Data Flow Analysis for Verifying Properties of Concurrent Programs

Matthew B. Dwyer
Lori A. Clarke

CMPSCI Technical Report 94–45
September 1994

Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Defense Advanced Research Projects Agency under Grant MDA972-91-J-1009.

Data Flow Analysis for Verifying Properties of Concurrent Programs *

Matthew B. Dwyer
Lori A. Clarke

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

In this paper we present an approach, based on data flow analysis, that can provide cost-effective analysis of concurrent programs with respect to explicitly stated correctness properties. Using this approach, a developer specifies a property of a concurrent program as a pattern of selected program events and asks the analysis to verify that all or no program executions satisfy the given property. We have developed a family of polynomial-time, conservative data flow analysis algorithms that support reasoning about these questions. To overcome the traditional inaccuracies of static analysis, we have also developed a range of techniques for improving the accuracy of the analysis results. One strength of our approach is the flexibility allowed in choosing and combining these techniques so as to increase accuracy without making analysis time impractical.

We have implemented a prototype toolset that automates the analysis for programs with explicit tasking and rendezvous style communication. We present preliminary experimental results using this toolset.

1 Introduction

The application of distributed and concurrent programming technology has moved from special purpose database and operating systems into the programming mainstream. Developers applying this technology to complex problems require cost-effective analysis techniques to gain confidence in the quality of their concurrent software. In this paper, we present an approach, based on *data flow analysis*, that has the potential to provide cost-effective analysis of concurrent programs with respect to explicitly stated correctness properties.

Although our approach is applicable to a wide range of concurrency and communication models, in this paper we restrict our discussion to programs with explicit tasking and rendezvous communication and illustrate our approach using Ada tasking programs. Using our approach, 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 ask the analysis to verify that all or no program executions satisfy the given property. We have developed a family of polynomial-time, conservative data flow analysis algorithms whose results can be used to address such questions.

To overcome the traditional inaccuracies of static analysis, we have developed a range of techniques for improving the accuracy of the results. *Prior* to analysis, *refinements* to the flow graph representation of the program, based on program and property specific information, increase the efficiency and accuracy of the subsequent analysis. *During* analysis, enforcement of *feasibility constraints*, which are based on the program being analyzed and the programming language in which it is written, improves the accuracy of the results. One strength of our approach is the flexibility allowed in choosing and combining these techniques, thus providing a means of controlling the tradeoff between accuracy of the analysis results and the execution time of the analysis. This results in an analysis approach that can provide high accuracy without making analysis time impractical.

Section 2 describes related work and discusses similarities and differences between prior research and our approach. We provide a high level overview of the basic analysis approach and state some theoretical results in Section 3. Following that, we describe techniques

*This work was sponsored by the Advanced Research Projects Agency under Grant # MDA972-91-J-1009.

for improving the accuracy and efficiency of the basic approach in Section 4. We discuss a prototype implementation of our analysis approach and present initial experimental results in Section 5. We conclude in Section 6 with a discussion of what we have learned about our analysis approach and plans for future work.

2 Related Work

There is a large body of research into automated techniques for reasoning about the behavior of concurrent programs.

State reachability approaches have been successfully applied to analyzing concurrent programs [Hol88, SMBT90, YTL+92]. Complexity results for reachability analysis [Tay83] imply that, in general, the size of the program state space and consequently the cost of analysis increases exponentially with program size. To address the need for scaling reachability analysis to large programs, researchers have investigated three techniques: 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 [BCM+90]. Although, in general, the costs of these techniques exhibit exponential growth, each has been successful in verifying properties of example programs.

An alternate analysis approach is to reason using necessary or sufficient conditions about a specified property. Necessary conditions of this form can be used to reason about whether all or no program executions satisfy a property [1]. In contrast, sufficient conditions of this form can be used to reason about whether some program execution satisfies a property. Researchers have used linear programming techniques to encode necessary [ABC+91] and sufficient [MSS89] conditions. Although, in the worst case, the cost of the linear programming technique is exponential, it has been successfully applied to a number of examples and allows for practical analysis of very large versions of those examples.

Data flow analyses have traditionally been based on similar necessary or sufficient conditions. Conceptually, data flow analysis involves a fix point computation over a flow graph of a pre-defined relation, which encodes the analysis question. In theory the class of relations that can be computed is very large; polynomial-time algorithms exist for a smaller but very useful class of relations [MR90]. For data flow analysis of sequential programs, the flow graph is often reducible and specialized algorithms can be applied that improve the execution time of analysis.

