Flow Analysis for Verifying Specifications of Concurrent and Distributed Software *

Matthew B. Dwyer
Dept. of Computing and Info. Sciences
Kansas State University
Manhattan, KS 66506
dwyer@cis.ksu.edu

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

August 23, 1999

Classification

D.2.4 Software/Program Verification, D.1.3 Concurrent Programming, D.3.4 Processors

Abstract

This paper presents FLAVERS, a finite state verification approach that analyzes whether concurrent or sequential programs satisfy user-defined correctness properties. In contrast to other finite-state verification techniques, FLAVERS is based on algorithms with low-order polynomial bounds on the running time. FLAVERS achieves this efficiency at the cost of precision. Users, however, can improve the precision of the results by selectively and judiciously incorporating additional semantic information into the analysis problem.

The FLAVERS analysis approach has been implemented for programs written in Ada. We report on an empirical study of the performance of applying the FLAVERS/Ada tool set to a collection of multi-tasking Ada programs. This study indicates that sufficient precision for proving program properties can be achieved and that the cost for such analysis grows as a low-order polynomial in the size of the program.

1 Introduction

The application of distributed and concurrent programming technology has moved from special purpose database and operating systems into the programming mainstream. This movement is motivated by increasingly demanding system performance requirements. Incorporating concurrency in software, however, greatly complicates the problem of reasoning about the correctness of an application. To combat this, cost-effective analysis techniques should be developed to help developers gain confidence in the quality of their concurrent software. In this paper, we present such an analysis approach, called FLAVERS (FLow Analysis for VERifying Specifications), that uses cost-effective program flow analysis techniques to analyze user-specified

*This work was supported by the Advanced Research Projects Agency under Grant MDA972-91-J-1009, the Office of Naval Research under Grant N00014-90-J-1791, and the National Science Foundation under Grants CCR-9633388, CCR-9703094, and CCR-9708184.

correctness properties of concurrent programs. Although FLAVERS is applicable to a wide range of distribution and concurrency models, as well as sequential systems, in this paper we discuss analysis of programs with explicit tasking and rendezvous communication and illustrate our approach using Ada tasking programs.

With FLAVERS, developers define a set of program events that they want to reason about and specify properties of concurrent programs as patterns of those program events. They then choose whether the analysis should attempt to verify that all or no program executions satisfy the given property. At the core of FLAVERS is a polynomial-time, conservative flow analysis algorithm whose results can be used to address such questions. This algorithm operates on a graph model of concurrent program behavior that explicitly represents inter-task communication, synchronization, and statement ordering.

As we demonstrate, it is possible to perform very efficient flow analyses with FLAVERS. The utility of such analyses should be judged not by efficiency alone, however, but also by the precision of the results they produce. To overcome the traditional imprecision of static analysis, we have developed an approach for improving the precision of the results. The underlying principle of this approach is to conduct a series of analyses, where each individual analysis incorporates increasing amounts of information about executable program behavior as it is revealed to be needed. We use constraints to represent this semantic information, where such constraints typically provide information about the feasibility of particularly relevant paths in the program model. The series of analyses progresses from inexpensive analyses of, perhaps, limited precision to, perhaps, increasingly expensive analyses capable of providing higher levels of precision, until terminating after sufficiently precise results have been obtained.

Another approach used to improve performance and precision of FLAVERS' analyses is to refine the program model on which the analysis is based by eliding unnecessary information while retaining all the information necessary for verifying a specific property. A strength of FLAVERS is its ability to combine a collection of refinements and constraints so as to increase precision without making analysis time impractical. Other static analysis techniques might also profit by adopting a similar approach for specializing the program model and analysis algorithms.

A tool set that supports the analysis of finite-state specifications of Ada tasking programs has been constructed. The tool set, called FLAVERS/Ada, has been used to analyze a variety of distributed and concurrent Ada applications [Cha96], as well as individual components of concurrent applications [Dwy97], communication protocols [NCO96], and architectural specifications [NACO97]. It has also been applied in industrial settings for analyzing properties of advanced distributed simulation systems [BMD96] and for demonstrating the adherence of such systems to high-level architectural requirements [SAIC97].

FLAVERS makes a number of contributions to flow analysis and concurrency analysis research:

- it provides a polynomial-time concurrency analysis technique for proving user-specified properties,
- it introduces an incremental process though which analyses of increasing precision can be constructed, and
- it demonstrates, through empirical evaluation, that low-cost, high-precision flow analysis is practical for verifying concurrent systems.

To establish the context for our presentation, the next section describes related work and discusses similarities and differences between prior research and our approach. Section 3 follows with a high-level overview of the FLAVERS approach and introduces each of the major components of a FLAVERS analysis. We then focus on each of these components in turn. Section 4 presents the model of program behavior that serves as the basis for our flow analysis. Section 5 describes the formalism for specifying properties to be analyzed. Section 6 presents the flow analysis algorithm that lies at the core of FLAVERS' approach for checking the specification against the program model. In Section 7, we describe the constraint mechanism for improving performance and the precision of the analysis results. After having described the FLAVERS analysis approach, in Section 8 we present data gathered from applying the FLAVERS/Ada tool set to the analysis of a collection of concurrent Ada applications. Section 9 summarizes the major contributions and discusses future directions.

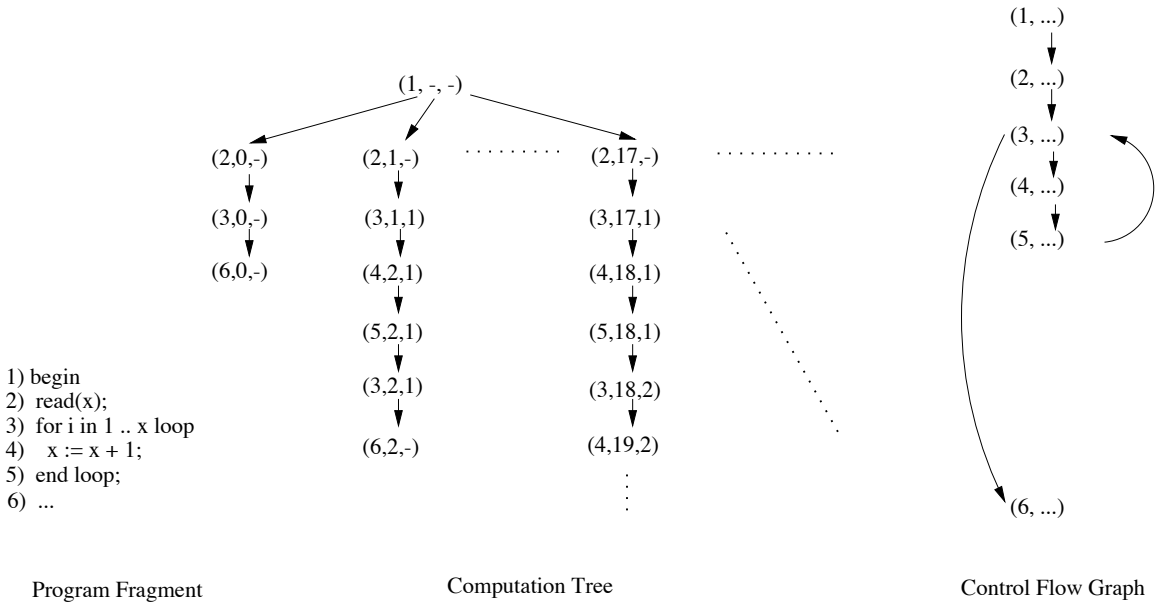

Figure 1: Program Models

2 Background and Related Work

FLAVERS applies flow analysis to reason about properties of sequential or concurrent software. It can be considered a finite state verification technique in that it attempts to prove properties without using the full power of theorem proving and instead reasons over a finite model of a program's behavior. Consequently, FLAVERS builds on and is related to work in flow analysis and finite state verification. We discuss the principles of static program analysis, in general, and then describe related work in these two areas in order to establish the context for our research.

2.1 Terminology

Static program analysis techniques attempt to determine truths about a program's behavior without executing the program. Such techniques can be considered compile time analyses and are independent of the execution of any particular test data. *Finite state verification* techniques are static analysis techniques that attempt to prove that the executable behavior of a program is consistent with a specification of its intended behavior. Finite state verification techniques are usually not as powerful as theorem proving techniques but can prove a wide range of interesting properties and are usually more tractable.

There are two primary components of any (finite state) verification analysis: a model of the executable program behavior and a specification of the intended program behavior or program property. Finite state verification techniques compare a finite model program model of the program's relevant executable behavior to a property specification to verify if the program's executions are consistent with this property.

Program models must accurately reflect the semantics of the implemented system. Most program models, however, are designed to approximate the system's semantics in order to educe their complexity and thereby enable efficient analysis. There are two kinds of information that a model captures: *control* and *data* states. A control state indicates the program operations which can be executed next, e.g., they can be thought of as program location counters. A data state indicates the current values of program variables. Approximations can be formed by projecting or abstracting the program's control and data states. One common class of models that we rely on in our work are control flow graphs (CFGs), which represent control states explicitly and, usually, do not store information about data states. To clarify the nature of the approximation in this

kind of model, consider Figure 1 which illustrates a program fragment, a part of its computation tree [Mil79], and a CFG for the fragment. Each node of the computation tree is a triple (ctl, x, i), where *ctl* is the statement number, x is the value of variable `x`, which for this example is assumed to be greater than or equal to zero, and i is the value of variable `i`. This example illustrates several important issues related to program models. For each path in the computation tree there exists a path in the CFG. Note, however, that many paths in the computation tree may map to the same path in the CFG and that there may exist paths in the CFG that do not correspond to computation tree paths. To illustrate this latter point consider a Boolean predicate that, because of previous computation, always evaluates to the constant true. The computation tree would have no paths where this predicate evaluates to false, but the CFG may represent both possible conditional edges emanating from the node representing the conditional predicate. Thus, the CFG approximates the computation tree by representing additional behaviors, which do not correspond to any program execution. Such spurious executions are called *infeasible* paths. We say that the CFG overestimates program behavior. There are other possible approximations. For example, a model could represent only some of the computation tree paths while containing no information not in the computation tree, thereby underestimating program behavior. Most finite state verification techniques, including FLAVERS, use models that overestimate the program behavior related to the property of interest. We say that such models are *conservative* or *safe* with respect to the set of all program behaviors related to a specified property.

The example CFG in Figure 1 eliminates all data components and preserves all control state components. Depending on the analysis, we may wish to encode partial data state information in the CFG. Alternatively we may only require partial control state information to perform an analysis. For example, for reaching definitions analysis we are interested only in read and write operations on data and do not need to represent the semantics of other program operations. This type of property-specific model can significantly reduce the size of the program model and thus reduce the cost of analysis while preserving information that is necessary for safe analysis.

Specifications are also often classified according to the kinds of properties they address. For example, a *reachability property specification* can be described as paths leading from the root of the computation tree to nodes whose state satisfies the required conditions. The best studied property of concurrent software, freedom from deadlock, is an example of a reachability property. Often, instead of specifying conditions for individual reachable states, a property specification is defined in terms of sequences of states or events that transition between states.

Specifications are often classified as describing safety or liveness properties. A *safety property* asserts that no prefix of a program execution enters a particular state (or contains a particular pattern of states or events). A *liveness property* asserts that a program execution should be able to continually enter a particular state (or repeat a particular pattern of states or events). Liveness properties apply to infinite program executions. A related, but finite, notion is that of a *bounded liveness property*. Such properties refer to regions of execution that are bounded by a pair of designated program states (or events), within which it must be possible to continually enter a particular program state (or repeat a particular pattern of states or events). To date, we have focused on using FLAVERS to verify properties of finite executions, such as safety and bounded liveness properties.

A wide variety of specification formalisms for describing the intended behavior of concurrent systems have been developed, such as temporal logics [Pnu85, CES86], automata-based formalisms [ABC$^+$91, Kur85, OO90], Petri nets [Mur89], and program algebras and calculi [BK84, Mil79]. Unfortunately, reasoning using the most general of these formalisms can be inefficient. Since a large, practical class of properties, including safety and bounded liveness, are captured by the much simpler theory of regular expressions and finite automata [DAC99], we will analyze specifications in this less general but more practical setting. In fact, if we are interested in arbitrarily long but finite program executions, such as executions that terminate, regular languages are as expressive as temporal logics [Wol83].

Limiting properties to this restricted (albeit useful) class, one can determine a conservative program model and formulate a conservative analysis over that model. For such analyses a positive analysis result implies that the specified property holds for the program; we call such a result *conclusive*. Conservative analyses do not provide for conclusive negative results. If an analysis says that a specified property fails to hold then

either the property fails to hold for the program or for some infeasible behavior in the program model. These negative results are termed *inconclusive*. While such results might be undesirable, there is no alternative to reasoning about approximate program models since the computation tree is much too large to construct in general.

2.2 Flow Analysis of Sequential Software

Program flow analysis was originally developed to enable optimizing compilers to generate efficient code [ASU85]. In flow analysis, specific patterns of behavior are analyzed by computing the fix-point of a relation defined over CFG nodes. While a CFG typically encodes only simple control information, the relations can encode additional data and control information. Examples of patterns of behavior include classic compiler optimization analyses such as reaching definitions, live variables, available expressions, and constant propagation [ASU85], and analyses that support validation and verification [FO76, How86, OO92]. Data flow frameworks [Hec77, MR90] are one means of formulating a specific flow analysis. For the broad class of problems with monotone bounded frameworks, convergence of the fix-point computation is guaranteed by use of general iterative solution algorithms [Hec77]. We use a bounded monotone framework in the analysis at the core of FLAVERS and describe this algorithm in Section 6.

FLAVERS uses an approach that combines multiple flow analyses into a single analysis [CC95, HR81, WZ91]. Combining multiple dependent flow analyses into a single flow analysis offers the potential to improve the precision of analysis results, for example, by eliminating some infeasible paths. In some cases the combined problem is actually less costly to execute than the constituent problems. Our combined analysis, based on finite-state automata, provide a simple approach to constructing qualified flow analyses [HR81] that has been effective in practice.

Flow analysis techniques for validation were originally applied to detect potential anomalies in sequential code in the Dave system [FO76]. This approach used a fixed set of flow analyses formulated to detect a fixed set of anomalies. The Cecil/Cesar system [OO90, OO92] generalized this approach by providing a single parameterized flow analysis, called state propagation analysis, that could be instantiated to detect a broad range of anomalies, including user defined anomalies. FLAVERS builds on this work by extending the analysis to concurrent programs and by incorporating a variety of mechanisms to increase the precision of the analysis results.

2.3 Concurrency Analysis

Most conservative concurrency analysis approaches assume an interleaving model of execution, where all possible orderings of a program's actions are taken into account. The computation tree is an interleaving model, but as noted, it is too large to be used as the basis for analysis. Reachability graphs, which can be derived by explicitly modeling all the syntactically correct interleaved traces through a collection of CFGs [Tay83b], are one of the most commonly used program models [Hol88, SMBT90, YTL+95]. Unfortunatley, the size of the reachability graph and consequently the cost of searching the graph increases exponentially with program size[Tay83a]. As with a CFG, we can choose to model components of the program data state explicitly in the reachability graph, but this will further increase the size of the reachability graph.

To increase the applicability of reachability analysis, researchers have investigated a variety of techniques including: reducing the state space based on the property being analyzed [DBDS93, GW91, Val90], building and analyzing the state space compositionally [CPS93, YY91], and using a symbolic representation of the state space [McM93]. Although none of these techniques have been able to reduce the worst-case complexity of analysis to be sub-exponential, each technique has been successfully applied to selected programs.

An alternative line of research has focused not on reasoning explicitly about the program state space, but on reasoning about necessary conditions for sequences of program actions to be contained in a program execution. Using such conditions, this analysis formulates a proof-by-contradiction by attempting, and failing, to prove the negation of a property. INCA [ABC+91] uses this approach by encoding the negated property and necessary conditions as linear inequalities whose solutions determine satisfaction of the property.

While this technique has exponential bounds on its running time, it has been successfully applied to a number of programs and allows for practical analysis of very large versions of those programs [Cor96].

Model-checking approaches [CES86, Hol97] view program executions as sequences of states that correspond to interpretations of a finite set of propositions. A program property is specified as a temporal logic formula. The model-checking algorithm systematically computes the set of sub-formula of the specification that are valid for the program, thereby arriving at a determination of the validity of the specification. The performance of early model-checking approaches suffered because the set of state sequences grows exponentially with program size. More recently the basic approach of model-checking has been adapted to symbolic program models, which often yield smaller state space representations but are still exponential in the worst-case [McM93].

While the bulk of flow analysis research has been aimed at sequential software, interesting recent work has applied flow analysis to the evaluation of concurrent software [CKS90, DS91, GS93, Mas93, RS90]. Most of this work has been directed at the analysis of a restricted class of behaviors, such as deadlock, and does not require representations of interleavings among the actions in separate tasks. These analyses use flow graph models that are, essentially, a collection of task CFGs with additional edges, or labels, to represent inter-task synchronization and communication. These include analyses to detect the potential for statements to execute concurrently [CKS90, DS91, MR93, Mer92, NA98, NAC99b], data races [TO80, NM90], reaching-definitions [GS93], and dead-code [RS90, CK93]. Knoop et. al. [KSV96] have shown that for a class of rapid bit-vector problems, a flow graph without explicit interleavings is conservative.

