pruning (Equation (1)) is used, then if a tuple $T_1 = ({t_1}^P, {t_1}^{C_1}, {t_1}^{C_2}, \ldots, {t_1}^{C_k})$ that has one of its constraint automata C_i in the violation state, that is, for some $1 \leq i \leq k$, ${t_1}^{C_i} = v^{C_i}$, is associated with node n of the TFG, then the propagation of tuple T_1 does not result in a tuple $T_2 = ({t_2}^P, {t_2}^{C_1}, {t_2}^{C_2}, \ldots, {t_2}^{C_k})$ being associated with n_{final}, such that ${t_2}^{C_i} \neq v^{C_i}$. If such a tuple T_2 could be associated with n_{final}, then state propagation would examine T_2 when determining if the property holds and removing T_1 would be unsafe. There are two cases to consider:

—First, suppose that $n \neq n_{final}$. Then if tuple T_1 is propagated along a path $n, n_1, \ldots, n_k, n_{final}$, then the tuple associated with n_{final} would be $T_2 = f^{n_{final}}(f^{n_k}(\cdots f^{n_1}(T_1)\cdots))$. If ${t_1}^{C_i} = v^{C_i}$, then, according to the propagation function in Equation (1), and because all transitions out of a violation state are self-transitions, ${t_2}^{C_i} = v^{C_i}$.

—Second, suppose that $n = n_{final}$. Then, since the final node has no outgoing edges, there is no way that T_1 can be propagated to any other nodes.

Since pruning tuples with constraints in violation states does not affect the result of state propagation, state propagation with constraints and with pruning tuples is conservative. □

A.4 Complexity of the State-Propagation Algorithm

The following theorem proves the complexity result for the state propagation algorithm from Figure 10.

THEOREM 5.2 (STATE-PROPAGATION COMPLEXITY). *Given a TFG, with nodes N^G, and a property automaton, with states S^P, state-propagation algorithm terminates in $\mathcal{O}(|S^P||N^G|^2)$ time.*

PROOF. First, note that the total number of worklist insertions is $\mathcal{O}(|E^G||S^P|)$, because a node is added to the worklist only if the *States* set of one of its predecessors changes, and each such set can change at most $|S^P|$ times. Since $|E^G| \leq |N^G|^2$, the complexity of maintaining the worklist falls within $\mathcal{O}(|S^P||N^G|^2)$.

We prove that the complexity of recomputing sets *States*, *IN*, V, and U, for any node n, amortized over the course of the algorithm, is $\mathcal{O}(|S^P||N^G|)$. Since there are $|N^G|$ nodes in the TFG, the statements of the theorem will follow.

Consider any node $n \in N^G$. In a run of the algorithm, each of the *IN* and V sets computed for n has at most $|S^P|$ states added to it from each of n's predecessors. Adding a property automaton state to a set or checking its containment in a set takes constant time. Since n can have at most $|N^G| - 1$ predecessors, the total number of operations involving sets *IN* and V for n is $\mathcal{O}(|S^P||N^G|)$.

Computing set U involves propagating a state through node n and checking if the resulting state is already in *States*[n]. Both of these operations take constant time. Since these operations are applied only once to each state that is propagated into n, the complexity of computing set U for n is $\mathcal{O}(|S^P|)$. Similarly, the complexity of computing *States*[n] is also $\mathcal{O}(|S^P|)$.

A state can be added to the *Flow* set of any edge (n, r) only once, after this state has been added to set *States*[n]. Therefore, in a run of the algorithm, at most $|S^P|$ states are added to *Flow*[(n, r)]. Since the number of edges is bounded by $|N^G|^2$, the total cost of adding states to the *Flow* sets is $\mathcal{O}(|S^P||N^G|^2)$. The total number of operations of removing states from *Flow* sets is also $\mathcal{O}(|S^P||N^G|^2)$, since each state put in a *Flow* set has to be removed from this set later.

Finally, determining the results of state propagation requires examining all states associated with the final node. Therefore, this operation is bounded by

the number of states in the property, $|S^P|$. The total complexity of this algorithm is $\mathcal{O}(|S^P||N^G|^2 + |S^P|) = \mathcal{O}(|S^P||N^G|^2)$. □

ACKNOWLEDGMENTS

The authors would like to thank Michael Sutherland of the UMass Statistical Consulting Center for his help in analyzing the experimental results and Rachel Cobleigh, Heather Conboy, Stephen Siegel, and Jianbin Tan for helping with several of the examples. Jay Corbett and Corina Păsăreanu contributed to the Chiron experiments. Leon Osterweil and George Avrunin have been valued colleagues, providing insightful comments over the years.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

AVRUNIN, G. S., BUY, U. A., CORBETT, J. C., DILLON, L. K., AND WILEDEN, J. C. 1991. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Engin. 17*, 11 (Nov.), 1204–1222.

AVRUNIN, G. S., CORBETT, J. C., DWYER, M. B., PĂSĂREANU, C. S., AND SIEGEL, S. F. 1999. Comparing finite-state verification techniques for concurrent software. TR 99-69, University of Massachusetts, Department of Computer Science. (Nov.).