Data flow analysis of concurrent programs requires that inter-task communication be represented. Early model checking approaches [CES86] are essentially data flow analyses that use the program data and control state reachability graph as a flow graph; performance suffered from the impractical size of this graph, however. More typically, data flow analyses either represent potential communication with edges in the flow graph [CKS90, GS93, MR91] or label nodes representing communication statements so that they can be matched during analysis [CK93, Mer92]. The former approach is appropriate if we view the flow graph as a repository of information about possible program executions that is refined over time by a variety of analyses [MR93], although the resulting flow graphs are invariably irreducible. Our approach also employs a flow graph that represents communication explicitly.

Masticola and Ryder [MR91] describe an analysis approach for checking deadlock freedom in Ada tasking programs that uses data flow analyses [MR93] to improve the accuracy of the analysis results. These data flow analyses are in the spirit of the refinements described in Section 4.1. Our approach differs from that of Masticola and Ryder in that we advocate selective application of a more comprehensive set of refinements, do not iteratively apply all refinements until convergence, further improve accuracy through the use of feasibility constraints, and support analysis of a rich class of properties as opposed to just deadlock.

Olender and Osterweil [OO92] developed an analysis for sequential programs, based on necessary conditions, that uses a simple version of the state propagation algorithm described in Section 3.3. In addition to the increase in complexity associated with the analysis of concurrent programs, our approach incorporates a variety of mechanisms to increase the accuracy of state propagation analysis. We note that our analysis is applicable to sequential programs without modification.

A number of analyses represent programs and properties of interest as formal automata [GW91, Kur85, OO90]. At a high level, the set of executable program paths are represented as strings accepted by a program automaton and the set of program paths that satisfy the property are strings accepted by a property automaton. One approach to verifying properties in this context is to test if the language of the program automata is contained in the language of the property automata. The complexity of such a test varies with the power of the automata used. The state propagation analysis described in Section 3.3 is a polynomial-time, conservative test to determine language containment for finite state automata.

A number of formalisms have been developed to express properties of programs, such as temporal logics [Pnu85] and regular expression based formalisms [ABC+91, Kur85, OO90]. Unfortunately, reasoning using the most general of these formalisms can be inef-

[1] The question *do all executions satisfy the property?* is the dual of *is it false that no executions satisfy the negation of the property?*.

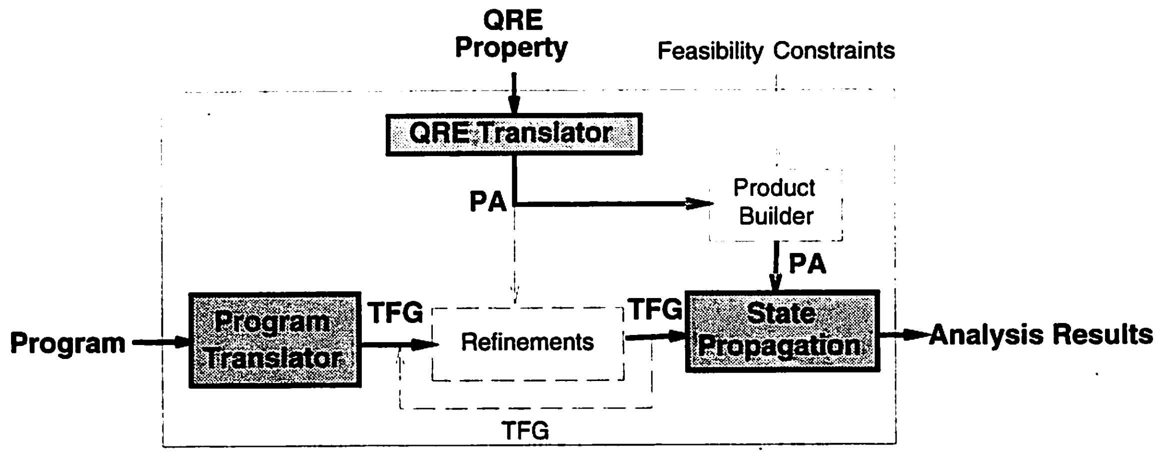

Figure 1: Architecture of Analysis Approach

ficient. 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, we will analyze properties in this less general but more practical setting.