For more general analysis, however, interleavings need to either be modeled in the graph or incorporated into the analysis algorithm. Interleavings occur, implicitly, when program tasks asynchronously execute local actions. In representing a sequence of actions executed across a collection of tasks, we must account for the fact that tasks may execute at different speeds or that scheduler decisions may cause different orderings of actions among the tasks. For a flow graph to model the set of all executions of a concurrent program conservatively, it must explicitly represent all possible such interleavings, or the analysis algorithm must be modified to derive and use this information. The flow graph model used in FLAVERS, and described in Section 4, explicitly but compactly represents these interleavings through the use of additional edges.

In their approach for checking deadlock freedom in Ada tasking programs, Masticola and Ryder [Mas93] incorporate an approach for refining the program model that is similar in spirit to the refinements described in Section 4. In their approach, they use flow analyses [MR93] to refine the program model prior to the final analysis, thereby increasing the precision of the analysis results. Our approach differs from that of Masticola and Ryder in that we advocate selective application of a more comprehensive set of refinements, apply refinements only a single time, further improve precision through the use of feasibility constraints, and support analysis of a wide class of properties as opposed to just deadlock.

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. At an abstract level, Steffen [Ste93] and Schmidt [Sch98] have shown that flow analyses can be formulated as model-checking problems and vice versa. Practically speaking, model checking techniques depend on the analyst coming up with an accurate and relatively concise model of the program. Traditionally, flow analyses have relied on a systematic mechanism for abstracting program information. If it is determined that this abstraction does not lead to analysis results that are sufficiently precise for the property being evaluated, then FLAVERS provides a mechanism for incorporating additional information into the representation that is intended to increase the accuracy of the results. Similar approaches have also been investigated for model-checking techniques [DS97, DP98].

FLAVERS is an application of flow analysis research to the verification of properties of concurrent (and sequential) systems. At its core is an extension to the state propagation analysis proposed by Olender and Osterweil [OO90], over a novel program model that explicitly and compactly represents interleaving. The FLAVERS tool set is designed to provide the ability to combine different flow analyses. These combined analyses can be incorporated into state propagation analysis to improve the overall precision of the analysis results. This flexible, incremental approach to analysis allows FLAVERS to use an efficient polynomial-time flow analysis to address concurrency analysis questions that, to date, have required exponential-time techniques.

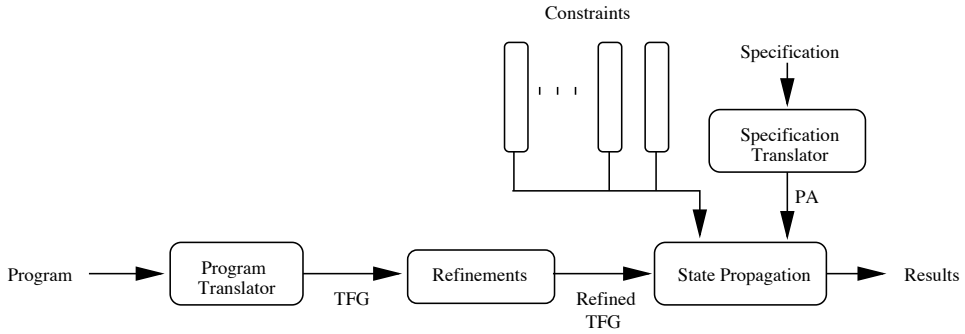

Figure 2: Architecture of FLAVERS

3 Overview

Figure 2 illustrates the major components in the FLAVERS architecture. Analysis with FLAVERS consists of the following steps:

- Extracting a model of executable behavior from a given program. This model is called a *Trace Flow Graph* (TFG).
- Refining the model to decrease its size and to more accurately reflect executable program behavior.
- Constructing a representation of the specification to be checked. This representation is called a *Property Automaton* (PA).
- Executing the state propagation flow analysis algorithm with the model and behavior representation as inputs.
- Presenting the results of that algorithm to the user.
- If necessary incorporating constraints into the analysis and re-executing.

The analysis results may demonstrate conclusively that the specification is consistent with the program. Alternately, the results may fail to provide such conclusive information; this can happen either because a fault in the program causes some executable behavior to be inconsistent with the specification or because the analysis is imprecise. In the former case, if a program fault is apparent it can be fixed and the analysis can be rerun. In cases where the program fault is not obvious or where imprecision is the cause of an inconclusive analysis result, incorporating constraints can be helpful. This last step of incorporating additional constraints can be performed multiple times, thereby providing an incremental approach for increasing precision and analysis cost. In the remainder of this section we briefly describe each of the components of the FLAVERS architecture.

Modeling Program Executions

Analysts may wish to reason about the execution of an application from a variety of perspectives. For example, they may be interested in determining whether some statement in a program uses a variable whose value is uninitialized; in this case they are interested in the definitions and uses of program variables. They may be interested in determining that whenever a read or write operation on a file abstraction is called, the file has been opened and will subsequently be closed; in this case users are interested in events that correspond to calls on the operations of the file abstraction. To address this variation in user focus, FLAVERS allows users to define *events*. Events are observable, indivisible program actions, such as the definition or use of a

variable, that are of interest to the analysis at hand[1]. Indivisible is defined with respect to the events. Thus, a complex expression could be defined as a single event if no other events of interest can be interleaved with this expression. The events are used to construct the model of program behavior as well as to formulate the properties.

To reason about the set of possible program executions we need a semantically well-founded model of those executions. This model need not represent all the details of the program executions, but it must represent sufficient information to support the desired conservative analyses. For FLAVERS this implies that the TFG program model must represent all sequences of program events that correspond to feasible program executions. Thus, a node in the TFG either represents important flow of control information or represents some program action that corresponds to an event of interest, or both. Hence, a statement in a program need not be modeled or it may be modeled by multiple nodes depending on the events of interest.

There are a variety of mechanisms for defining the mapping between program actions and events. For example, FLAVERS automatically defines some events, such as inter-task communication, and lets analysts define others through the use of stylized comments embedded in the program text.

To assure conservativeness, the TFG may overestimate the executable sequences of program events, and in doing so may include some infeasible sequences. The refinements in Figure 2 can be used to eliminate some of these sequences as well as to reduce the size of the TFG and thereby reduce the cost of the analysis. Refinements are similar in spirit to the analyses and transformations of the internal representations used in optimizing compilers. FLAVERS, however, is not constrained by a desire to produce results in a matter of seconds. Thus, FLAVERS can apply more sophisticated refinements than might be considered in a compiler in order to increase the precision of the model and the results of subsequent analyses.

Property Specifications

Properties are specified as sequences over the set of observable events. In the current implementation of FLAVERS, such sequences are specified as regular expressions. These regular expressions are translated into the PA, a deterministic finite state automaton (DFSA). In practice, any formalisms that can be converted to a DFSA, for example graphical interval logic [DKM+94], could be used to describe properties in FLAVERS. Although fundamentally limited to expressing arbitrarily long, but finite, sequences of events, we have found that a wide range of interesting properties of concurrent programs can be efficiently represented as DFSAs. In addition to specifying sequences of events, the user must indicate to FLAVERS whether to check that the property characterizes all program executions or, alternatively, whether it characterizes no program execution.

State Propagation Analysis

FLAVERS castes the question of whether program behavior is consistent with a property as a state propagation flow analysis problem [OO90]. We have adapted this algorithm to apply it to concurrent programs and have extended it to incorporate constraints. This algorithm avoids enumerating all feasible sequences of program events by collapsing them into equivalence classes based on the structure of the property being analyzed. The result is an analysis whose complexity grows as a polynomial in the number of TFG nodes and that computes a conservative answer to the analysis question. Conclusive analysis results provide as high-assurance as any other formal verification method. Inconclusive analysis results provide information about the source of the program faults or about imprecision in the program model that should be eliminated in order to improve the precision of the results.

[1]For the purpose of this paper, we assume that all subprogram invocations are expanded inline and that program tasks are defined statically. Thus, procedure invocations and returns may be marked by distinguishing events.

Increasing Precision

The state propagation algorithm trades a reduction in precision for a reduction in the cost of analysis. A distinguishing feature of FLAVERS, and one of its major strengths, is the ability to modify this cost-precision tradeoff. To gain efficiency the program model and state propagation algorithm over-approximate the event sequences of the system under analysis. Constraints that encode semantic information about program data and control states can be added to sharpen this approximation. This allows the state propagation algorithm to avoid considering some infeasible event sequences. It is important to note that FLAVERS is capable of producing conclusive analysis results even when some infeasible sequences remain in the model. Constraints are only necessary when there is an infeasible sequence that is inconsistent with the property being analyzed; elimination of those sequences will enable conclusive results to be produced.

4 A Model of Distributed Program Execution

In principle, a program may manipulate data that can range over an infinite set of values. Thus, in general, the precise execution behavior of a software system cannot be captured by a finite-state model. To enable efficient analysis of complex software we use an abstract finite-state model of program execution behavior constructed in such a way that information that is relevant to a desired analysis is preserved in the model. In this section we describe the TFG model, demonstrate that the model is small and easy to construct, and show that it supports conservative state propagation analysis. Since the TFG over-approximates the set of possible sequences of program events that could occur in a program execution, we also discuss refinements that can improve the precision of this over-approximation.

4.1 Denoting Program Behavior

In order for FLAVERS to check a specification against modeled program behavior, both must be defined using a common semantic foundation. We do this by defining a common set of symbols to which program actions can be mapped and with which specifications will be formulated.

Definition 1 *An* **event alphabet**, $\Sigma = \{\tau, a, b, c, \ldots\}$, *is a finite set of symbols, where each symbol,* $a, \ldots$, *in* Σ *models a unique program action. The symbol* τ *is an element of all event alphabets; it stands for the occurrence of some program actions that are not explicitly modeled for a given analysis.*

The alphabet defines the scope and granularity for both the program model and the property specification. Users identify the actions of the program that they are interested in reasoning about and specify an event name for each that is then bound to each node in the TFG that represents such an action. For simplicity in the state propagation algorithm, we require that each node has only one event associated with it; thus the granularity of the nodes is determined by the granularity of the actions associated with an event. In this setting, the TFG defines a language of strings over Σ where each string corresponds to a possible program execution. We refer to such strings as program traces and the alphabet over which the strings are defined is either stated explicitly or can be inferred from the context. Similarly, specifications define a language of strings over Σ.

4.2 Trace Flow Graph

TFGs are defined in terms of CFGs for each task in the program We assume that a CFG for a task is a conservative (or safe) approximation of the possible sequences of actions that a task could execute [MR90]. A CFG is assumed to be defined such that the action associated with a node immediately precedes the action associated with another node if there exists a control flow edge from the first to the second. More generally an action may precede another if there is some path from the first to the second. To verify a property about events, state propagation analysis must consider all possible orderings of those events as allowed by the program. A CFG, where each node is annotated with the associated event, provides a program model that

is adequate for analyzing sequential programs. For concurrent and distributed programs this is not the case, since nodes may be ordered both within a single task, as in a CFG, and across task boundaries. A TFG is formed from the collection of CFGs for all the tasks in the program, with the addition of nodes to represent inter-task synchronization actions and edges to represent the inter-task orderings between nodes in different tasks. We do this in such a way that the annotated TFG for a program represents each possible sequence of events that could occur in the program as a path in the graph. Conceptually, the TFG is defined as:

- a forest of CFGs, one for each task in the program
- additional nodes to represent inter-task synchronization and communication actions
- additional edges to represent inter-task event orderings

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

Task CFGs

For simplicity we assume that distributed systems are made up of tasks, executing as separate threads of control. A task, T_i, has a CFG, $(N_i, S_i, F_i, E_i, L_i)$ where each node N_i represents an action[2], S_i and F_i are elements of N_i that represent the unique nodes associated with the first and last actions in the task, respectively, $E_i \subseteq N_i \times N_i$ are edges describing that on some potential execution the action associated with the first node immediately precedes the action associated with the second, and L_i relates a node N_i to the symbol denoting a program action. The TFG is formed from the collection of CFGs so that the *local nodes* and *local control edges* of a TFG are the union of all the CFG nodes and the union of all of the CFG edges, respectively. The label associated with a TFG node is the same as the label associated with the corresponding CFG node if the action is an element of the event alphabet, and τ otherwise.

Inter-task Nodes

For concurrent and distributed software it is often necessary to model the synchronization of multiple program tasks. For example, with the Ada rendezvous, a pair of tasks may execute *matching* communication statements in the form of a call and an accept of a specific task entry. In TFGs we model all potential synchronization actions by adding a *communication* node between two matching synchronous interactions and labeling it with an interaction event. For each pair of matching synchronous interactions, $x \in N_j$ and $y \in N_i$ where $i \neq j$ and $L_j(x)$ and $L_i(y)$ are symbols for matching interactions, for example $call_A$ and $accept_A$, we add $c_{\{x,y\}}$ to N, the set of TFG nodes, with $L(c_{\{x,y\}})$ defined as a symbol for the synchronization event, for example A. We refer to the collection of all $c_{\{x,y\}}$ nodes as C_{nodes} and the union of the label relations for those nodes as $L_{C_{nodes}}$. In addition to the C_{nodes} we define nodes $S = n_{initial}$ and $F = n_{final}$ to represent the initial and final events in the execution of a distributed program built from the individual tasks. We define $L(n_{initial}) = L(n_{final}) = \tau$ for these new nodes, i.e., $(n_{initial}, \tau)$ and (n_{final}, τ) are in the TFG label relation.

Inter-task Edges

The communication nodes are connected to interacting tasks by introducing additional edges into the TFG. For each node $c_{\{x,y\}} \in C_{nodes}$, replace edge $(x, x') \in E$ with $(x, c_{\{x,y\}})$ and $(c_{\{x,y\}}, x')$ and replace edge $(y, y') \in E$ with $(y, c_{\{x,y\}})$ and $(c_{\{x,y\}}, y')$; we denote the replaced edges with R and the added edges with C_{edges}. These edges introduce flow graph paths that express the ordering of events that either immediately precede or follow synchronization activities in tasks.

[2]We assume that the granularity of the nodes in a task CFG depends on the actions associated with the event alphabet. For example, if evaluation order of arguments for a procedure call are relevant for the analysis, then the CFG must represent the legal evaluation orders explicitly.

For each task CFG, i, we add edges, $(n_{initial}, S_i)$ and (F_i, n_{final}), to E to connect the global initial and final nodes to those for the individual tasks.

Up to this point the edges in the TFG have not explicitly represented the possible ordering of events in different tasks that may execute concurrently. To address this need, TFGs include *may immediately precede* (MIP) edges.

Initially, we assume that all pairs of nodes in *different* tasks require a MIP edge between them. Communication nodes are considered to belong to both the tasks that they are synchronizing, and thus MIP edges would not be created between a communication node and any of the nodes in either of the two associated tasks. Similarly, the initial and final program nodes are considered to belong to all tasks and therefore are not associated with any MIP edges. Thus, for every pair of nodes in different tasks, $x \in N_i$ and $y \in N_j$ where $i \neq j$, a MIP edge (x, y) is added to the TFG. For each communication node $c_{\{x,y\}}$ with $x \in N_i$ and $y \in N_j$ a MIP edge is added between this communication node and all TFG nodes not in N_i or N_j. We denote the set of all MIP edges by M. MIP edges capture the pairwise interleaving of events executing in different tasks. Explicitly encoding interleavings of pairs of events, rather than of sequences of events, enables construction of a model that is polynomial in the number of modeled events, rather than exponential. This savings in model size comes at the expense of precision, since the pairwise interleavings may result in the introduction of infeasible inter-task event orderings.

Summary of TFG Definition

We have defined the TFG as a flow graph whose nodes are the nodes of each individual task CFG with additional synchronous interaction, initial and final nodes. The edges of the TFG preserve the local control flow edges of the individual CFGs but also include edges to explicitly represent synchronization and interleaving.

Definition 2 *Assume that a program is composed of tasks, where each task, T_i, has a CFG, $(N_i, S_i, F_i, E_i, L_i)$ where each node N_i represents an action, S_i and F_i represent the unique nodes associated with the first and last action in the task,respectively, $E_i \subseteq N_i \times N_i$ are edges, and L_i relates a node in N_i to a symbol denoting a program action. Also assume that the set of events Σ is a subset of the actions labeled in the set of CFGs. A* **trace flow graph** *is a labelled directed graph (N, S, F, E, L) where:*

$$
\begin{aligned}
N &= C_{nodes} \cup \{n_{initial}\} \cup \{n_{final}\} \cup \bigcup_i N_i \\
S &= n_{initial} \\
F &= n_{final} \\
E &= \bigcup_i \{(n_{initial}, S_i)\} \cup \bigcup_i \{(F_i, n_{final})\} \cup \bigcup_i E_i \cup M \cup C_{edges} - R \\
L &= L_{C_{nodes}} \cup \{(n_{initial}, \tau), (n_{final}, \tau)\} \cup \bigcup_i L_i
\end{aligned}
$$

Concatenating the symbols labelling the nodes visited along the path yields a trace of program events. The collection of event traces for all such TFG paths approximates, by overestimation, the set of possible sequences of events that can occur during program execution.