BALL, T. AND RAJAMANI, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th SPIN Workshop*, M. B. Dwyer, Ed. Lecture Notes in Computer Science, vol. 2057. 101–122.

BOUWENS, C., MCKENZIE, R., AND DEAN, C. 1996. Investigating static data flow analysis for advanced distributed simulation verification. In *Proceedings of the 15th Workshop in the Interoperability of Distributed Interactive Simulation*. 473–478.

BRYANT, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24*, 3 (Sept.), 293–318.

CHAMILLARD, A. T., CLARKE, L. A., AND AVRUNIN, G. S. 1996. An empirical comparison of static concurrency analysis techniques. TR 96-84, University of Massachusetts, Department of Computer Science. (May).

CHEUNG, S.-C. AND KRAMER, J. 1993. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the 4th European Software Engineering Conference*, I. Sommerville and M. Paul, Eds. Lecture Notes in Computer Science, vol. 717. 283–300.

CHEUNG, S.-C. AND KRAMER, J. 1996. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Engin. Method. 5*, 4 (Oct.), 334–377.

CLARKE, JR., E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

CLEAVELAND, R., PARROW, J., AND STEFFEN, B. 1993. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst. 15*, 1 (Jan.), 36–72.

CLICK, C. AND COOPER, K. D. 1995. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst. 17*, 2 (March), 181–196.

COBLEIGH, J. M., CLARKE, L. A., AND OSTERWEIL, L. J. 2001. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proceedings of the 23rd International Conference on Software Engineering*. 37–46.

COBLEIGH, J. M., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2003. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. 331–346.

CORBETT, J. C. AND AVRUNIN, G. S. 1994. Towards scalable compositional analysis. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 53–61.

CORBETT, J. C. AND AVRUNIN, G. S. 1995. Using integer programming to verify general safety and liveness properties. *Form. Meth. Syst. Des. 6*, 1 (Jan.), 97–123.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PĂSĂREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*. 439–448.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*. 238–252.

DEMARTINI, C., IOSIF, R., AND SISTO, R. 1999. A deadlock detection tool for concurrent Java programs. *Softw.–Prac. Exper. 29*, 7 (June), 577–603.

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ.

DURI, S., BUY, U., DEVARAPALLI, R., AND SHATZ, S. M. 1993. Using state space reduction methods for deadlock analysis in Ada tasking. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*. 51–60.

DWYER, M. B. 1995. Data flow analysis for verifying correctness properties of concurrent programs. Ph.D. thesis, University of Massachusetts, Amherst.

DWYER, M. B. 1997. Modular flow analysis for concurrent software. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*. 264–273.

DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*. 16–22.

DWYER, M. B. AND CLARKE, L. A. 1994. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 62–75.

DWYER, M. B., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PĂSĂREANU, C. S., ROBBY, VISSER, W., AND ZHEN, H. 2001. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*. 177–187.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 234–245.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. 72–83.

HECHT, M. S. 1977. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier, North-Holland.

HELMBOLD, D. AND LUCKHAM, D. 1985. Debugging Ada tasking programs. *IEEE Softw. 2*, 2 (March), 47–57.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*. 58–70.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct.), 576–580.

HOLLEY, L. H. AND ROSEN, B. K. 1981. Qualified data flow problems. *IEEE Trans. Softw. Engin. SE-7*, 1 (Jan.), 60–78.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Trans. Softw. Engin. 23*, 5 (May), 279–295.

HOLZMANN, G. J. 2000. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 7th SPIN Workshop*, K. Havelund, J. Penix, and W. Visser, Eds. Lecture Notes in Computer Science, vol. 1885. 131–147.

HOWDEN, W. E. 1986. A functional approach to program testing and analysis. *IEEE Trans. Softw. Engin. SE-12*, 10 (Oct.), 997–1005.

HUISMAN, M., JACOBS, B., AND VAN DEN BERG, J. 1999. A case study in class library verification: Java's Vector class. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, A. M. D. Moreira and S. Demeyer, Eds. Lecture Notes in Computer Science, vol. 1743. 109–110.

JACKSON, D. AND VAZIRI, M. 2000. Finding bugs with a constraint solver. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*. 14–21.

JOHNSON, S. C. 1978. Lint, a C program checker. In *Unix Programmer's Manual, AT&T Bell Laboratories*.

KELLER, R. K., CAMERON, M., TAYLOR, R. N., AND TROUP, D. B. 1991. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering*. 208–218.

MANNA, Z. AND PNUELI, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

MARLOWE, T. J. AND RYDER, B. G. 1990. Properties of data flow frameworks. *Acta Infomat. 28*, 2, 121–163.

MASTICOLA, S. P. 1993. Static detection of deadlocks in polynomial time. Ph.D. thesis, Rutgers University.

MASTICOLA, S. P., MARLOWE, T. J., AND RYDER, B. G. 1995. Lattice frameworks for multiscore and bidirectional data flow problems. *ACM Trans. Program. Lang. Syst. 17*, 5 (Sept.), 777–803.