3 Overview

In this section, we describe our flow graph representation for concurrent programs, the formalism for expressing program properties, and the basic analysis algorithm.

Figure 1 provides a high level view of the components and information flow in our analysis approach. For the basic version of our analysis the bold components and arrows in the figure are relevant; the dashed components and arrows can be ignored. The *program translator* constructs a *trace flow graph* (TFG) from the Ada source code for a program, or part of a program. After specifying the desired property as a *quantified regular expression* (QRE), the *QRE translator* constructs a *property automaton* (PA) that represents the property. The generic *state propagation* data flow analysis algorithm is instantiated for the PA and applied to the TFG. The results of state propagation analysis indicate whether the program satisfies the specified property.

To reduce the cost of analysis or increase its accuracy, *refinements* can be applied to the TFG or *feasibility constraints* can be incorporated into the PA by the *product builder*. Such improvements are optional and are thus depicted by dashed lines in Figure 1.

3.1 Program Representation

The trace flow graph is a conservative representation of executable program *event traces*, which are strings over the set of program events. A TFG can be thought of as an automaton that accepts event strings. Since it is conservative, it is guaranteed to accept all event strings

```
task body T1 is
begin
  x := TRUE; -- EVENT "a"
  T2.E1; -- EVENT "b"
end T1;
```

```
task body T2 is
begin
  accept E1; -- EVENT "b̄"
  x := FALSE; -- EVENT "c"
end T2;
```

Figure 2: Ada tasks for example

that correspond to program executions. As with most flow graph representations, it may also accept event strings that do not correspond to program executions.

In this paper we describe TFGs in terms of Ada language constructs and illustrate with examples of Ada programs. Figure 2 is a simple Ada tasking program that consists of two tasks and a single task communication, used to order the definitions of the boolean variable x.

Users describe an *alphabet of program events*, denoted by Σ, such that each symbol corresponds to the execution of some program event, such as a reference to a variable or execution of an entry call. All program event alphabets contain the τ symbol that corresponds to the program executing an event that is not of interest. A number of mechanisms for defining this alphabet are possible, such as using a fixed set of events, specifying the set of program entities of interest (e.g., variables, functions, task entries) and generating the set of events related to those entities, or introducing program comments that define events. For expediency, we use program comments to define events of interest as illustrated by the `-- EVENT "symbol"` comments in Figure 2. Note that for a calling event, say b, we represent a corresponding accept statement with a $\bar{b}$ symbol.

TFGs are constructed from a set of *control flow graphs* (CFG)s that represent the tasks of the program. Formally, a TFG is (N, E, S, T, L, P) where N is the set of nodes, E consists of 3 kinds of directed edges representing control flow, communication and poten-