4.3 An Example and its TFG

Figure 3 illustrates a task that operates as a binary semaphore; it presents two communication entries `P` and `V`. The state of the semaphore is maintained by the control state of the task, rather than by a local Boolean variable. There are two clients that signal the semaphore by way of entry calls onto `P` and `V`. Having acquired the semaphore the clients proceed with some local work, modeled here as the program events `work1` and `work2`.

```
task body Semaphore is
begin
  loop
    exit when done;
    accept P;
    accept V;
  end loop;
end Semaphore;
```

```
task body Client1 is
begin
  loop
    exit when done;
    Semaphore.P;
    null; -- Events[work1]
    Semaphore.V;
  end loop;
end Client1;
```

```
task body Client2 is
begin
  loop
    exit when done;
    Semaphore.P;
    null; -- Events[work2]
    Semaphore.V;
  end loop;
end Client2;
```

Figure 3: Ada Tasks for Semaphore Example

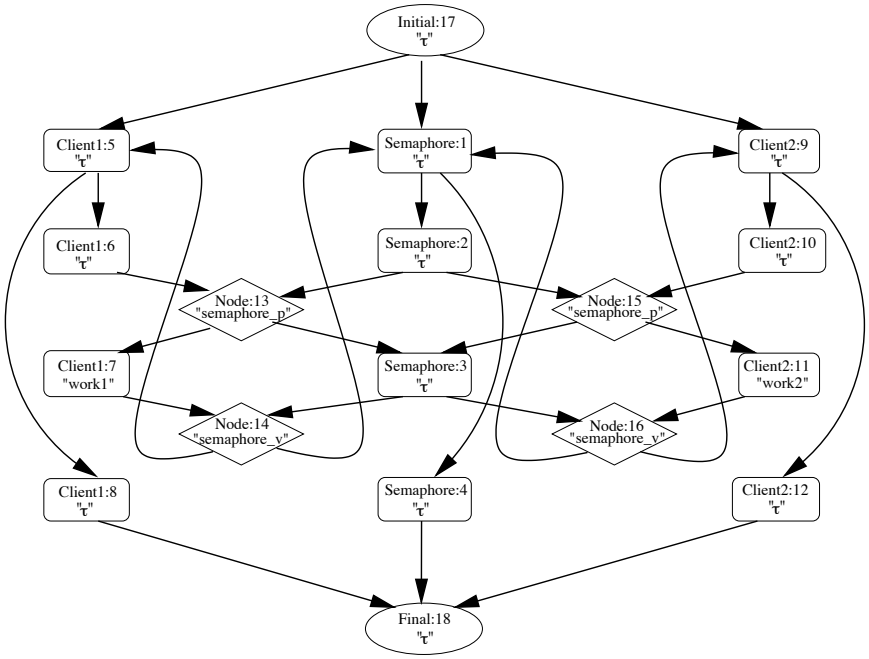

Figure 4: Trace Flow Graph for Semaphore Example

Assuming we are interested in properties that are concerned with the order in which the semaphore's entries can be called and work can occur, there are four program events of interest: `Semaphore.P`, `Semaphore.V`, `work1`, and `work2`. The TFG for the example is illustrated in Figure 4 with MIP edges elided to make the figure more readable. Communication nodes are depicted as diamonds and labelled with "node", node number and the communication event[3]. Local nodes are depicted as rounded rectangles, and each is labeled with the name of the task it is associated with, its node number, and its associated event. Ovals represent the initial and final nodes, labeled as such along with their assigned node number and τ event.

In the semaphore example, the inter-task nodes are the initial and final nodes, numbered `17` and `18`, and the communication nodes, numbered `13` thru `15`. The other nodes are local nodes, and thus, each is associated with one task.

Naive construction of MIP edges between all pairs of nodes in different tasks would require 128 such edges for this example. Figure 5 illustrates the 10 (dashed) MIP edges that occur emanating from node 11. As we discuss later in this section, we can eliminate MIP edges in cases where we can prove that the pairwise orderings are either infeasible or are captured by some other path through the TFG.

[3]The FLAVERS/Ada toolset converts identifiers to lower-case and replaces "." with "_", to eliminate conflicts with the "." operator in the regular expression syntax defined in Figureřeffig:qre-syntax.

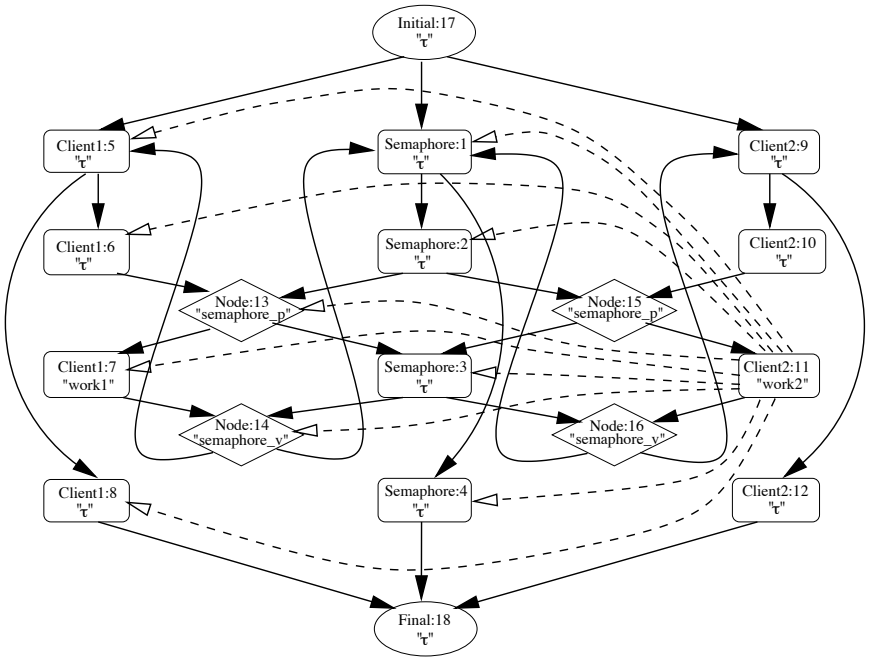

Figure 5: Semaphore TFG with Node 11's MIP Edges

4.4 Constructing A TFG

Construction of a TFG from the collection of task CFGs for a system proceeds in three phases. The first phase takes a forest of task CFGs as input and creates an isomorphic graph where the CFG actions are mapped into the corresponding event alphabet Σ. The nodes at this point are the local nodes in the TFG. In the second phase, pairs of local nodes that correspond to entry calls and accepts on the same task entry are identified. For each such pair, a communication node is created with incident edges connecting it to the appropriate local nodes. These local nodes are re-labeled with τ, since their joint execution is now captured by the label on the communication node. The third, and final, phase creates MIP edges between all TFG nodes in different tasks. The full details of this construction algorithm are given in [Dwy95].

4.5 Conservativeness of the TFG Model

Conservative state propagation analysis requires that the TFG be conservative with respect to all executable sequences of program events. To satisfy this requirement, we could produce a program model that includes a path for each interleaving of the individual task CFG paths. While conservative, such a model would contain many paths that are infeasible with respect to inter-task communications, since according to the semantics of the rendezvous communication mechanism, a task blocks at communication action until the corresponding communication action occurs in another task. Consequently, a simple interleaving of CFG paths may be inconsistent with these communication semantics, e.g., by allowing a task's path to pass through an entry call without a a corresponding accept occuring in the called task. Thus, we define a **communication consistent path** to be a path that adheres to the synchronization semantics of the language being modeled.

Definition 3 *A* **communication consistent path** *for Ada is an interleaving of a set of CFG paths, one for each CFG in the system, such that for each task T_i each occurrence of a node associated with an entry call (accept) in T_i has at most one corresponding accept (entry call) occurrence in some task T_j, $j/= i$, and, assuming that this entry call (accept) node occurs on the path before the corresponding accept (entry call) node, no nodes in T_i occur on the interleaved path following this entry call (accept) node until after the corresponding accept (entry call) node occurs on the path.*

This allows the possibility of indefinite blocking at a communication statement, thereby modeling deadlocking executions, while disallowing a single occurrence of a call (accept) to match multiple accepts (calls), which is semantically impossible.

We prove that for all communication consistent paths, for a given set of CFGs, that the TFG built from those CFGs is a conservative model of the possible sequences of those program actions associated with events. Note that for a communication consistent path, we can determine the path associated with each CFG participating in that path. Also note that a CFG has actions associated with its nodes but the TFG is only interested in a subset of those actions, namely those that map to events. (Remember that a TFG node associated with a CFG node labeled with an action not associated with an event is labeled with τ.) Thus, for simplicity we refer to the event sequences associated with a communication consistent path, with a CFG path derived from a communication consistent path, or with a path in the TFG. Since all CFG nodes have a corresponding node in the TFG with a corresponding label (i.e., an event or τ), the question of path correspondence can be reduced to a question of the existence of edges that allow appropriate sequencing of nodes.

For the proof, we need to record information about the unmatched communication statements encountered so far as we traverse a communication consistent path. We define $Unmatched(T_i)$ to be the unmatched communications in the communication consistent path for task $T_i$if one exists, and *null* otherwise[4].

Theorem 1 (TFG Safety)
For each communication consistent path in a given set of CFGs, there exists a path through the TFG where, the order of non-τ labelled nodes on the TFG path and communication consistent path is the same.

Proof:

We have, by TFG construction, that there exists a MIP edge between TFG nodes in different tasks. Let, $Tnode|\bigcup_i N_i \rightarrow N$ map each CFG node to the TFG node that was constructed to model it.

The proof is by induction on the length of the communication consistent path. We prove not only that the theorem holds, but also that the last node added to the communication consistent path corresponds to the node that extends the TFG path.

Base Step: Path of length 1

There are as many possible paths of length 1 as there are tasks in the program, one for the start node of each task CFG. The communication consistent path $n_{initial}, Tnode(S_i)$ is a path of length 1 that includes the initial and start node of task T_i; this path is also present in the TFG. S_i and $Tnode(S_i)$ correspond by definition.

Inductive Step: Path of length k

Given a communication consistent path of length k, by hypothesis there exists a TFG path, call it p^k_{TFG}, that contains the same sequence of non-τ events as the given path.. Let, T_n denote the identity of the task which contains the last node, n_{last}, on the communication consistent path.

While there are many possible extensions of the communication consistent CFG path, these extensions can be grouped into three cases.

Next node is local to T_n This case includes all control flow successors of n_{last} that are not entry calls or accept statements. For any such successor, s, the desired TFG path is p^k_{TFG} extended with an edge from $Tnode(n_{last})$ to $Tnode(s)$. Such an edge is guaranteed to exist because each task CFG is logically embedded in the TFG.

[4]For clarity we consider the possibility of a single unmatched communication statement per task. This definition, however, can be extended to account for the possibility of multiple unmatched statements arising from Ada select statements

Next node is local to task other than T_n This case accounts for extension of the communication consistent path to a task other than T_n. Let s be the CFG node that forms this extension. The desired TFG path is p^k_{TFG} extended with a MIP edge from $Tnode(n_{last})$ to $Tnode(s)$. Since there is a MIP edge between all nodes in different tasks in the TFG, such a MIP edge is guaranteed to exist.

Next node is a communication action This case includes all control flow and MIP successors of n_{last} that are entry calls or accept statements. Let s be the CFG node for the call or accept and the task containing that node be T_s. Note that for a communication statement $Tnode(s)$ is a local τ-labelled TFG node rather than a node added to model a joint synchronous communication.

For an entry call, $T_i.E_j$, there are two possibilities:

$Unmatched(T_i)$ **is an accept statement for** E_j In this case, there exists a TFG communication node $c_{\{s,Unmatched(T_i)\}}$ whose unique predecessor in T_s is $Tnode(s)$. The new TFG path is p^k_{TFG} extended with two edges: a control flow edge (if $T_s = T_n$) or MIP edge (if $T_s \neq T_n$) from n_{last} to $Tnode(s)$ and control flow edge from $Tnode(s)$ to $c_{\{s,Unmatched(T_i)\}}$. We set $Unmatched(T_i)$ to $null$.

otherwise In this case, the new TFG path is p^k_{TFG} extended with one edge: a control flow edge (if $T_s = T_n$) or MIP edge (if $T_s \neq T_n$) from n_{last} to $Tnode(s)$. We set $Unmatched(T_s)$ to s.

For an accept statement on entry E_j, there are also two possibilities:

There exists a task, T_i, where $Unmatched(T_i)$ is an entry call on $T_s.E_j$ In this case, there exists a TFG communication node $c_{\{Unmatched(T_i),s\}}$ whose unique predecessor in T_s is $Tnode(s)$. The new TFG path is p^k_{TFG} extended with two edges: a control flow edge (if $T_s = T_n$) or MIP edge (if $T_s \neq T_n$) from n_{last} to $Tnode(s)$ and control flow edge from $Tnode(s)$ to $c_{\{Unmatched(T_i),s\}}$. We set $Unmatched(T_i)$ to $null$. Note that this case handles the situation where there are calls in several tasks blocked on entry $T_s.E_j$.

otherwise Same as the otherwise case for entry calls.

All of the TFG edges used to extend the TFG path are guaranteed to exist by the definition of the TFG. Furthermore, the final nodes in the p^k_{TFG} are the corresponding nodes in the extended communication consistent path. For the matched communication statements, they are labelled TFG communication nodes, and for unmatched communication statements they are τ-labelled local TFG nodes.

□

This theorem implies that each possible communication consistent interleaving of CFG paths is represented in the TFG; in general, the reverse is not true. There exist paths in the TFG that do not correspond to executable interleavings of CFG paths.

4.6 Complexity of the TFG Model

In this sub-section, we describe the worst-case time complexity of the TFG construction algorithm and the worst-case space complexity of the TFG itself. Ideally we would like to choose a measure for expressing these complexities that is closely related to natural measures of the program, for example, number of tasks, number of statements, number of entry call and accept statements. The TFG, however, is dependent on Σ. The number of events we are interested in for a given program may vary widely depending on the specification we are attempting to reason about. For this reason, we express complexity in terms of numbers of program statements with the reasonable assumption that the number of program events per statement can be bounded by some constant.

Theorem 2 (Size of TFG)
The number of nodes in a TFG, N, is $O(S^2)$ where S is the number of statements in the modeled program. The number of edges in a TFG is $O(S^4)$.

Proof:

Let c be the maximum number of events per program statement, S the number of statements, C the number of entry call and R the number of accept statements. The number of nodes in the TFG is $O(2 + c(S - (C + R)) + c(C * R))$. This is the sum of the initial/final nodes, the nodes from modeled CFG nodes and the communication nodes. In the worst case, if $C = R = S/2$ the bound on N becomes $O(S^2)$.

A TFG has at most a single (uni or bi-directional) edge between any pair of nodes. Therefore, the number of TFG edges is in the worst case $O(N^2) = O(S^4)$.

□

We note that the product of entry calls and accepts 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 program. Our experience has been that the number of TFG nodes is usually linear in the number of program statements in practice.

Theorem 3 (Complexity of Construction Algorithm)
The time required to construct a TFG from a collection of CFGs is $O(S^4)$, where S is the number of program statements.

Proof:

Phases 1 and 2, described in Section 4.4, are $O(S^2)$ since there are at most $(S/2)^2$ communication nodes. For each pair of TFG nodes we potentially create a MIP edge. From Theorem 2 the number of TFG nodes is $O(S^2)$. Thus, phase 3 requires $O(S^4)$ operations and its cost dominates the other phases.

□

We note that it is often the case that N is much less than S since many of the statements in a program are irrelevant for the purposes of reasoning about a particular specification. Furthermore, as mentioned above, in practice communication statements are a small minority of program statements. Together these factors typically make the cost of constructing a TFG linear in the number of program statements.

4.7 TFG Refinements

The proof of Theorem 1 shows that TFG paths include the event orderings that can occur in all potential program executions. These paths may also include orderings that are not executable. There are three major sources of inaccuracy in the TFG: unexecutable control flow paths, unexecutable intra-task communication, and unexecutable orderings of asynchronous program events. Imprecision related to asynchronous event orderings occurs because of the presence of MIP edges, which introduce infeasible paths in the TFG. Figure 6 illustrates a TFG fragment with some (solid) control flow edges and two (dashed) MIP edge. These MIP edges represent potentially executable sequences of events of the form $\ldots ab \ldots$ and $\ldots ba \ldots$. If we assume that neither node `3` in `Task1` nor node `7` in `Task2` are nested in a control flow loop, then at most a single instance of the a event at node 3 and the b event at node 7 may occur in an executable sequence of program events. The MIP edge, however, forms a cycle; this introduces arbitrarily long TFG paths containing nodes 3 and 7.

There are two goals in refining the TFG: reducing its size and eliminating behaviors that are not executed. Reducing the size of the TFG can reduce the cost of performing state propagation analysis. Eliminating

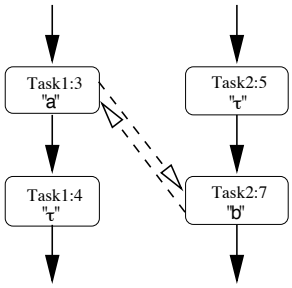

Figure 6: MIP Paths

unexecutable behaviors can improve the precision of state propagation analysis results as well as reduce its cost.