MASTICOLA, S. P. AND RYDER, B. G. 1991. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of Workshop on Parallel and Distributed Debugging*. 97–107.

MASTICOLA, S. P. AND RYDER, B. G. 1993. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 129–138.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall.

MÜLLER-OLM, M., SCHMIDT, D. A., AND STEFFEN, B. 1999. Model-checking: A tutorial introduction. In *Proceedings of Static Analysis, Sixth International Symposium*, A. Cortesi and G. Filé, Eds. Lecture Notes in Computer Science, vol. 1694. 330–354.

NAUMOVICH, G. AND AVRUNIN, G. S. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 24–34.

NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. 1999a. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*. 399–410.

NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. 1999b. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. 338–354.

NAUMOVICH, G., AVRUNIN, G. S., CLARKE, L. A., AND OSTERWEIL, L. J. 1997. Applying static analysis to software architectures. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, M. Jazayeri and H. Schauer, Eds. Lecture Notes in Computer Science, vol. 1301. 77–93.

NAUMOVICH, G. AND CLARKE, L. A. 2000. Classifying properties: An alternative to the safety-liveness classification. In *Proceedings of the 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 159–168.

NAUMOVICH, G., CLARKE, L. A., AND COBLEIGH, J. M. 1999. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*. 57–65.

NAUMOVICH, G., CLARKE, L. A., AND OSTERWEIL, L. J. 1996. Verification of communication protocols using data flow analysis. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 93–105.

NAUMOVICH, G., CLARKE, L. A., AND OSTERWEIL, L. J. 1998. Efficient composite data flow analysis applied to concurrent programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*. 51–58.

NETZER, R. H. B. AND MILLER, B. P. 1990. Detecting data races in parallel program executions. In *Proceedings of Languages and Compilers for Parallel Computing, Fourth International Workshop*. 109–129.

OLENDER, K. M. AND OSTERWEIL, L. J. 1990. Cecil: A sequencing constraint language for automatic static analysis gereration. *IEEE Trans. Softw. Engin. 16*, 3 (March), 268–280.

OLENDER, K. M. AND OSTERWEIL, L. J. 1992. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Engin. Method. 1*, 1 (Jan.), 21–52.

OSTERWEIL, L. J. AND FOSDICK, L. D. 1976. DAVE - a validation error detection and documentation system for Fortran programs. *Softw.–Prac. Exper. 6*, 4 (Oct.–Dec.), 473–486.

PELED, D. 1998. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, A. J. Hu and M. Y. Vardi, Eds. Lecture Notes in Computer Science, vol. 1427. 17–28.

RAMALINGAM, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22*, 2 (March), 416–430.

REPS, T. W., HORWITZ, S., AND SAGIV, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. 49–61.

RYDER, B. G. 1974. The PFORT verifier. *Softw.–Prac. Exper. 4*, 4 (Oct.–Dec.), 359–377.

SALCIANU, A. AND RINARD, M. C. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 12–23.

SCHMIDT, D. A. 1998. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. 38–48.

SCIENCE APPLICATIONS INTERNATIONAL CORPORATION. 1997. Advanced interoperability technology development: Investigating static data flow analysis for advanced distributed simulation verification. Tech. rep., SAIC. (May).

SMITH, R. L., AVRUNIN, G. S., CLARKE, L. A., AND OSTERWEIL, L. J. 2002. Propel: An approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering*. 11–21.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual*, 2nd ed. Prentice-Hall.

TAN, J., AVRUNIN, G. S., AND CLARKE, L. A. 2004. Heuristic-based model refinement for FLAVERS. In *Proceedings of the 26th International Conference on Software Engineering*. 635–644.

TAN, J., AVRUNIN, G. S., CLARKE, L. A., ZILBERSTEIN, S., AND LEUE, S. 2004. Heuristic-guided counterexample search in FLAVERS. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering* 201–210.

TAYLOR, R. N. 1983a. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informat. 19*, 1 (April), 57–84.

TAYLOR, R. N. 1983b. A general-purpose algorithm for analyzing concurrent programs. *Comm. ACM 26*, 5 (May), 362–376.

TAYLOR, R. N., BELZ, F. C., CLARKE, L. A., OSTERWEIL, L. J., SELBY, R. W., WILEDEN, J. C., WOLF, A. L., AND YOUNG, M. 1988. Foundations for the Arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. 1–13.

VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S.-J. 2000. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*. 3–12.

YEH, W. J. AND YOUNG, M. 1991. Compositional reachability analysis using process algebra. In *Proceedings of the 1991 Symposium on Testing, Analysis, and Verification*. 49–59.

YOUNG, M., TAYLOR, R. N., LEVINE, D. L., NIES, K. A., AND BRODBECK, D. 1995. A concurrency analysis tool suite for Ada programs: Rational, design, and preliminary experience. *ACM Trans. Softw. Engin. Method. 4*, 1 (Jan.), 65–106.

Received August 1999; revised January 2004; accepted August 2004