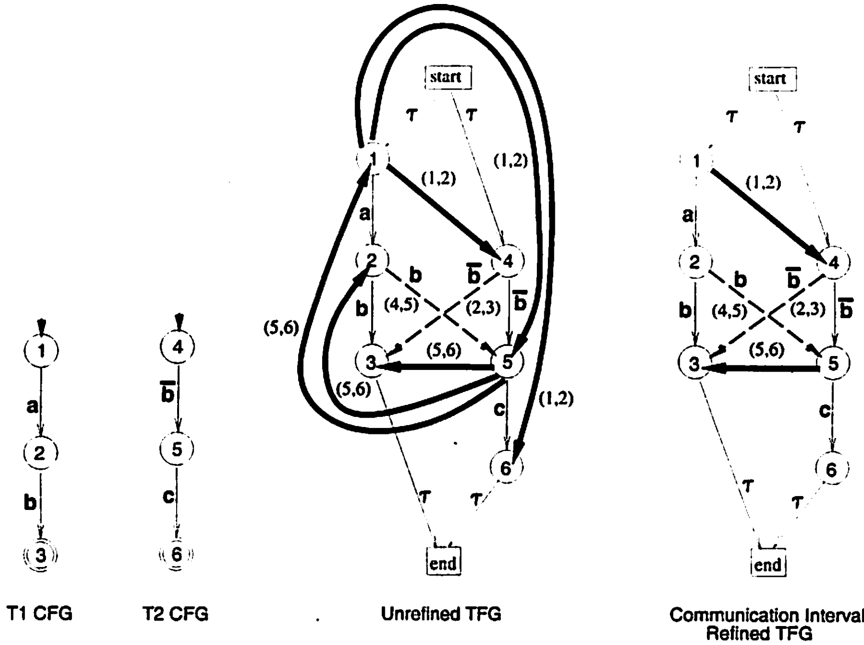

Figure 3: Trace Flow Graph for example

tial program event ordering information, $S \in N$ is the unique start node, $T \in N$ is the unique terminal node, $L : E \rightarrow \Sigma$ maps an edge to an event symbol, and $P : E \rightarrow E$ maps communication and ordering edges to an associated partner edge. The semantics of edge kinds and the edge partner mapping is discussed below. For the example in Figure 2 the CFGs for the tasks and the initial, unrefined TFG constructed from them are illustrated in Figure 3.

Nodes TFG nodes represent control states of individual tasks in the program. There is a TFG node for each node in the set of CFGs for the tasks in the program. In Figure 3 we number TFG nodes for identification purposes. Nodes $\{1,2,3\}$ correspond to control states of task T1, where node 1 is the start and node 3 the terminal state. Nodes $\{4,5,6\}$ correspond to control states of task T2, where node 4 is the start and node 6 the terminal state. We add two additional nodes `start` and `end` that represent the start and end states of the program. The outgoing TFG edges from a node represent the program events that can be executed when a task is in the state represented by the node.

Edges An edge is labeled by a symbol from the program event alphabet. For our example the alphabet is $\{a, b, \bar{b}, c\}$. Although not illustrated in Figure 3, multiple TFG edges can be labeled with the same symbol.

We represent program events that are local to a task as a TFG *control flow* (CF) edge whose source and destination represent control states in the same task. These are analogous to CFG edges. In the unrefined TFG of Figure 3, the solid edge (1,2), labeled by a, represents the execution of `x := TRUE` in task T1, and is derived from edge (1,2) of the CFG for T1. We also add τ labeled edges from(to) the `start`(`end`) node to(from) the nodes corresponding to the enter(exit) nodes of each CFG.

Like some other flow graphs for concurrent programs, TFGs represent task communication with explicit edges. We extract the set of potential communications from the CFGs by matching the labels on CFG edges, for example b and $\bar{b}$. We are interested in capturing the fact that the TFG nodes that follow a communication have two predecessors, one in the same task as the node, which is represented by a CFG edge, and the other in another task with a matching communication statement. We introduce a *communication* (COM) edge to capture the communication predecessor in the other task. To illustrate, in the unrefined TFG in Figure 3 the predecessors of node 3 are node 2, by way of the control flow edge, and node 4, by way of the, dashed, communication edge. Similarly, node 5 has a pair of incoming control flow and communication edges. Control flow edges that represent Ada entry call and accept statements may have multiple associated communication edges. Each communication edge, however, has a single control flow edge associated with it: we refer to that control flow edge as the *communication partner*. A communication edge is annotated with the identity of its partner. We note that accept statements with bodies are modeled as separate start and end communications.

Unlike other concurrent flow graph representations, TFGs capture the potential interleaving of asynchronously executing program events. We represent pairs of edges that may interleave execution order by introducing additional edges into the TFG. For each non-τ labeled control flow or communication edge exiting a TFG node we add a *may immediately precede* (MIP) edge from that node to all nodes in other tasks. We refer to the control flow or communication edge associated with a MIP edge as a *MIP partner* and annotate the MIP edge with the identity of its partner. The semantics of a MIP edge are that the event labeling its partner may immediately precede execution of any event initiated from the destination node of the MIP edge. To illustrate, consider the unrefined TFG in Figure 3. Nodes *start*, 3 and 6 have no exiting MIP edges because all of their exiting edges are labeled τ. Node *end* has no exiting edges of any kind. Node 1 has a single exiting control flow edge (1,2) labeled by a. We add, bold, MIP edges (1,4), (1,5), (1,6) to represent the possibility that event a may immediately precede events $\bar{b}$ exiting node 4, c exiting node 5, and τ exiting node 6. The MIP edges leaving node 5 are constructed analogously. [2]

In constructing a TFG from a collection of CFGs for an Ada program we make use of the fact that rendezvous communication requires that two tasks synchronize to communicate. Therefore, communication events between the tasks cannot execute asynchronously with respect to any of the other events in the two tasks. Thus, if we have a communication edge we need not add any MIP edges for that edge. In our example, all edges exiting node 2 represent communication with task T2 so we need not include MIP edges for them; node 4 is treated analogously. We note that if we had non-communication related control flow edges exiting node 2, then we would introduce MIP edges for only those control flow edges.