One type of refinement collapses sequences of τ events. Since τ events represent actions that we are not interested in, we can reduce a sequence $a, \tau, \ldots, \tau, b$ to the sequence a,b. Thus, we are really interested in maintaining the ordering of all non-τ events. In the remainder of this subsection we describe two refinements, simple MIP edge refinements and τ labeled node refinements. Both refinements reduce the size of the TFG but preserve its conservativeness with respect to all sequences of non-τ labeled events.

Simple MIP Edge Refinement

Naive construction of MIP edges between all pairs of nodes in different tasks requires 128 such edges for the Semaphore example. We can eliminate many of those edges in cases where we can prove that the pairwise orderings are either infeasible or are captured by some other path through the TFG. In particular, we can eliminate all MIP edges between nodes where at least one has a τ label. To see why, we need to consider the impact on the sequence of non-τ events when a path includes a MIP edge with a τ label at the source or at the destination of that traversed MIP edge.

Assume that a MIP edge between a τ labelled node, t, and some other node, d, is traversed on a TFG path. For each non-τ labelled node, say a, that is in the same task as t and for which there is path from a to t on which only τ nodes occur, we could traverse the MIP edge (a, d) rather than (t, d). Since the path $a \ldots t$ has only τ symbols the resultant sequence of symbols is identical. We could make a similar argument when traversing the MIP edge in the other direction, when the τ label is on the destination node. In this case, we are interested in finding τ only paths to non-τ successor nodes of the destination node.

Thus, the TFG after this refinement is guaranteed to include all possible non-τ event orderings. In practice, these MIP edge refinements are built into the implementation of the TFG construction algorithm rather than as post-construction TFG transformations. For the Semaphore example, MIP edge refinements reduce the number of MIP edges from 128 to 10.

It is worth noting, that even with this refinement, the set of MIP edges that remains is overly conservative. For many programs, significant numbers of additional MIP edges can be eliminated by further analysis and transformations while retaining TFG conservativeness. We have defined *communication interval refinement* (CI) and implemented it in the FLAVERS/Ada toolset. CI refinement identifies and eliminates MIP edges from the TFG that connect nodes in communicating tasks that, because of the pattern of task synchronization used, cannot immediately precede one another. In Section 8, we present data on how CI refinement increases the precision of analysis results, but we refer the reader to [Dwy95] for the details. We note that recent work [NA98] that exploits *may happen in parallel* analysis subsumes CI refinement.

Alphabet Refinement

The resulting TFG can be further transformed to eliminate edges and nodes that do not add to the set of non-τ event sequences. These transformations include: collapsing local edges leading to τ labeled nodes,

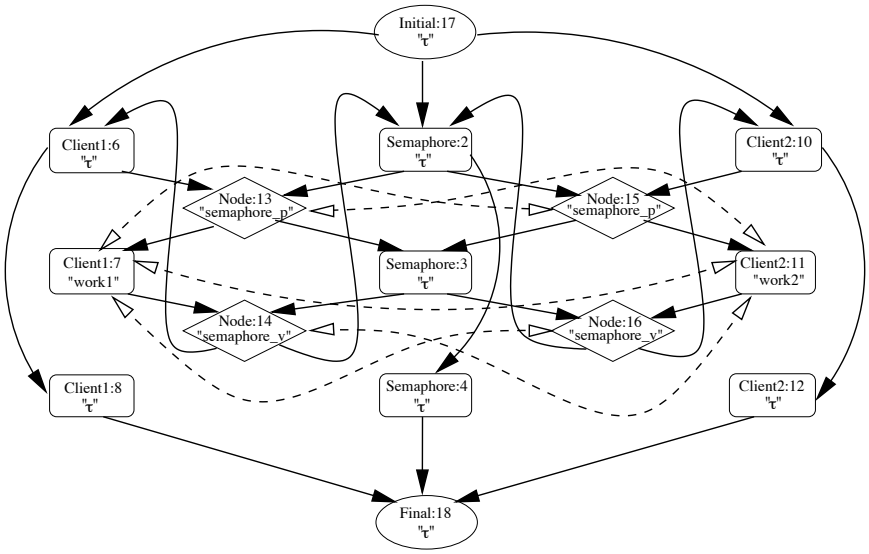

Figure 7: Refined Semaphore TFG with All MIP Edges Included

eliminating those nodes, and collapsing parallel edges. The resulting TFG will have no consecutive local τ labeled nodes.

Alphabet refinement is implemented in the FLAVERS/Ada toolset as a partition-refinement algorithm [AHU74, ASU85]. This algorithm incrementally reduces the size of the graph by merging pairs of nodes where the sequences of symbols labeling paths leading to (from) one of the nodes are the same as the sequences of symbols labeling paths leading to (from) the other. This process stops when no more nodes can be merged. Given the fixed size of the label alphabet, this algorithm is bounded at $O(|N|log(|N|))$ time, where N is the set of TFG nodes [AHU74].

4.8 A Refined TFG for the Semaphore Example

Figure 7 illustrates the alphabet refined TFG that preserves symbol sequences over the alphabet {`work1, work2, semaphore_p, semaphore_v`}; to make the figure more readable bi-directional edges are used to represent pairs of MIP edges. Two nodes have been eliminated by the partition refinement algorithm and the basic MIP refinements have eliminated 118 MIP edges.

5 Describing Executable Behavior

Finite-state verification techniques require a specification of intended behavior to which the executable behavior of a program is compared. In some analysis approaches the intended behavior of interest is implicit, as in the many analyses that attempt to demonstrate freedom from deadlock. In other approaches, including FLAVERS, the intended behavior is specified explicitly by an analyst. While it may be possible to completely specify the intended behavior of a program, large monolithic specifications may be as difficult to construct and reason about as the implementation itself. Thus, in practice, it is often more desirable to write small specifications of important properties of the program. With an analysis approach like FLAVERS, breaking a large specification into smaller partial specifications allows the analysis to be tailored for each part in turn. We discuss the details of this tailoring in Section 7 and present examples in Section 8. FLAVERS supports specification of and reasoning about partial specifications that can be translated into or expressed directly as finite-state automata.

Olender and Osterweil [OO90] introduced *quantified regular expressions* (QRE)s to specify regular expres-

```
id          :  [a-zA-Z][a-zA-Z0-9_:=<>]*
pos         :  [1-9][0-9]*

idlist      :  id
            |  id "," idlist

qre         :  alphabet quantifier expression

alphabet    :  "{" idlist "}"

quantifier  :  "all"
            |  "none"

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

Figure 8: QRE Syntax

sions over control flow graph representations of a program. Standard algorithms convert a user provided QRE specification into a DFSA; users may supply the DFSA directly if they choose. This *property automaton*, or PA, is used as input to the state propagation algorithm. In this section, we first define the syntax and semantics of QREs, we then describe a syntactic style for writing QREs that makes writing specifications easier, and we then define the PA and its construction.

5.1 Syntax of QREs

Figure 8 gives the syntax for QREs. For convenience we have defined a few higher-level operators in the syntax. In particular, the "[- ...]" expression is used for describing sub-strings that exclude specified events; this operator is used extensively in the QRE style described below.

5.2 Semantics of QREs

A QRE consists of an alphabet, a quantifier and a regular expression. The alphabet of a property specification, which we denote as $\Sigma_{property}$, defines a set of event symbols that may be used in the expression. The meaning of each event symbol is established by a mapping to the appropriate program action, where an occurrence of the symbol represents the execution of the associated action. QREs contain a standard regular expression defined over the specified alphabet. Figure 9 defines the semantics of the expression operators in terms of the languages of their operands. Quantifiers indicate the kind of comparison that is to be made between the language of the regular expression and the sequences of event symbols that represent potential program executions. The `all` quantifier indicates that all event sequences described by a TFG, G, should lie in the language of the specified regular expression, E, i.e., $\mathcal{L}(G) \subseteq \mathcal{L}(E)$; this is referred to as the *language containment* test. The `none` quantifier indicates that no event sequence described by a TFG, G, should lie in the language of the specified regular expression, E, i.e., $\mathcal{L}(G) \cap \mathcal{L}(E) = \emptyset$; this is referred to as the *empty language intersection* test.

5.3 An Example Specification

We can use QREs, or DFSAs, to specify a precedence property of the semaphore example given in Figure 3. The property states that prior to any work being performed the semaphore task must have its P entry called. The QRE for the specification of this property is

```
{work1, work2, semaphore_p, semaphore_v}
all
[- work1,work2]* | ([- semaphore_p,work1,work2]*; semaphore_p; .*)
```

$$
\begin{array}{lcl}
\mathcal{L}(\texttt{tau}) & = & \{\} \\
\mathcal{L}(\texttt{x}) & = & \{x\} \\
\mathcal{L}(\texttt{.}) & = & \{x | x \in \Sigma_{property}\} \\
\mathcal{L}(\texttt{[-idlist]}) & = & \{x | x \in \{\Sigma_{property} - \{\text{idlist}\}\}\} \\
\mathcal{L}(\texttt{[idlist]}) & = & \{x | x \in \{\text{idlist}\}\} \\
\mathcal{L}(\texttt{e}_1\texttt{;e}_2) & = & \{xy | x \in \mathcal{L}(\texttt{e}_1) \wedge y \in \mathcal{L}(\texttt{e}_2)\} \\
\mathcal{L}(\texttt{e}_1\texttt{|e}_2) & = & \{x | x \in \mathcal{L}(\texttt{e}_1) \vee x \in \mathcal{L}(\texttt{e}_2)\} \\
\mathcal{L}(\texttt{(e)}) & = & \mathcal{L}(\texttt{e}) \\
\mathcal{L}(\texttt{e\^{}k}) & = & \mathcal{L}(\texttt{e}_1\texttt{;(e}_2\texttt{;(...; e}_k\texttt{)...))} \\
\mathcal{L}(\texttt{e*}) & = & \{\} \cup \bigcup_{i=1}^{\infty} \mathcal{L}(\texttt{e\^{}}i) \\
\mathcal{L}(\texttt{e+}) & = & \bigcup_{i=1}^{\infty} \mathcal{L}(\texttt{e\^{}}i) \\
\mathcal{L}(\texttt{e?}) & = & \mathcal{L}(\texttt{e}) \cup \{\}
\end{array}
$$

Figure 9: QRE Semantics

5.4 A QRE Style

QREs allow for patterns of events to be described as arbitrary regular expressions. Users, however, often prefer to have some guidance in how to express a particular ordering relationship as a formal QRE specification. To address this, we have developed a style for writing QREs that we find makes it easier to both read and write specifications. The style consists of expressions constructed out of two types of sub-expressions: intervals that exclude a set of events, of the form `[- a]`, and required events, of the form `a`. The idea of intervals that require and exclude events is derived from Corbett's ω-starless expressions [Cor92].

QREs in this style often begin with an outer excluding interval that is iterated; this allows program executions that are unrelated to the specification to be trivially accepted. The initial interval is followed by an iterated sequence of alternating required and excluding intervals; intuitively, this alternating sequence captures the pattern of events that is required of a satisfying execution.

```
[- start]*(start;[- excluded,next]*;next; ...;[- start]*)*
```

While this style may lead to slightly longer regular expressions, we find that the structure of these expressions is useful. Our experience has been that this style allows specification of interval-like QREs that can be used to express temporal relationships between system events. Sequences of these intervals provide building blocks that are similar to the operators typically provided in temporal logics. Data races, mutual exclusion, general forms of invariance, response and precedence properties can all be specified using this QRE style. Recent work has expanded and generalized this notion of specification style into a pattern system for property specification [DAC99]. The response QRE shown earlier in this section is an instance of a *global response* pattern in this system.

5.5 Property Automata

Using standard techniques we can construct a finite state automaton from the regular expression of a QRE. This property automaton accepts all event sequences over the event alphabet that correspond to the property of interest. Formally,

Definition 4 *A* **property automaton** *is a deterministic finite-state automaton* $(S, \delta, A, s, \Sigma_{property})$*, where:*
$S = \{s_1, s_2, ..., s_k\}$ *is the set of PA states that represent equivalence classes of prefixes of strings over* $\Sigma_{property}$*,*
$\delta \mid S \times \Sigma_{property} \rightarrow S$ *is the state transition function,*
$A \subseteq S$ *are the accepting states that are reached only by strings that satisfy the property specification,*

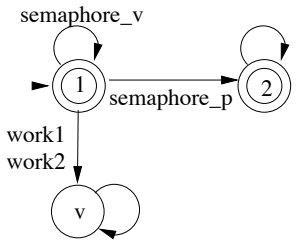

Figure 10: Property Automaton for Precedence Property

$s \in S$ is the unique start state,
$\Sigma_{property} \subseteq \Sigma$ is the alphabet of the property.

The property automaton constructed for the semaphore precedence property has three states and is illustrated in Figure 10. Many PAs contain non-accepting states with no exiting, non self-loop transitions. Such states represent the fact that a string leading to that state has violated the property specification in such a way that no extension of the string can possibly satisfy the property; we call these *violation* states. In Figure 10 the violation state is represented by state v.

Constructing a PA

The PA is a deterministic finite-state automaton. Prior to constructing the PA we convert a given QRE expression to a canonical expression that uses only the operators "|",";", and "*". In the worst case, the number of symbols in the new expression will be no larger than the number of symbols in the original expression times the number of symbols in the property. For any program, we can bound the size of the alphabet of any property specification, denoted $|\Sigma_{property}|$, to be no larger than the size of the program event alphabet, i.e., $|\Sigma_{property}| \leq |\Sigma|$. In this case the increase in the number of symbols from QREs to canonical expressions is bounded by $|\Sigma|$, which is a constant for a given program.

In FLAVERS, we currently construct the PA directly from the regular expression of the QRE using algorithm 3.5 from Aho, Sethi and Ullman [ASU85]. This avoids the construction of an intermediate non-deterministic finite-state automaton. We note that there exist expressions for which this algorithm requires exponential time. In our experience, we have encountered very few realistic specifications of concurrent programs that exhibit this problem. We discuss one such property specification in Section 8 and an alternate means of formulating the specification that avoids the problem.

6 State Propagation Analysis

The aim of FLAVERS is to compare the executable behavior of a program with a specification of intended behavior; state propagation analysis is the mechanism used for performing this comparison. In this section, we describe a practical algorithm for checking the execution behavior represented by the TFG against the intended behavior represented by the PA. State propagation analysis is not new. Howden [How86] and later Olender and Osterweil [OO92] developed state propagation algorithms for checking a regular property of a sequential program modeled as a finite-state automaton; our work builds on the results of Olender and Osterweil. In this section, we explain how state propagation works, we then extend state propagation analysis to apply to a concurrent program modeled as a TFG. Throughout this section we use the terminology of data flow frameworks [Hec77].

A State Propagation Flow Analysis

Let P be a property automaton that accepts a sequence of program events, and let G be a trace flow graph, whose paths include all executable sequences of program events leading to a node. As describe in

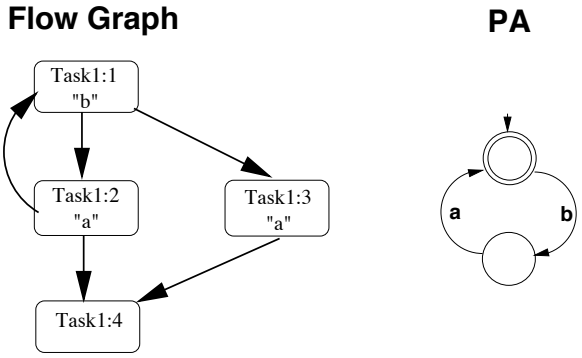

Figure 11: P-equivalent Paths

Section 5, we check consistency between the property and program by checking either language containment, $\mathcal{L}(G) \subseteq \mathcal{L}(P)$, or empty language intersection, $\mathcal{L}(G) \cap \mathcal{L}(P) = \emptyset$.

To simplify the discussion consider just the language containment test. Ultimately, we are not interested in whether *each* individual executable event sequence is accepted by P; rather, we are only interested in whether *all* such event sequences are accepted. If we can devise a means of collapsing flow graph paths that are equivalent with respect to being accepted by P then we may be able to reduce the work required. The data flow formulation of state propagation analysis developed by Olender and Osterweil does exactly that. Let δ^* be the extension of δ, P's state transition function, from symbols in Σ to sequences of symbols in Σ. Two paths, $p_1 = n_{initial} \rightarrow \ldots \rightarrow n_1$ and $p_2 = n_{initial} \rightarrow \ldots \rightarrow n_2$, are *P-equivalent* if they cause P to transition to the same state, $\delta^*(s, String(p_1)) = \delta^*(s, String(p_2))$, where s is the start state of P. Figure 11 illustrates a simple flow graph where all paths leading from node 1 to node 3 are P-equivalent with respect to the given PA that accepts the regular language $(ba)^*$. Equivalent paths include; $\delta^*(1, String(1 \rightarrow 4)) = \delta^*(1, ba) = 1$ and $\delta^*(1, String(1 \rightarrow 2 \rightarrow 1 \rightarrow 2)) = \delta^*(1, baba) = 1$. The strength of this collapsing lies in the fact that it is based on the structure of the property automaton. The collapsing can dramatically reduce the time required for performing the language containment computation.

Formally, state propagation analysis computes for each node the set of P's states that can be reached by any path to that node from the start node of the graph, $States(n) = \{\delta^*(s,p) : \forall p : p = n_{initial} \rightarrow \ldots \rightarrow n\}$. We define a meet semi-lattice over the power-set of P's states with union for meet and super-set as the ordering relation, $L_{sp} = (\mathcal{P}(S), \supseteq, \cup)$, where S is the set P's states, and lattice values $\perp = S$ and $\top = \emptyset$. The function space is defined by extending all total functions over P's states to the power-set of P's states, $F_{sp} = \{F : F(X) = \cup_{x \in X} f(x)\}$, where $X \in \mathcal{P}(S)$. We also add a function $F_\emptyset | \mathcal{P}(S) \rightarrow \emptyset$ to F_{sp}.

Theorem 4
F_{sp} is a monotone function space defined over L_{sp}.

Proof: Monotonicity requires that four properties hold of the functions [Hec77]:

1) (*identities)* If we extend the identity function over S to $\mathcal{P}(S)$ we get the identity function for $\mathcal{P}(S)$. This function is clearly in F_{sp}, which contains all extensions of total functions defined over S.

2) (*closure)* Since the set of total unary functions defined over S is closed under composition so is their extension to $\mathcal{P}(S)$. The addition of $F_\emptyset$ preserves closure since: $\forall f \in F_{sp} : F_\emptyset \circ f = f \circ F_\emptyset = F_\emptyset$

3) (*constants)* For each element of $\mathcal{P}(S)$ there must be a function that produces that element when given $\perp = S$. We construct these functions out of the composition of *filtering functions*. These filtering functions are defined as:

$$f_{i \rightarrow j}(x) \quad = \quad \begin{cases} x & \texttt{if} \quad x \neq i \\ j & \texttt{if} \quad x = i \end{cases}$$

This is a total function defined over S so its extension will be in F_{sp}. For any element $X \in \mathcal{P}(S)$ we can define its complement $Y = S - X$. For any element $x \in X$, we can define the composition of filter functions for Y as:

$$f_Y \quad = \quad \bigcirc_{y \in Y} f_{y \to x}$$

Thus, f_Y applied to S will remove all elements in Y leaving only the elements in X. Clearly, the function f_Y is an element of F_{sp} since it is built up from the composition of filter functions that are themselves in F_{sp}.

4) (*monotonicity)* In L_{sp} we have $\sqcap = \cup$ and $\sqsubseteq = \supseteq$. We need to show that

$$\forall f \in F : \forall x, y \in V \quad : \quad f(x \cup y) \supseteq f(x) \cup f(y)$$

If $f = F_\emptyset$ then this holds since $F_\emptyset$ always produces $\emptyset$. For all other functions:

$$\begin{aligned} f(X \cup Y) &= \{f(z) : z \in (X \cup Y)\} \\ &= \{f(z) : z \in X\} \cup \{f(z) : z \in Y\} \\ &= f(X) \cup f(Y) \end{aligned}$$

since $=$ implies $\supseteq$[5].

□

We can now define the state propagation data flow framework.

Theorem 5
$D_{sp} = (L_{sp}, F_{sp})$ is a monotone data flow analysis framework.

Proof:

From Theorem 4 and the definition of L_{sp}.
□

The framework is instantiated by defining the function map. The function map M_{sp} is defined such that for each node, n, to each value flowing into that node, P's state transition function is applied to produce the value at the node, $f_n(X) = \{\delta^P(L(n), X)\}$, where δ^P is the element-wise extension of δ to the set of P's states. To satisfy the frameworks initial conditions, namely that the value of initial node is $\perp$, we map the constant function that returns P's start state to the initial TFG node, $f_{n_{initial}}(X) = \{s\}$.

As long as we have a flow graph that represents all executable sequences of program events, we can instantiate D_{sp} for it and be sure that its solution contains conservative values for *States* at each flow graph node. From Theorem 1 we know that the TFG represents, for each node, all executable sequences of events that end with the symbol of that node. Using the function map from above, we construct $I_{sp} = (TFG, M_{sp})$ as an instance of D_{sp} . The solution at the final TFG node, $States(n_{final})$, can be compared to the set of accepting states of P, A, to produce a conservative version of either a language containment, $States(n_{final}) \subseteq A$, or empty intersection test, $States(n_{final}) \cap A = \emptyset$.

In addition to providing the complexity and conservativeness results described below, formulation of our analysis as a monotone dataflow framework provides an algorithm for its solution. We present the state propagation algorithm as an instance of Hecht's iterative worklist algorithm [Hec77].

Algorithm 1 (State Propagation Solver)
Input:

A trace flow graph, G, and a property automaton, P.

[5]This is actually a slightly stronger property of function spaces called *distributivity.*

Output:

A set of states of P for the final nodes of G

We use three auxiliary data structures in this algorithm: $Vals$ which is an array of length $|N|$ that holds sets of states for each node, $Wlist$ which is a queue of nodes with at most $|N|$ elements, and v which holds a subset of the states of P.

Initialization:

$$\forall n \in N \qquad Vals[n] = \begin{cases} \{s\} & \text{if } n = n_{initial} \\ \emptyset & \text{otherwise} \end{cases}$$

$$Wlist \quad = \quad \{n_{initial}\}$$

Main Loop:

We evaluate the following statements repeatedly until $Wlist = \emptyset$: [6]

at this point $Wlist = n, n_1, n_2 \ldots n_k$
(1) $Wlist = n_1, n_2, \ldots n_k$
(2) $v = Vals[n]$
(3) $Vals[n] = \delta^P(\bigcup_{p \in Preds(n)} Vals[p], L(n))$
(4) if $v \neq Vals[n]$ then
(5) $Wlist = n_1, n_2, \ldots, n_k, n_{s_1}, n_{s_2}, \ldots$
where $n_{s_i} \in Succs(n)$ and for $1 \leq l \leq k \quad n_{s_i} \neq n_l$
end if

In general, a TFG is irreducible when it contains MIP edges. Thus, a large class of specialized solution methods for analyses formulated over reducible flow graphs are inapplicable when using TFGs. This is one of the main reasons we chose a general iterative worklist algorithm, which does not have this limitation. In addition, simple variants of this algorithm provide the opportunity for further improving the accuracy of flow analysis of concurrent programs that engage in synchronous communication. These variants exploit complete-lattice frameworks formulations of the analysis [Dwy95], which are equivalent to a node-based formulation of meet-of-join frameworks [MMR95].

The node ordering enforced by this algorithm causes computation of the value at a node only when it has the potential to change. If multiple predecessors of a node change their value before that node can be recomputed, the algorithm will only schedule a single recomputation for the node. Together, these ordering improvements avoid some unnecessary computation at nodes which have no chance of changing their values. Careful consideration of Algorithm 1 reveals a number of additional opportunities for eliminating unnecessary computation. As described, for each node the algorithm stores a value that reflects the effects of δ^P at that node; this is called the *out* value for the node. We can easily modify Algorithm 1 to store an additional value that contains the cumulative values at that nodes predecessor; this is called the *in* value for the node. In deciding whether a successor, s, should be put on the worklist, we can then perform the comparison $out(n) \not\supset in(s)$, to determine whether the value computed at n can possibly affect the value at s. If it cannot, then there is no need to put s on the worklist. We can also keep track of the set of predecessors of a node n that have changed value, call it $Changed$, and only compute $in(n) = in(n) \cup (\bigcup_{p \in Changed} out(p))$ rather than considering all predecessors. These optimizations will not improve the worst-case time bound for Algorithm 1, but for dense graphs and graphs whose nodes have high fan-in or fan-out they can yield significant practical speedup. The FLAVERS/Ada toolset implements these improvements.

We can express a bound on the running time of this algorithm in terms of the number of automaton states and the number of TFG nodes.

[6] We describe the worklist as initially containing $k+1$ elements; this is a notational convenience where if $k = 0$ then $W = n$.

Theorem 6 (State Propagation Complexity)
Given a TFG, with nodes N, and a PA, with states S, if we apply Algorithm 1 it will terminate in $O(|S||N|^2)$ time.

Sketch of Proof:

The body of the main loop is $O(|N|)$, since the containment test and append operations in line 5 are $O(1)$, and, in the worst-case, where a node has all other nodes as successors, $O(|N|)$ such operations are performed.

The number of iterations of the body, or visits to a node, can be bounded by considering the number of possible values, $Vals$, that a node can achieve. A node's value is initialized to, at most, $\{s\}$ and can increase in size at most $|S|-1$ times before it equals S, the set of all states of the automaton. Since there are $|N|$ nodes in the graph there are at most $|S||N|$ such increasing iterations. Thus, there are $O(|S||N|)$ iterations of the main loop and the theorem holds.
□

Theorem 7 (Conservativeness of State Propagation)
The maximum fixed point solution of an instance of D_{sp} is a conservative estimate of $States(n)$ for each node in the TFG.

Proof:

From Theorem 3 of [KU77] we have that the solution, X, of an instance of D_{sp} is given as:

$$\begin{aligned} X[n_{initial}] &= S \\ \forall n \in N - \{n_{initial}\} : X[n] &= \delta^P(\bigcup_{p \in Preds(n)}, L(n))X[p]) \end{aligned}$$

By induction on the lengths of paths leading to a node, n, it follows that all paths leading from $n_{initial}$ to predecessors of n have correct *States* values. These paths are extended by a single step to reach n by the second equation given above. Thus, the solution X satisfies the definition of *States* and the theorem holds.
□

Example State Propagation

To illustrate the state propagation algorithm. We apply the algorithm to check the precedence property of the semaphore example given in Section 5.3. We illustrate the final sets of PA states associated with the nodes of the alphabet refined TFG in Figure 12. Note that we are primarily interested in the set of states that has propagated to the *final* node, i.e., $\{1, 2\}$. Since that set is contained in the set of accepting nodes for the PA the property is said to hold *conclusively* for the semaphore program.

7 Feasibility Constraints

FLAVERS, like all practical flow analyses, is based on abstractions of program control and data information that are encoded into the flow graph. It is these abstractions that reduce the size of the graph, thereby enabling efficient analysis of software systems. Along with this efficiency, however, comes imprecision. When used for compiler optimization this imprecision might lead to some missed opportunities for code improvement. Since FLAVERS is intended for verification of system behavior, imprecision that leads to an inconclusive analysis result is more noticeable and damaging to the utility of the analysis.

FLAVERS incorporates an approach for inserting into the analysis additional semantic information that is not explicitly represented in the TFG. This effectively increases the semantic content, thereby potentially

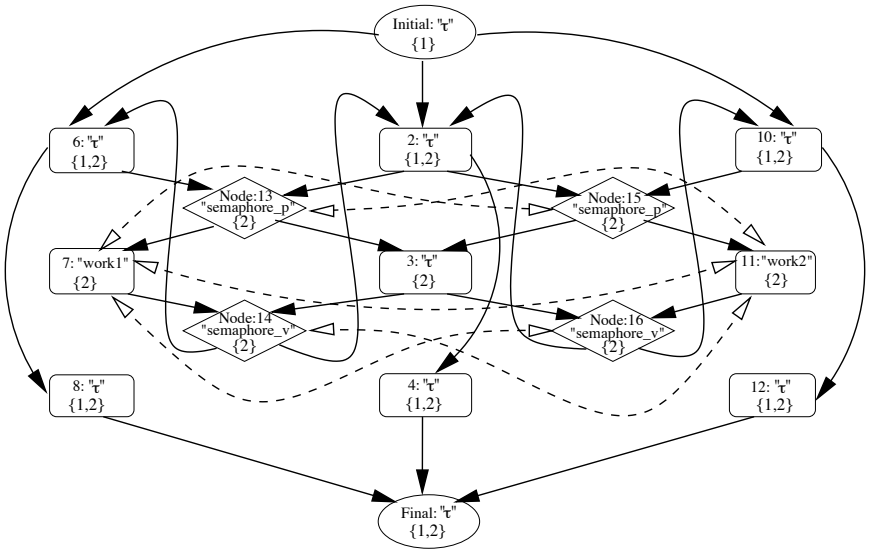

Figure 12: Refined Trace Flow Graph with Propagated PA States

eliminating some inconclusive results that occurred due to the approximate TFG model. Since the TFG overestimates executable program behavior, strengthening the semantic content can be viewed as reducing this overestimate.

We advocate a process where constraints are selectively introduced by the analyst when it is determined that the analysis results are imprecise without this additional information. A constraint basically encodes a necessary condition for improving the precision of the program model during analysis. If a constraint is determined to be false along some TFG path, then that path is not executable. We can think of a constraint as filtering out some targeted infeasible executions that affect the analysis. A conjunction of necessary conditions is itself a necessary condition that is at least as strong as any of the conjuncts in isolation. Thus, multiple constraints pass through only those executions that are consistent with all of the necessary conditions.

In FLAVERS, constraints can themselves be thought of as flow analyses, where each constraint provides a unique *violation* value. The interpretation of this value is that the necessary condition encoded in the constraint has been violated. Constraint flow analyses are combined with the state propagation analysis to form a qualified flow analysis [HR81]. The space of possible constraint analyses is broad, but, in this paper we focus on the sub-class of constraints that can be encoded as a DFSAs. There are different approaches that can be employed to combine such constraints with the property specification during state propagation analysis. The approach we describe in this section, and the one implemented in the FLAVERS/Ada toolset used to gather the data in Section 8, is to form the smash product [Gun92] of the constraint and property automata, and then propagate states of this product automaton. An alternate approach, which forms reachable smash product states on-the-fly, has been also been implemented in the FLAVERS/Ada toolset and a performance comparison between the two is reported in [NCO98].

The TFG, as with most flow graphs, abstracts data values. This can lead to the representation of infeasible control flow paths. Constraints that encode variable values and transitions between those values can be used to recover some data related information. We have found this to be such a common and useful constraint that the FLAVERS/Ada toolset provides automated support for creation of certain kinds of *variable automaton constraints*. The presence of MIP edges in the TFG can introduce infeasible interleavings of events in separate tasks. While refinements can remove some of these infeasibilities, constraints that enforce local task control flow can be used to eliminate still more. This too is a very useful type of constraint, and so the FLAVERS/Ada toolset provide automated support for such *task automaton constraints*. Although variable and task automaton constraints are the only kinds of constraints where FLAVERS currently provides

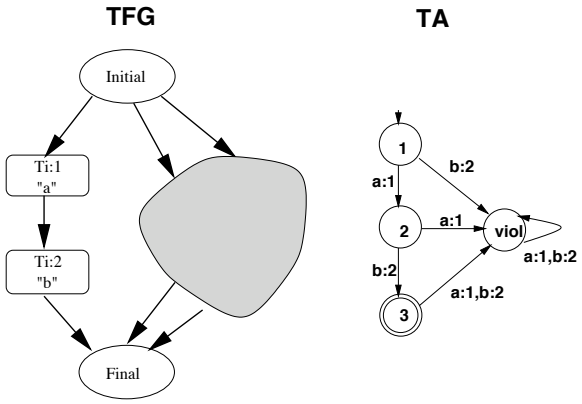

Figure 13: Example TFG and TA

automated support, the concept is very general. The behavior of missing components of partially defined software systems can be captured in FLAVERS analysis through *interface* constraints [Dwy97]. The hardware context [NCO96] and user interaction profiles [BMD96] for a software system can also be incorporated into analysis through *environment* constraints.

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

7.1 Task Automaton Constraints

As illustrated in Figure 6 the presence of MIP edges in the TFG introduces paths that may violate event orderings that are encoded as control flow edges in the TFG. One approach to eliminating consideration of some infeasible MIP edges during state propagation is to enforce the control flow orderings for at least one of the associated tasks. We do this by encoding the feasible state transitions of an individual task as a feasibility constraint. During state propagation, this TA constrains the analysis to consider only TFG paths that correspond to the set of event sequences modeled for that task.

To construct a TA we build the node-edge dual of a control flow graph by converting nodes to transitions and edges to states, by adding a violation state and by defining all the corresponding transitions to that state. We could derive the TA directly from the CFG, mapping nodes to their appropriate event symbols, or, as we have chosen to do here, we derive the TA from the task's corresponding CFG that is embedded in the TFG. This way any refinements that may have been applied to the TFG will be reflected in the task's sub-flow-graph as well.

A TA is constructed for task T_i from the sub-flow-graph of the TFG with $\{n : n \in N \wedge Task(n) = T_i\}$ and the set of local edges in this sub-flow-graph that are incident upon these nodes. The initial and final nodes of the sub-flow-graph determine the start and accept states of the automaton, respectively. Each local node in the task sub-flow-graph of the TFG has a corresponding symbol in the TA. This symbol is a concatenation of the event label for the node with the node's unique id. Thus, the symbol $a : 1$ represents the instance of event a at node 1 in the TFG. We do this since we now need to know, not only what event occurred, but the node where it occurred.

Figure 13 illustrates a fragment of a TFG and the TA for task `Ti`. For this example, there are four states: violation, `v`, and one for each of `Ti`'s local edges. The start state, which for this example is state 1, is associated with the edge from the initial node to the start node of the task, i.e., the local successor of $n_{initial}$. State 3 is the accepting state.

Formally, let $Id(n)$ map each TFG node to a unique positive integer and let *concat* be string concatenation.

Definition 5 *A* **task automaton constraint** *is a deterministic finite-state automaton* $(S_{TA}, \delta_{TA}, A_{TA}, s_{TA}, \Sigma_{TA})$

where :
$S_{TA} = N_i \cup \{v\}$ *is the set of states (one for each local node in the task sub-flow-graph and a violation state,* v*, to represent paths in the TFG that violate task control flow orderings);*
$\delta_{TA} = \{n_i \xrightarrow{a} n_j \mid (n_i, n_j) \in E_i \wedge a = concat(L(n_i), Id(n_i))\} \cup \{n_i \xrightarrow{a} v \mid a \in \Sigma_{TA} - \{concat(L(n_i), Id(n_i))\}\} \cup \{v \xrightarrow{a} v \mid a \in \Sigma_{TA}\}$ *defines a transition for each local node in the task sub-flow-graph and where all other transitions lead to the* violation *state;*
$A_{TA} = F_i$ *is the accepting state corresponding to the exit edge of the task sub-flow-graph;*
$s_{TA} = S_i$ *is the start state corresponding to the start edge of the task sub-flow-graph;*
$\Sigma_{TA} = \{\tau\} \cup \{s | \forall n \in N_i : s = concat(L(n), Id(n))\}$ *is the TA alphabet consisting of symbols formed by the concatenation of the node label with the unique node id, one for each local node in the task sub-flow-graph.*

The structure of this automata is such that it not only records the occurrence of program events, but also the points in the task where the events occur. This enables the TA to enter its violation state if a sequence of occurrences of program events that violate the control flow structure of the task is encountered.

A task automaton is constructed by traversing the sub-flow-graph of the TFG corresponding to a particular task. We note that to enforce the conditions encoded in a TA during state propagation, the TFG alphabet must use the appropriate symbols of the TA alphabet for each node in the modeled task. This involves converting the TFG to use the label/edge id symbols as node labels; we also introduce parallel PA transitions at each PA state, s, such that for each symbol, $a \in \Sigma$, we define $\delta(s, a : id) = \delta(s, a)$ for each symbol $a : id \in \Sigma_{TA}$. The TFG relabeling process requires $O(|N_i|)$ steps.

Theorem 8 (Correctness of TA)
If a sequence of program events leads to a TA violation state then it correspond to an infeasible program execution.

Proof:

> Call the modeled task T_i. Consider any sequence, s, of program events over the symbols $\Sigma \cup \Sigma_{TA}$ [7]. This sequence is of the form, $\omega = a_1, a_2, a_3, \ldots, a_k$. Due to the structure of TAs, labels that are not in Σ_{TA} will trigger self-loop transitions thereby preserving all TA state information stored at a TFG node. Call ω_{TA} the projection of ω onto Σ_{TA}.
>
> From the definition of a TFG, all executable sequences of program events that occur in T_i are represented as control flow paths in the sub-flow-graph for T_i. Assume that $\delta^*_{TA}(\omega_{TA}, s_{TA}) = v$. In this case, $\exists_{a \in \Sigma_{TA}}$ such that $\omega_{TA} = \alpha_1 a \alpha_2$ and $\delta^*_{TA}(\alpha_1, s_{TA}) \neq v$ and $\delta_{TA}(a, \delta^*_{TA}(\alpha_1, s_{TA})) = v$. By construction of the TA this can only happen if a TFG node corresponding to $\delta^*_{TA}(\alpha_1, s_{TA})$, i.e., a node n where, $States(n) = \delta^*_{TA}(\alpha_1, s_{TA})$, does not have a successor whose label is the first component of a. If there is no such edge in the TFG then the sequence cannot be executable since the TFG is conservative. Thus, the theorem holds.

□
Construction of a TA requires $O(|N_i|^2)$ steps where $N_i \in N$ is the set of nodes in T_i. In the worst-case the sub-flow-graph for the task in question is fully-connected, thus we need to construct N_i transitions for each of the N_i states of the TA. Thus the entire algorithm is $O(|N_i|^2)$. In Section 8, we will see that constructing TAs is very fast in practice.

7.2 Variable Automaton Constraints

Flow graphs do not typically model program variables. For many programs, however, accurate analysis depends on modeling some of the critical program variables. We are often interested in modeling variables that are used in conditional statements or in guards that control communication statements. Many of these variables are defined over small finite domains and modified in a disciplined way. Examples include Boolean

[7] Such a sequence corresponds to a path from a TFG that has had the labels of nodes in T_i transformed as described above.

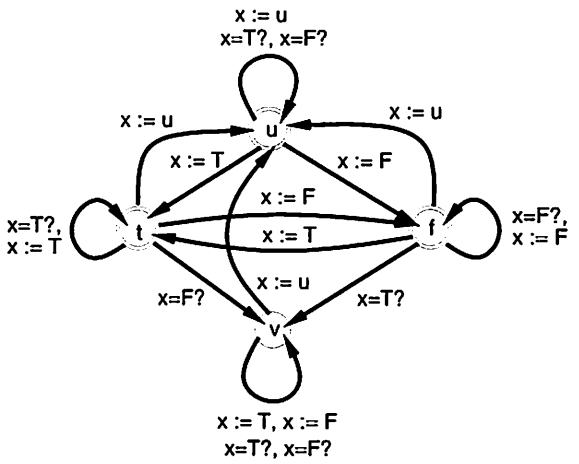

Figure 14: VA for Boolean Variable

variables to which only constant values are assigned and bounded counter variables to which only increment and decrement operations are applied. We have developed a technique for encoding the state and state transitions of such program variables as VA constraints.

VA transitions represent modifications to the value of the variable and the results of conditional tests of the values of those variables used in branch decisions. The violation state, v, represents that a path in the TFG violates the semantics of a program branch decision with respect to the current variable state.

The VA constraints for a Boolean variable is shown in Figure 14. In this example, assignments to the variable are modeled as `x:=T` and `x:=F`; we include `x:=u` to represent situations where an unknown value is assigned to the variable. Results of tests in conditionals are represented by their success, `x=T?`, and failure, `x=F?`. There are transitions to the violation state, from the true state when the `x=F?` test result is encountered and from the false state when the `x=T?` test result is encountered, i.e., when the result of a test is inconsistent with the value of the variable described by a VA state. All Boolean variables have VAs that are identical in structure.

More formally,

Definition 6 ***A* Boolean variable automaton** *for a Boolean variable, x, is a deterministic finite-state automaton $(S_{VA}, \delta_{VA}, A_{VA}, s_{VA}, \Sigma_{VA})$, where:*
$S_{VA} = \{u, t, f, v\}$ is the set of states that define the abstract values that are modeled for the variable, u for unknown value, t for true, f for false, and v for violation;
$\delta_{VA} \mid S_{VA} \times \Sigma_{VA} \rightarrow S_{VA}$ defines the transitions as in Figure 14;
the accepting states $A_{VA} = \{u, t, f\}$;
the start state $s_{VA} = u$ models an unknown value;
$\Sigma_{VA} = \{"x := u", "x := T", "x = T?", "x := F", "x = F?"\}$ is an alphabet that defines the program events that can effect the value of the modeled variable.

Theorem 9 (Correctness of Boolean VA)
If a sequence of program events leads to the VA violation state then it corresponds to an infeasible program execution.

Proof:

By the structure of a Boolean VA, there are only two ways to reach the VA violation state: being in state *true* and encountering an "$x = F?$" test result or being in state *false* and encountering an "$x = T?$" test result.

If a Boolean variable has the value *true* upon reaching a conditional that tests if that variable has the value *true* the program will take the true branch. The false branch of such a conditional

is labeled "$x = F$?", thus propagation of the *true* value across the false branch will lead to the violation state. Taking the false branch at this point is clearly in violation of program semantics and is consequently infeasible in any real program execution.

Analagous arguments follows for $false$ variable values and tests for $false$ values in conditionals.

□

To enforce the conditions encoded in a VA during state propagation, the TFG alphabet must include the symbols in the VA alphabet. The increase in precision that results from incorporating a VA into an analysis depends on the ability to identify program statements that cause variable state transitions. In many cases this can be done statically, as is usually the case with Boolean variables. We have developed definitions of variable automata for additional classes of variables including bounded counter variables and variables defined over enumerated types.

Boolean variable automata are constructed in $O(1)$ steps since they involve a simple renaming of the transitions in Figure 14. Bounded counter variable automata are constructed in a similar manner in $O(k)$ steps where k is the maximum value of the counter. Enumerated variable automata are constructed in a similar manner in $O(k)$ steps where k is the number of enumerated values. The FLAVERS/Ada toolset allow 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 program variable name.

We note that, in principle, a wide range of finite-state abstractions of program variables could be encoded as variable automata and thereby incorporated into FLAVERS using state propagation analysis. Furthermore, for abstractions that do not admit a simple encoding we can still accommodate them in FLAVERS as long as they have a unique violation state.

7.3 Combining Constraint and Property Analyses

To incorporate constraints into FLAVERS analyses, the state propagation analysis is solved in conjunction with analyses for each constraint. Two approaches to achieve this as discussed in [NCO98]. One approach is to use an instance of the state propagation analysis for each constraint automata and to solve those simultaneously with the state propagation analysis for the property automaton. In this section, we describe an alternate approach that is possible for constraints encoded as DFSAs and report on experiments that use this approach in Section 8.

As described previously, constraint automata are structured such that a flow graph path that drives a constraint to its violation value is known to be infeasible. Given a PA and a collection of feasibility constraints, $FC_1, \ldots, FC_n$, encoded as finite automata one can construct a product automaton, which we call a *constrained property automaton* (CPA).

Definition 7 *A* **constrained property automaton** *is a deterministic finite-state automaton* $(S_{CPA}, \delta_{CPA}, A_{CPA}, s_{CPA}, \Sigma_{CPA})$*, where:*
$S_{CPA} = \{(s_p, s_1, \ldots, s_k) : s_p \in S_{PA} \wedge \forall_{1<=i<=k} s_i \in S_{FC_i}\}$*, the set of states consists of tuples of states of the constituent automata;*
δ_{CPA}*, the transition function is the component-wise application of the constituent automata transition functions to a CPA state;*
$A_{CPA} = \{(a_p, a_2, \ldots, a_k) : a_[\in A_{PA} \wedge \forall_{1<=i<=k} a_i \in A_{FC_i}\}$*, the accepting state corresponds to the tuple of accepting states in all the constituent automata;*
$s_{CPA} = (s_{PA}, s_{FC_1}, \ldots, s_{FC_k})$*, the start state is the tuple of constituent start states;*
$\Sigma_{CPA} = \Sigma_{property} \cup \bigcup_{i \in 1,\ldots,k} \Sigma_{FC_i}$ *the alphabet is the union of the constituent alphabets.*

The CPA enforces the conjunction of the necessary conditions encoded in the FCs. If any of the individual conditions of the FCs is violated then the conjunction is violated. In practice, all CPA states that represent a violation state in any of the FC automata can be collapsed into a single violation state without losing

precision in detecting infeasible sub-paths in the TFG. This collapsing is the primary difference between the smash-product [Gun92] and a standard automaton product.

Interpreting the results of state propagation of a CPA, as opposed to a PA, has one slight difference from the description in Section 3. When comparing the terminal TFG *States* values to the CPA accept states, we ignore the CPA violation state. This removes any contribution to state propagation results of TFG paths that violate any of the conditions enforced by the FCs encoded in the CPA.

One significant CPA optimization is the use of the union, rather than product, of the constituent alphabets to reduce the size of the alphabet. This is correct since all FC automata alphabets and the PA alphabet are subsets of Σ. This optimization can dramatically reduce the storage required to represent δ_{CPA}. Other optimizations can be applied to CPA construction as described in [Dwy95]. Despite such optimizations the exponential nature of the CPA is unavoidable. The intent is that FLAVERS's ability to use a small number of judiciously selected necessary constraints will allow control over the potentially rapid growth in the cost of CPA-based analysis.

8 Evaluation of FLAVERS

The polynomial bounds on the running time for FLAVERS suggests that this approach may scale better than exponentially bounded approaches. Such bounds, however, are worst-case and say nothing about what one might expect for the cost of analysis on typical applications. To develop an understanding of the feasibility and practicality of FLAVERS, we applied it to the analysis of a variety of programs and correctness properties of those programs.

In the rest of this section, we describe our methodology for applying FLAVERS to a given analysis problem, we then describe in detail the results of our empirical evaluation, and follow up with a number of observations about these results.

8.1 Methodology

Our empirical evaluation was performed using an implementation of FLAVERS that is targeted for Ada tasking programs. The toolset is built on top of Arcadia [TBC+88] infrastructure components and is referred to as the FLAVERS/Ada toolset version 1.0.

The programs we selected to be analyzed in this evaluation were all Ada tasking programs that had been used as examples in the concurrency analysis literature. Some of these programs have been well studied and a variety of finite state verification techniques have been applied to them. We chose a mixture of programs including scalable programs with replicated tasks, scalable programs with dissimilar tasks, and non-scalable programs with dissimilar tasks. For some of these programs there are well-known properties to be checked; for others we developed our own properties and specifications by reverse engineering the application. The goal was to come up with properties that might be checked by a developer of the application. Although we learned a great deal from this evaluation, since the sample size is small and not known to be representative, we do not claim that the results can be generalized to all Ada tasking programs.

There are two important concerns in this evaluation: cost and precision. For evaluating the former, we consider the cost of constructing all of 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 refinements and feasibility constraints that were included in order to achieve a conclusive result. We considered alphabet refinement, communication refinement, task automaton constraints for each task, and variable automaton constraints for each variable. We followed a process where we first considered the basic analysis without any refinements or constraints, and then added refinements and constraints as they seemed appropriate for the problem. A more rigorous approach would be needed in order to determine the optimal configuration, where optimal would mean that it took the least resources to produce conclusive results. Note that sometimes adding a constraint reduces the execution time, so even after a conclusive configuration is found, additional constraints would need to be considered to

determine an optimal configuration. Our goal was not to find the best execution time but to determine if there was a reasonable process that a developer might follow to easily find a conclusive result. As discussed later, the results of this evaluation has influenced the order in which we now select refinements and constraints. For example, we now always do alphabet refinement before adding any additional constraints, as described in Section 4.

We also gathered information about the rate of growth of analysis cost by varying the size of those programs that are scalable. In some cases there is a concomitant increase in the size of the property that needs to be checked as well. Before scaling a program, we first considered a small, yet reasonable, instantiation of that program and then found a configuration of refinements and constraints that led to conclusive results. Since we ran our experiments in batches, we sometimes had more than one candidate configuration. In such cases, we selected the one that required the least execution time. We then used this configuration as we scaled 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 problem is scaled to a larger size. This is often the case, however, and was indeed the case for all the scalable programs and properties that we considered in our experiment.

In the sections that follow we present data on the refinements and constraints that were used and on the run-time costs. Our primary measure of analysis time is the sum of user and system time for a conclusive analysis result as measured by `/bin/time` on on a normally loaded multi-user SPARC 10/30 with 32 megabytes of physical memory. The FLAVERS/Ada tools are compiled using the `SunAda 1.1(j)` compilation system with optimizations disabled (to work around known compiler bugs). To simulate the use of FLAVERS/Ada in practice, we ran the series of tools needed to go from TFG construction through state propagation analysis. We ran each analysis three times and took the average of those times. The run-times include all overhead costs related to the object management infrastructure on which the tools are built. The reported run times are extremely large and reflect the fact that this evaluation was done on a prototype system designed for flexible experimentation [DC96] and not for efficiency. A more efficient version of the FLAVERS/Ada tool set that includes several optimizations is currently under development and preliminary data suggests that it achieves 2 orders of magnitude better performance than the prototype described here. The timing results reported here, however, are still useful since they represent an upper bound on the performance of the FLAVERS approach.

To reason about the rate of growth of the cost of analysis we plot analysis time versus the number of nodes in the TFG for the program under analysis. These plots are made on a log-log scale. We judge rate of growth by comparing the slopes of these plots to the slopes of *reference* polynomials, such as N^3 where N is the number of TFG nodes. If the slope of the analysis time plot is less than that of the reference then the rate of growth is less than the reference polynomial. We refer to these as *rate-of-growth* plots.

8.2 Detailed Empirical Results

We applied FLAVERS/Ada to four scalable and one non-scalable Ada tasking programs. The source code for these programs is given in [Dwy95]. For all of the experiments with scalable programs, we started with a small, yet sensible, version of the program and looked at progressively larger versions of the program. Each larger version is twice the size of the preceding one; we look at five versions for each problem. Unless explicitly noted the tool was able to handle each size problem.

8.2.1 A Simple Protocol Problem

This simple protocol is a program that illustrates a common mechanism for resource management: exclusive locks. Corbett [Cor92] presents it to illustrate capabilities of the constrained expressions (INCA) toolset. We use a slightly modified version of that code, since the original was not a legal Ada program.

The system consists of at least 3 tasks, a lock manager task, a communication channel task, and two or more client tasks; thus, it is scalable in the number of clients. The client tasks attempt to gain access to a resource. Once a task holds the lock to that resource, it sends a message over a simulated communication

channel. A lock manager task is responsible for controlling exclusive access to the resource. Our code differs from Corbett's in that we use a single entry for acquiring and releasing the resource as opposed to separate entries for each client task. This is a more practical implementation since the lock manager task need not change with every change in the number of clients. The communication channel task can be thought of as a passive data server; it accepts data and transmits it over the simulated channel.

We checked a number of properties related to the possibility that messages from different clients are interleaved as they are sent over the channel. We wanted to check the global property that "when client i sends a header then client i's packet will be sent before any other header". A QRE specification of such a property is problematic because it will depend on the number of clients. We initially explored a simple and locally restricted version of such a specification. The QRE **header-packet** for client 1 of a 2 client version of the protocol is:

```
{h1, p1, h2, p2}
all
   [-h1]*;(h1;[-p1,h1,h2]*;p1;[-h1]*)*
```

It says that if a header from client 1, *h1*, is ever sent then we will not see a header from either client 1 or 2 until we see a packet from client 1, *p1*. We applied our refinement and constraint selection process to explore the effects of different refinements and feasibility constraints on the precision and cost of analysis of this property. We performed seven different variations of FLAVERS analyses: basic, alphabet-refined, alphabet and communication interval refined, and alphabet refined with 1, 2, 3 and 4 TAs. We found three analyses that obtain conclusive results and the cheapest, by an order of magnitude, was alphabet and communication interval refinement combined. In the analyses of successively larger versions of the protocol program we applied both of these refinements.

We next considered how the cost of analysis scaled as we increase the number of client tasks in the protocol program. For this program, since we know that all the client tasks are identical, we could argue that once we show that this property holds between client 1 and client2, then it holds among all pairs of clients. In general, the clients may not be identical, however. Hence, we attempted to evaluate how the FLAVERS analyses would scale if we tried to verify the global property. Thus, in addition to the single task **header-packet** property described above, we considered two global variants of this property: multiple application of the single task version (where the analysis time is summed over this set) and a composite version. Multiple application of the single task version involves constructing a different **header-packet** property for each program task; these QREs are all simple renamings of one another. The analysis time is then summed over each. The composite version is an iterated disjunction of each of these individual single task versions.

We start with the single task property. Figure 15 plots total analysis time versus the number of nodes in the TFG. This is a rate-of-growth plot that includes a reference line whose slope is N^3. We can see that the rate of growth of the alphabet and communication interval refined analysis is sub-cubic for this problem. Total analysis time for the largest example, 32 clients, was approximately 19 minutes.

The multiple version analysis runs, for each Client task in the program, a different version of the single task property. We use the fact that the global property of the program we wish to reason about can be decomposed into a conjunction of a number of more local properties, i.e., relating events in a smaller set of program tasks. Each of these conjuncts can then be analyzed in isolation. The design of the FLAVERS/Ada tools allows us to reuse almost all of the analysis artifacts across all of the single analyses; thus the cost of TFG construction and refinement is amortized over many individual analyses. Because each of the single task QREs share a common alphabet, the refined TFGs can be reused. Thus, only the QRE construction and state propagation analysis need to be executed for each client task of the program. Figure 15 also plots rate of growth for analysis via multiple **header-packet** analyses. We include an additional reference line whose slope is $T * N^3$ where T is the number of tasks. We see from this plot that the cost of analysis appears to grow faster than N^3 but less than $T * N^3$. Total analysis time for the largest example, 32 clients, was approximately 3 hours and 44 minutes.

Another approach to analyzing the global property of the program is to write a single QRE that specifies

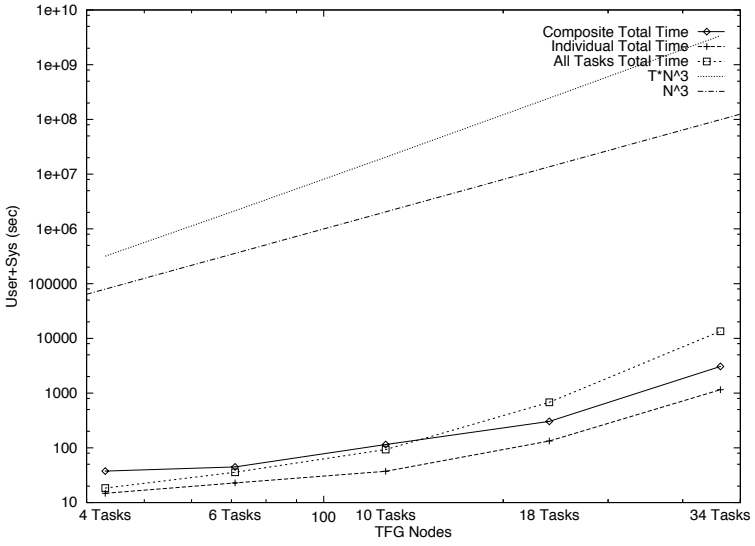

Figure 15: Total Analysis Times for **header-packet**

the desired pattern of behavior for each task in the system. This QRE can grow quite long; for the 32 client program the alphabet consists of 64 symbols and the regular expression contains 1386 instances of those symbols and the regular operators. While this appears impracticably large for a specification, it turns out that the property is very regular and was easily produced by expansion of a simple macro. For the two client program the QRE is as follows:

```
{h1, p1, h2, p2}
all
   [-h1,h2]*; ( (h1;[-p1,h1,h2]*;p1 | h2;[-p2,h1,h2]*;p2); [-h1,h2]* )*
```

We note that unlike the previous versions of the **header-packet** property, this composite QRE will scale as the number of clients scales. Figure 15 includes a rate-of-growth plot for analysis of the composite property. We see from this plot that the cost of analysis appears to grow faster than N^3 but less than $T * N^3$. Total analysis time for the largest example, 32 clients, was approximately 51 minutes.

We have illustrated two FLAVERS analyses that are capable of verifying the global **header-packet** property. While the multiple analyses approach is more effective for small problems it quickly increases above the cost of the composite analysis, crossing at a program with approximately twelve tasks. We note that this instance of a multiple analyses approach is about the best we could hope for; we have maximized the amount of reuse of analysis artifacts. While we cannot generalize from just this example, it is clear that a composite analysis can fare well in comparison to multiple analyses.

We now turn to the analysis of a property of the protocol program that specifies that there should be **no-orphan-packets.** Intuitively, every packet sent should have a matching preceding header. For this study, we specified a weaker version of this property, that the first packet sent by a single task cannot appear before its matching header, as the following QRE:

```
{h1, p1, h2, p2 }
none
   [-h1]*;p1;.*
```

This is weaker, as conceivably the error could appear on subsequent packets. Unlike the previous properties, this one specifies that no program execution satisfies the specified pattern of behavior. We tried basic, alphabet and communication interval refined analysis and found that alphabet and communication interval refinements combined produced conclusive results.

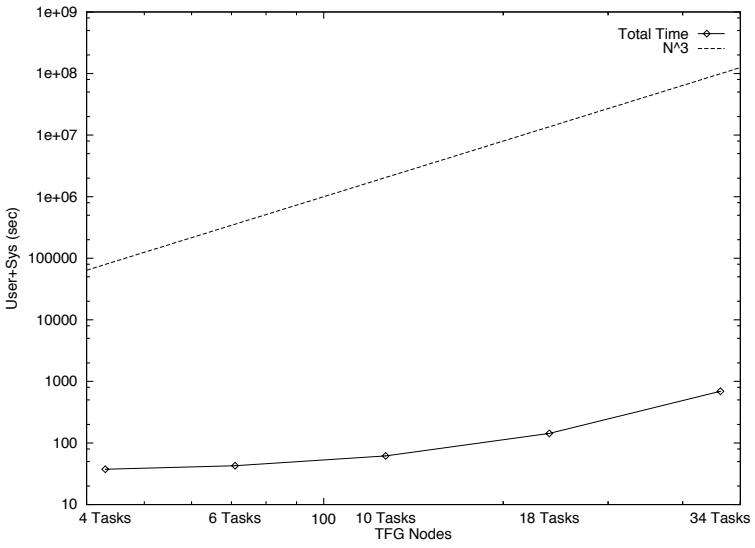

Figure 16: Total Analysis Time for **no-orphan-packets**

In Figure 16 we present the rate-of-growth plot of the total analysis time for this property. We see from this plot that the cost of analysis appears to grow less than N^3. Total analysis time for the largest example, 32 clients, was approximately 11 minutes.

8.2.2 The Readers/Writers Problem

The readers/writers problem is a standard example that implements a means of safe access to shared data. A number of researchers have analyzed versions of the readers/writers problem [ABC+91, DBDS93]. The state space of the program grows exponentially with the number of tasks, where a state of the program records the states of the readers and writers. Assuming a client task can only be in one of two states, we end up with 2^{r+w} states, where r is number of readers and w is number of writers. Thus, if we can demonstrate polynomial growth of an accurate analysis we will have improved on naive reachability analysis.

The readers/writers problem consists of a central data server, called the control task, and a collection of reader and writer tasks; thus, 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 data server, however, can support clients that both read and write. Our code differs slightly from the examples that others have analyzed [CA94] in that we include explicit program termination code; the controller will only shutdown when there are no active readers or writers. Unlike many of the other example programs that we looked at, such as the protocol and dining philosopher programs, the global state of a readers/writers program is not completely captured by the control states of the client tasks. The control task maintains some local state variables that determines the pattern of communication events that it engages in. 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`.

There are two alternatives for reasoning about the execution of the clients: we can reason about events local to the clients or we can reason about events that are shared with the control task. The **header-packet** property of the protocol program was formulated in terms of local client events. We take the other approach here and specify patterns of events in terms of `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 control task entries. We use the abbreviations `wstart`, `wstop`, `rstart` and `rstop`, respectively, to make the QREs more readable.

The QRE for the **exclusive-read-write** property is:

{wstart, wstop, rstart, rstop}
all
[-wstart]*;(wstart;[-rstart,rstop,wstart,wstop]*;wstop;[-wstart]*)*

This QRE excludes the possibility of a write or a read being initiated while a writer is active, however, it does not preclude a reader from being active when a writer starts. An additional QRE can be used to insure this **no-read-upon-write** condition:

{wstart, rstart, rstop}
none
.*;rstart;[-rstop]*;wstart;.*

We performed five different variations of FLAVERS analysis on the 2 reader 2 writer program for the **exclusive-read-write** property: basic, alphabet refined, alphabet and communication interval refined, alphabet refined with a TA for the control task, and alphabet refined with a VA for the `WriterPresent` Boolean variable in the control task. We found that only alphabet refinement with the `WriterPresent` VA obtained conclusive results. In analyses of this property for successively larger versions of the readers/writers program, we applied these refinements and constraints, and as expected, FLAVERS continued to produce conclusive results.

In analyzing the **no-read-upon-write** property we start with the assumption that modeling the `WriterPresent` variable will be necessary; as it proved invaluable in gaining precise analysis results for the previous property. We found that incorporating a VA for that variable was insufficient for gaining conclusive analysis results for the **no-read-upon-write** property. We perform two different variations of FLAVERS analysis on the 2 reader 2 writer program for the **no-read-upon-write** property: alphabet refinement with a `WriterPresent` VA constraint and alphabet refinement with `WriterPresent` VA and an `ActiveReaders` counter VA constraint. The `ActiveReaders` variable is, like `WriterPresent`, local to the control task; it is manipulated as a bounded counter variable. This variable controls the maximum number of readers that can be reading at any time. It can be bounded at any value up to the number of readers in the program. It turns out that we need only use a Boolean abstraction of this counter variable to check the specified property. This is because we only need to know whether some reader is active and not how many are active. In the data presented below we bound the number of readers at 2 for all sizes of the program. We found that for the **no-read-upon-write** property only the alphabet refinement with both VAs obtained conclusive results. In analyses of successively larger versions of the readers/writers program we applied these refinements and feasibility constraints and continued to obtain conclusive results.

We now consider a different property of the readers/writers program. Intuitively, the shared data repository should contain some data before a reader accesses it; this is typical of producer consumer problems where the data repository is some kind of queue. We specify that on all program executions a write must precede the first read as the following QRE:

{wstart, rstart}
all
[-rstart]*;wstart*;.*

We perform five different variations of FLAVERS analysis on the 2 reader 2 writer program for the **write-first** property: basic, alphabet refined, alphabet and communication interval refined, alphabet with a TA, and alphabet with a `WriterPresent` VA. Unlike the analysis for **exclusive-read-write**, the incorporation of a TA for the control task was sufficient for producing conclusive analysis results. This is because the patterns of behavior that **write-first** expresses are prefixes of program executions that are encoded in the control flow structure of the control task. Only after the shared data server has been initialized will the local state variables influence the pattern of communication. We found two analyses for the **write-first** property that obtain conclusive results, alphabet refinement with the TA or the VA, but using the VA analysis was less costly. In analyses of successively larger versions of the program, we applied alphabet refinements with the VA feasibility constraints.

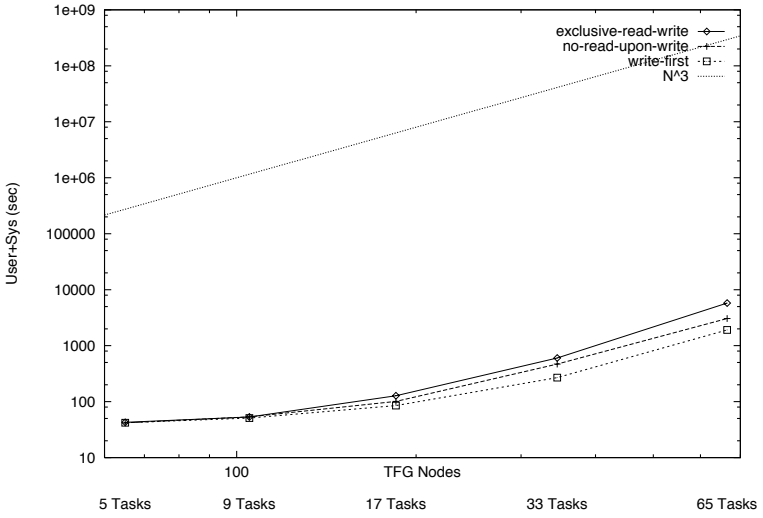

Figure 17: Total Times for readers/writers Analyses

To evaluate how the cost of analysis scales for this program, we began with two readers and two writers, and then increased the size of the program by doubling the number of readers and writers, up to 64 clients, for each of the three properties with their selected refinements and constraints. Figure 17 plots total analysis time versus the number of nodes in the TFG for the three different analyses. We can see that the rate of growth is sub-cubic for all three analyses. Total analysis time for the largest example, 32 readers and 32 writers, was approximately 96 minutes, 51 minutes, and 32 minutes, for the three properties respectively.

8.2.3 The Gas Station Problem

The gas station problem is a simulation of an automated self-serve gas station [HL85]. The gas station consists of a collection of client tasks and a collection of tasks that act as a server. It is similar to the readers/writers problem in that both have a server subsystem and client tasks. In the readers/writers problem the server is the control task. In the gas station the server is a scalable collection of co-operating tasks. Thus, the gas station problem is scalable in two significantly different dimensions: number of clients and size of server.

The server component 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 clients, or customers, pay for gas, pump it, and get their change. Our code differs slightly from the examples that others have analyzed [CA94] in that we include explicit program termination code; the operator will shutdown when there are no active customers and after it has successfully turned the pump off.

We check a property related to exclusive access to a pump. From a black box perspective this is a potentially interesting property to check, since insuring that a single customer may be using the pump at any time is a desirable system property. From a white box perspective, however, the property appears trivial to check due to the control flow in the pump tasks. The QRE for the **one-per-pump** property is:

```
{gas_pump_start_pumping, gas_pump_stop_pumping}
all
   [-gas_pump_start_pumping]*;
   (  gas_pump_start_pumping;
      [-gas_pump_start_pumping,gas_pump_stop_pumping]*;
```

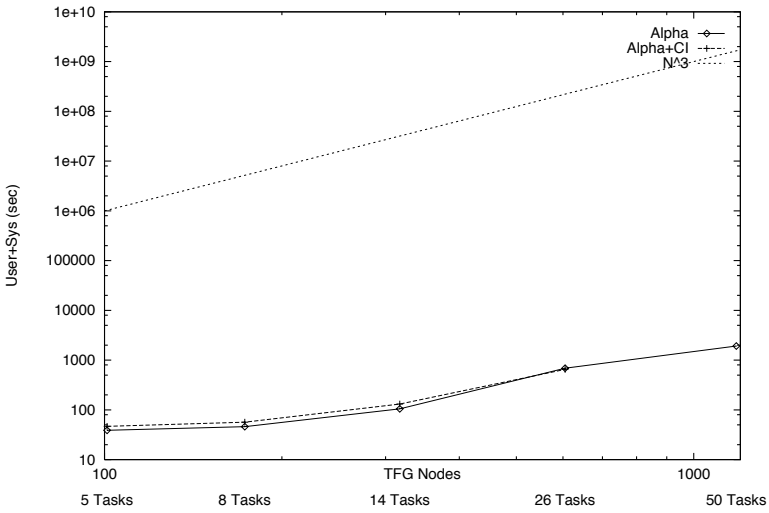

Figure 18: Total Analysis Times for **one-per-pump**