It is clear from this simple example that the TFG overestimates the executable event orderings of the program. In Section 4.1 we discuss how to eliminate some unexecutable orderings. Finally, we note that there are potentially many MIP edges in a TFG; this is not surprising since the number of pairs of asynchronously executing program statements can be large.

We state two important results on the size and structure of TFGs: they are irreducible graphs and $|E| = \mathcal{O}(|N|^3)$. There have been a number of data flow algorithms developed that are applicable only to reducible graphs that are thus not applicable to TFGs. The complexity of data flow algorithms is often expressed in terms of the number of flow graph nodes, under the assumption that the number of edges is at most the square of the number of nodes. TFGs violate this assumption and, therefore, we must evaluate the complexity of data flow algorithms in this light.

[2] Although not illustrated by the example in this paper, to more accurately model a program, TFGs can be constructed that use multiple CF edges to represent a single program statement and that duplicate and specialize portions of the TFG that represent Ada accept bodies.

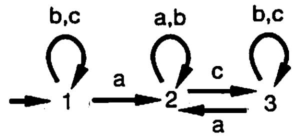

Figure 4: Property Automaton for Response Property

For simplicity, in the remainder of the paper we treat the symbols representing call and accept statements for the same entry as identical. That is, b and $\bar{b}$ will now be treated as b in our example.

3.2 Property Representation

To specify properties, we use a part of the Cecil formalism [OO90] called quantified regular expressions. QREs have three components: a property alphabet $\Sigma_{property}$, a quantifier Q, and a regular expression R. Syntactically, QREs look like $\{\Sigma_{property}\}QR$, where $\Sigma_{property} \subseteq \Sigma$, $Q \in \{\forall, \exists\}$, and R is a regular expression over $\Sigma_{property}$. The alphabet is a set of symbols representing distinct program events. The quantifier determines whether we test if the pattern described by R is exhibited on all program executions or on no program execution. The regular expressions in this formalism are standard and include concatenation $(a;b)$, disjunction $(a|b)$, Kleene closure (a^*) and difference $(\{a,b\}-b)$ of symbols and sets of symbols from $\Sigma_{property}$.

We have been able to specify a number of useful properties in a stylized form of QREs over two classes of sub-expressions: intervals that exclude a set of events, of the form $(\Sigma_{property} - \{a,b\})^*$, 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]. Data races, mutual exclusion, general forms of invariance, response and precedence properties have been specified with this form of QRE.

As an example consider an event based formulation of a response property. If we have events a, b, and c, we can specify "after an a event occurs, eventually a c event will occur" as

$\{a,b,c\}\forall$

$(\Sigma_{property} - a)^*; (a; (\Sigma_{property} - c)^*; c; (\Sigma_{property} - a)^*)^*$

Using standard techniques we can construct from the regular expression of a QRE a finite automaton that we call a *property automaton* (PA). A property automaton accepts all event strings over the property alphabet that correspond to the property of interest. Formally, a PA is $(T_{PA}, \Sigma, \delta, S_{PA}, A_{PA})$ where T_{PA} is the set of automaton states, Σ is the alphabet, $\delta : T_{PA} \times \Sigma \rightarrow T_{PA}$ is the state transition function, $S_{PA} \in T_{PA}$ is the start state, and $A_{PA} \subseteq T_{PA}$ are the accepting states. Figure 4 illustrates the PA for the response QRE described above. Many PAs contain a non-accepting state that has no exiting, non self-loop transitions. Such states represent the fact that a string leading to that state has

```
Input:  TFG and a PA
Output:  the States relation
Algorithm:
  1) States(start node of TFG) = S_PA
     all other States values = ∅
  2) initialize worklist to exit edges of start node
  3) while worklist is not empty
    4) dequeue next edge from worklist
    5) compute current states for source(s) of edge
    6) for each current state, s,
      7) States(destination of edge) ∪ = δ(s, L(edge))
    8) add edges whose source has new States
       values to worklist
```

Figure 5: State Propagation Algorithm

violated the property in such a way that no extension of the string can possibly satisfy the property; we call these states *trap* states.