```
    gas_pump_stop_pumping;
    [-gas_pump_start_pumping]*
)*
```

We performed three different variations of FLAVERS analysis on the 3 customer 1 pump program for the **one-per-pump** property: basic, alphabet refined, alphabet and communication interval refined. The latter two produced conclusive results, but alphabet refined was significantly less expensive. Thus, we used alphabet refinement when we scaled the problem.

To evaluate how the cost of the analysis scaled, we increased the number of customer tasks in the gas station program. We began with 3 customers, then increased the size of the program by doubling the number of customers at each step. Figures 18 gives the rate-of-growth plots and again shows that the analysis of this problem is sub-cubic. Total time for the alphabet refined analysis of the largest example, 48 customers, was approximately 32 minutes.

8.2.4 The Dining Philosophers Problem

A number of researchers have analyzed versions of the dining philosophers problem [ABC+91, CA94, DBDS93, MR93, YTL+95] to insure deadlock freedom.

The dining philosophers problem consists of equal numbers of philosopher and fork tasks. The program has at least 4 tasks; it is scalable in the number of philosophers and their forks. The tasks are organized into a ring with alternating philosopher and fork tasks. Conceptually, the philosophers are the active entities; each philosopher task has access to two forks, left and right. Philosophers attempt to gain access to both forks simultaneously by communicating with the appropriate fork tasks. Each fork is shared by two philosophers.

A dining philosophers program has the most distributed control of all of the programs considered in this evaluation. Most of the example programs, e.g., readers/writers, gas station, and protocol programs, have centralized servers. The DARTES system, discussed below, has a centralized master task.

We wanted to check the property that "adjacent philosophers cannot eat concurrently", which we refer to as the **neighbors-think** property. As in the case of the **header-packet** property we considered the possibility of specifying a number of versions of the property. Using the fact that neighbors must acquire a shared fork, we specify the property for a single philosopher as the QRE:

```
{ phils_f1_d1, phils_f1_u1 }
```

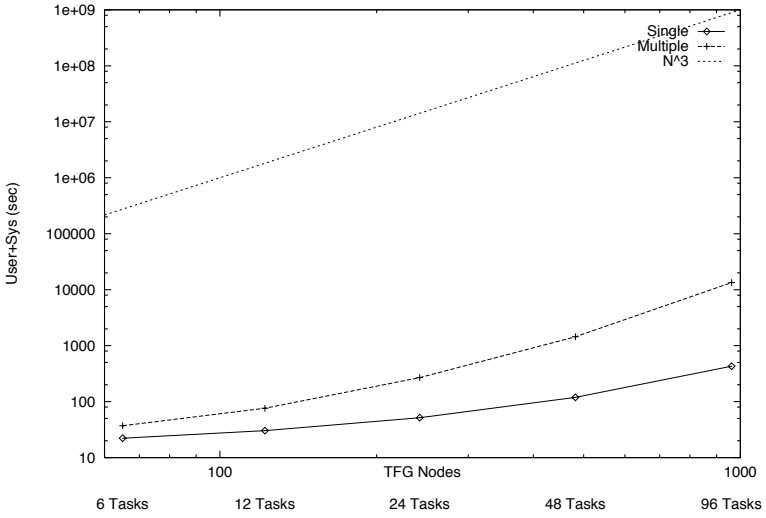

Figure 19: Total Analysis Times for **neighbors-think**

```
all
   [-phils_f1_u1]*;
   (  phils_f1_u1;[- phils_f1_u1]*;
      phils_f1_d1;[- phils_f1_u1]*
   )*
```

From a black box perspective this is a potentially interesting property to check, since insuring that at most one of a pair of adjacent philosopher may be using the fork at any time is a desirable system property.

From a white box perspective, however, the property appears trivial to check due to the control flow in the fork tasks. Given our experience with the protocol problem we considered specifying a composite version of the property. Unfortunately, the complexity of that QRE grows very rapidly since we must allow the forks to operate independently; the result is a specification of the interleavings of the desired event sequences for each fork. This is the first example we encountered where a natural property of a system requires an exponentially long QRE specification. The number of PA states for such a specification will grow exponentially as well. In addition, this property suggests an entire class of such specifications. The alternate approach of checking the pattern of behavior for each fork in isolation seems more attractive; this is analogous to the multiple **header-packet** analysis for the protocol program.

Based on experience we started our analysis series with alphabet refinement and found that FLAVERS produced conclusive analysis results.

To consider how the cost of the alphabet refined analysis scales as we increase the number of philosophers and fork tasks in the program, we began with three philosophers and three forks and increased the size of the program by doubling the number of philosophers and forks at each step. Figure 19 plots total analysis time versus the number of nodes in the TFG for the alphabet refined analysis of a single neighbor. The rate of growth for this analysis also appears to be sub-cubic. Total analysis time for the largest example, 48 philosophers and 48 forks, was approximately 7 minutes.

8.2.5 The DARTES Application

DARTES is a concurrent weapon simulation application. 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 consider.

Masticola [Mas93] and Corbett [CA94] have analyzed versions of the program to check for deadlock

freedom. We use a version that is identical to the one used in Corbett's experiments. This is a variant of Masticola's version, which was not a legal Ada program. We note that both Corbett's and Masticola's versions are hand-inlined versions of the original application that preserves the communication structure of the application but not all of the details of the computation.

The system consists of 32 tasks and is non-scalable. The simulation is structured such that there are two main controlling tasks: one to initialize the simulation and one to shut it down. The other 30 tasks are broken into two classes: data servers and simulation tasks. The data servers provide access to shared data that is needed by multiple simulation tasks. The simulation tasks read from some collection of data servers, compute new values for a component of the simulation, and update data servers appropriately. All of the data servers have the same control structure. The simulation tasks have similar control structure; they vary based on the local computation and the number of data servers they interact with. While data servers and simulation tasks have somewhat different control structures, they share a common pattern of interaction with the control task. Given the uniformity of these tasks, we chose to analyze properties related to the interaction of the control tasks with various data servers and simulation tasks.

The **no-premature-go** property describes a pattern of events that are local to a single task. We selected the task **DISPLAY_x_STATUS_UPDATE** and specify the property that "on no program execution can the go action precede the first pair of initialize and stop actions". For brevity we write the fully-qualified Ada name of task **DISPLAY_x_STATUS_UPDATE** as **T** in the QRE for this property:

```
{T_go, T_initialize, T_stop }
none
   [-T_go]*;T_go;[-T_initialize]*;T_initialize;[-T_stop]*;T_stop
```

We perform analysis to insure that the behavior of the other tasks in the system cannot interfere with the specified operation of **T**.

Unlike some of the other examples, such as dining philosophers, the tasks in the DARTES program have relatively complex internal control flow. Applying the basic FLAVERS analysis to the **no-premature-go** property caused the FLAVERS/Ada toolset to exceed its storage capacity. Incorporating alphabet refinement, however, allows the analysis to easily handle this problem.

For this problem, conclusive analysis results were obtained in less than 6 minutes.

8.3 Discussion

We believe that the empirical results in this section clearly demonstrate the feasibility of FLAVERS analyses. In particular, it demonstrates that there exist natural properties of non-trivial programs that can be verified using an analysis approach whose cost increases as a low-order polynomial in the size of the program.

For many of the analysis problems we found that there were multiple FLAVERS analyses that produced conclusive analysis results. In some cases, such as the single **header-packet** specification of the protocol program, the difference in the cost of analysis was more than an order of magnitude. In other cases, such as the **write-first** specification of the readers/writers program, the difference in the cost of analysis was very small. As a practical matter we want to find the least expensive FLAVERS analysis that provides the level of precision we require. We must, however, factor the cost of finding that analysis into the overall cost of checking the desired specification on a program. We have adopted a greedy approach, taking the first precise analysis we find when looking at the smallest instance of a program. As we look at larger programs, however, this choice may not be the best since some analyses will scale better than others. For example, the composite **header-packet** specification is more costly to analyze than the multiple **header-packet** specification for small instances of the protocol program. For protocols with more than 12 clients, however, the composite specification becomes cheaper since it grows at a lower rate than for the multiple specification.

Intuitively, one of the most important factors that influences the cost of FLAVERS analysis is the degree to which a specification is *local* or *global*. A local specification refers to program events whose only instances are in a small cluster of tasks. A global specification refers to program events whose instances are scattered widely throughout the program. If we imagine using only task automata to add information to a FLAVERS

analysis, it is clear that, in general, the fewer TAs we add the less costly the analysis. A local specification will, in general, require fewer TAs than a more global specification. The properties analyzed in the previous sections vary from being local to a pair of tasks to including events from all program tasks. By using conservative specifications of the behavior of parts of a system we can focus a FLAVERS analysis on specific sub-systems and win both performance and accuracy improvements.

For all of the scalable analysis problems we considered the rate of growth was bounded by either N^3 or TN^3, where T is the number of program tasks and N is the number of TFG nodes. Some of the rate-of-growth curves exhibit a slight cupped shape; this is due, at least in part, to the fact that for some of the analyses the measures of cost grow as a function of both the size of the TFG and the size of the property and constraints used. For all of these analysis problems it is the case that the slope between any pair of points on the associated rate-of-growth plot is sub-cubic. For a number of the programs considered it is well known that the cost of unreduced state space enumeration techniques will scale exponentially with the number of tasks. Thus, it is not the case that we are choosing "easy" problems, rather FLAVERS is demonstrating polynomial growth on problems that grow exponentially in the number of potential program executions. Since we have considered a relatively small collection of programs and properties in this evaluation, we cannot generalize our observations about the rate of growth of the cost of FLAVERS analysis to as yet untried analysis problems.

All finite state verification techniques reason about the execution behavior of non-finite state systems by approximating that behavior with a finite model. Flow analyses have typically used weaker approximations to gain performance at the expense of precision. When applying flow analysis, and FLAVERS in particular, to verification problems, one of the key questions is whether the analysis can produce sufficiently precise results. Our experience suggests that by incorporating additional information into FLAVERS analyses, spurious results can be eliminated. For all of the examples considered in this section, the precision of analysis results was sufficient to verify a given specification with respect to a given program. The cost of this increased precision varied with the program and specification under analysis. FLAVERS analyses can obtain the same level of precision as state space enumeration techniques. Incorporating TAs for all program tasks and VAs for variables that are feasible to model is equivalent to exploring the state space of the program. Fortunately, it is usually the case that we don't need that much information to gain precise analysis results with FLAVERS.

9 Conclusion

We have presented FLAVERS, a finite state verification approach that analyzes whether concurrent or sequential programs satisfy user-defined correctness properties. In contrast to other finite-state verification techniques, FLAVERS is based on algorithms with low-order polynomial bounds on the running time. FLAVERS achieves this efficiency at the cost of precision. Analysts, however, can improve the precision of the results 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 and carried out on an early prototype of the system, indicates that sufficient precision can be achieved relatively easily and that the cost for such analysis grows as a low-order polynomial in the size of the program. Our experimental evaluation also provided insight into which refinements are most effective. For example, alphabet refinement is now always done, and, based on our findings, more effective techniques for eliminating unnecessary MIP edges are being aggressively explored [NA98, NAC99b, NCC99].

In addition to the evaluation reported here, the toolset has been applied in a number of interesting ways. It has been used in an empirical evaluation of concurrency analysis techniques [Cha96], to analyze communication protocols [NCO96], partial software systems [Dwy97], and architectural system descriptions [NACO97].

Although this paper has described the approach in some detail for Ada programs, the approach is relatively

language independent. Each new programming language must be carefully translated into the program 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 for Jovial. We are currently exploring some alternative models for Java [NAC99a]. Interestingly, the state propagation algorithm has not needed to be changed for these different languages, although we would not expect this to always be the case.

In addition to being applicable to programming languages, 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. We have emphsized how it can be used to verify properties of systems. This could be viewed as a complementary activity to testing. Or 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.

Distributed applications are becoming extremely common. Such systems are more difficult to reason about, to test, and to debug. Because of non-determinism, and thus the inability to even replicate test results, it is extremely important that practical techniques be developed to help reason about such systems. Although the experimental evaluation reported here is very preliminary and the example programs are small, the worst case bounds and subsequent work on furthering optimizing the prototype, lead us to believe that the FLAVERS approach will mature into a technique that is of practical use to developers of distributed software.

References

[ABC+91] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading MA, 1974.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BMD96] Christina Bouwens, Rick McKenzie, and Christopher Dean. Investigating static data flow analysis for advanced distributed simulation verification. In *Proceedings of the 15th Workshop in the Interoperability of Distributed Interactive Simulation*, pages 473–478, September 1996.

[CA94] J.C. Corbett and G.S. Avrunin. Towards scalable compositional analysis. *Software Engineering Notes*, 19(5):53–61, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cha96] A.T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques*. PhD thesis, University of Massachusetts at Amherst, May 1996.

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

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second Symposion on Principles and Practice of Parallel Programming*. ACM, 1990.

[Cor92] J.C. Corbett. Verifying general safety and liveness properties with integer programming. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, Montreal, Canada, July 1992.

[Cor96] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[DAC99] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of 21st International Conference on Software Engineering*, May 1999.

[DBDS93] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Using state space methods for deadlock analysis in Ada tasking. *Software Engineering Notes*, 18(3):51–60, July 1993. Proceedings of the International Symposium on Software Testing and Analysis.

[DC96] M.B. Dwyer and L.A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, March 1996.

[DKM+94] Laura K. Dillon, G. Kutty, Louise E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[DP98] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998. to appear.

[DS91] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, October 1991.

[DS97] M.B. Dwyer and D.A. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*, October 1997.

[Dwy95] M.B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts at Amherst, September 1995.

[Dwy97] M.B. Dwyer. Modular flow analysis for concurrent software. In *Proceeding of the 12th IEEE Conference on Automated Software Engineering*, November 1997.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[GS93] D. Grunwald and H. Srinivasan. Efficient computation of precedence information in parallel programs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[Gun92] C. Gunter. *Semantics of Programming Languages, Structures and Techniques*. MIT Press, 1992.

[GW91] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Second Workshop on Computer Aided Verification*, pages 417–428, July 1991.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.

[HL85] D. Helmbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[Hol88] G.J. Holzmann. An improved reachability analysis technique. *Software: Practice and Experience*, 18(2):137–161, February 1988.

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[How86] W.E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12:997–1004, October 1986.

[HR81] L.H. Holley and B.K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[KSV96] Jens Knoop, Bernhard Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18:268–299, 1996.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Kur85] R.P. Kurshan. Modeling concurrent processes. In B. Gopinath, editor, *Computers and Communications*, volume 31 of *Proceedings of Symposia in Applied Mathematics*, pages 45–57, 1985.

[Mas93] S.P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Rutgers University, May 1993.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mer92] N. Mercouroff. An algorithm for analyzing communicating processes. In *Proceedings of Mathematical Foundation of Programming Semantics '91*, Pittsburgh, PA, March 1992. published in LNCS 598.

[Mil79] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[MMR95] S.P. Masticola, T.J. Marlowe, and B.G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MR93] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[NA98] G.N. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of stat ements that may happen in parallel. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 1998.

[NAC99a] G.N. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of 21st International Conference on Software Engineering*, May 1999.

[NAC99b] G.N. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *Proceedings of the ESEC/FSE 99, Joint 7th European Software Engineering Conference(ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, September 1999.

[NACO97] G.N. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying static analysis to software architectures. In *LNCS 1301*. The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997.

[NCC99] G.N. Naumovich, L.A. Clarke, and J.M. Cobleigh. 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*, September 1999.

[NCO96] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.

[NCO98] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. 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*, June 1998.

[NM90] R.H. Netzer and B.P. Miller. Detecting data races in parallel program executions. In *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[OO90] K.M. Olender and L.J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[OO92] K.M. Olender and L.J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[Pnu85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1985.

[RS90] J.H. Reif and S.A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1), 1990.

[SAIC97] Science Applications International Corporation. Advanced interoperability technology development: Investigating static data flow analysis for advanced distributed simulation verification. Technical report, SAIC, Orlando, FL, May 1997.

[Sch98] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Programming Languages*, January 1998. to appear.

[SMBT90] S.M. Shatz, K. Mai, C. Black, and S. Tu. Design and implementation of a petri net based toolkit for ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[Ste93] Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *International Journal on Science of Computer Programming*, 21:115–139, 1993.

[Tay83a] R.N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26, May 1983.

[TBC+88] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of SIGSOFT88: Third Symposium on Software Development Environment*, pages 1–13, November 1988. Published as ACM SIGPLAN Notices 24(2) and as SIGSOFT Software Engineering Notes, 13(5) November 1988.

[TO80] R.N. Taylor and L.J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–277, May 1980.

[Val90] A. Valmari. A stubborn attack on state explosion. Computer Technology Laboratory report 26.4.1990, Technical Research Centre of Finland, 1990.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1-2):72–93, 1983.

[WZ91] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[YTL+95] M. Young, R.N. Taylor, D.L. Levine, K.A. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64–106, January 1995.

[YY91] W.J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification*, pages 49–59, Victoria, Canada, October 1991.