3.3 State Propagation Analysis

One approach to analyzing properties specified as QREs is to submit each TFG event trace to the PA and check if the PA accepts it. This is impractical because the number of TFG event traces is, in general, infinite. Our analysis collapses event traces that lead to the same PA state into a single value and thus greatly reduces the amount of work required to analyze QREs. This can be implemented by a data flow analysis algorithm. We use a standard iterative worklist algorithm [Hec77] with modifications to enforce conditions related to the semantics of TFG edges. The main data structures are a *worklist* of TFG edges and *States*, the set of PA states for each TFG node. Figure 5 gives a high level description of the algorithm. In step 5 we consider separately communication edges and their partners, control flow edges, MIP edges that have control flow edges as partners, and MIP edges that have communication edges as partners. For communication edges we use the fact that two statements can successfully rendezvous only if they are reached simultaneously on the same program execution. In our model this corresponds to having the same event trace lead to the source nodes of the communication edges for a given rendezvous. The algorithm enforces this condition by only propagating PA states that are common to the source nodes of a communication pair across a communication edge.

This simple version of the state propagation algorithm has a worst case bound on its running time of $O(|T_{PA}|^2|E|)$ or equivalently $O(|T_{PA}|^2|N|^3)$. We note that the value of *States* for each TFG node moves monotonically up the PA state powerset lattice as the algorithm progresses. We have developed a slightly more complicated version of the algorithm that, for negligible cost, keeps track of the state values considered for each TFG edge. For this algorithm we can bound the number of calls to δ by $|E||T_{PA}|$. The bound on the running time of this more complicated algorithm is $O(|T_{PA}||E|)$ or equivalently $O(|T_{PA}||N|^3)$.

3.4 Results of State Propagation

Complete program executions correspond to event traces that start at the entering node of the TFG and end in states that correspond to a termination state for each task in the program. We capture this information by intersecting the *States* values for the TFG nodes that correspond to termination states for each task. This is a conservative estimate of the PA states that can be reached by complete executions of the concurrent program.

Analysis of a QRE that specifies the $\forall$ quantifier requires a test for language containment. If the intersection of *States* values for task termination nodes is a subset of the accepting PA states, then all executable program traces satisfy the property; we say that the *language containment* test is true. If the result is false, however, we say it is *inconclusive* because the TFG trace(s) that did not satisfy the property may or may not be feasible.

Analysis of a QRE that specifies the $\exists$ quantifier requires a test for non-empty language intersection. If the intersection of *States* values for task termination nodes contains no accepting PA state, then there exists no executable program trace that satisfies the property; we say that the *non-empty language intersection* test is false. If the result is true, however, we say it is *inconclusive* because the TFG trace(s) that satisfied the property may or may not be feasible.

While inconclusive results are insufficient for verifying properties of programs they can provide useful analysis information. In particular, post-processing of the *States* values may be able to determine the feasibility of the traces in question.

4 Increasing Efficiency and Accuracy

Cost-effective data flow analysis requires that testing of necessary conditions encoded in the analysis is efficient and that the conditions are strong enough to provide conclusive results in most cases.

The worst case time complexity bounds of our state propagation algorithms indicate that the efficiency of state propagation analysis is strongly dependent on the number of edges and nodes in the TFG. We introduce TFG refinements into the basic analysis illustrated in Figure 1 to reduce the number of edges and nodes such that analysis of the resulting TFG is more efficient without sacrificing accuracy.

The structure of the TFG enforces some, but not all, event ordering constraints of the program. The state propagation algorithm enforces constraints on task synchronization at communication and program termination states. There are three major sources of inaccuracy in the TFG representation: unexecutable control flow paths, unexecutable communication between matching statements, and unexecutable orderings of asynchronous program events. We introduce TFG refinements into the basic analysis illustrated in Figure 1 to eliminate sources of inaccuracy that are independent of particular TFG paths. We introduce feasibility constraints and a mechanism for combining them with the property automaton into the basic analysis illustrated in Figure 1 to eliminate sources of inaccuracy that depend on particular TFG paths.

4.1 Refinements

There are two goals in refining the TFG: reducing its size and eliminating behaviors that are never executed. Reducing the size of the TFG will reduce the cost of performing state propagation. Eliminating unexecutable behaviors strengthens the conditions encoded in the TFG and consequently improves the accuracy of state propagation. We have developed refinements that reason about a variety of program information. Many refinements only consider a subset of the TFG edges. For example, in the description of communication intervals that follows we do not need to consider MIP edges. Therefore the cost of performing refinements is only dependent on the size of the relevant parts of the TFG. We describe two of our refinements.

Alphabet Refinement For state propagation of a given property to be correct, the TFG need only represent the set of events in the property alphabet and the ordering relationships between those events. We can perform *alphabet refinement* of a TFG for a given property by relabeling with τ the edges in the TFG that do not have labels in the property alphabet. The resulting TFG can be further transformed to eliminate edges and nodes that do not add to the set of traces over the reduced event alphabet. These transformations are accomplished through application of a partition refinement algorithm whose running time is bounded by a low-order polynomial in the number of TFG nodes. We will see in Section 5 that this simple refinement can have a great effect on the size of the TFG and on the cost of state propagation analysis.

Communication Interval Refinement Many programs contain critical sections, transaction like structures, regions of mutual exclusion, protocols for acquiring and releasing resources, etc. We have developed *communication interval refinement* to eliminate behaviors from the TFG that can never be executed based on the semantics of these kinds of program structures. A communication interval consists of a pair of execution regions in distinct tasks that are bounded by a common pair of statically recognizable communication events, called the *start* and *end* events of the interval. We consider the start and end of the program as implicit communication events. Intuitively, an interval consists of regions of statements in two tasks such that if one task is executing in the region associated with the interval, then the other task must be executing in its associated region. We can represent an interval as a set of TFG nodes such that each node lies between the start and end events in each of the tasks involved in the interval. For the unrefined TFG in Figure 3 there are two communication intervals involving tasks T1 and T2. Interval 1 has the start of the program as its *start* event and b as its *end* event, and consists of nodes 1 and 2 in task T1 and node 4 in task T2. Interval 2 has b as its *start* event and the end of the program as its *end* event, and consists of node 3 in task T1 and nodes 5 and 6 in task T2.

Events in different tasks that lie in disjoint intervals can never be immediate predecessors of each other because there is always an intervening event, the start or end event of the interval. We can, therefore, eliminate MIP edges that cross the boundaries of intervals. The refined TFG in Figure 3 illustrates the results of applying communication interval refinement to the unrefined TFG. The MIP edges (1,5), (1,6), (5,1) and (5,2) all cross interval boundaries [3]. The cost of performing communication interval refinement is dominated by the cost of computing control flow dominator and post-dominator relations for the control flow graph of each task and considering each pair of communicating tasks, both of these are bounded by a low-order polynomial in the number of TFG nodes. We will see in Section 5 that this refinement can have a great effect on the size of the TFG and the cost and accuracy of state propagation analysis. We note that communication interval refinement can be viewed as a generalization of Masticola and Ryder's critical section analysis [MR93].

Figure 6 illustrates the working of the state propagation algorithm in Figure 5, with the communication interval refined TFG from Figure 3 and the PA from Figure 4. The rows are presented in the order that they are taken off of the worklist in line 4 of the algorithm. Note that the computation of the current source *States* value is dependent on the edge kind. For communication edges we intersect the *States* values for the source COM edge and its partner. Note also that when considering a MIP edge we use the label of its partner edge in evaluating δ. Since the QRE for this property uses a $\forall$ quantifier we check that $States(3) \cap States(6) = \{2,3\} \cap \{3\} \subseteq \{1,3\} = A_{PA}$ to verify that the response property holds on all executions of the example program.

[3] For this simple example, there are other ordering refinements that can determine the unexecutability of these edges.

Edge from *worklist*	Edge kind	Edge Partner	Current Source *States*	Current Dest *States*	Line 7 of State Propagation Algorithm
(start,1)	CF	-	$\{1\}$	$\emptyset$	$States(1) = \emptyset \cup \delta(1, \tau) = \{1\}$
(start,4)	CF	-	$\{1\}$	$\emptyset$	$States(4) = \emptyset \cup \delta(1, \tau) = \{1\}$
(1,2)	CF	-	$\{1\}$	$\emptyset$	$States(2) = \emptyset \cup \delta(1, a) = \{2\}$
(1,4)	MIP	(1,2)	$\{1\}$	$\{1\}$	$States(4) = \{1\} \cup \delta(1, a) = \{1,2\}$
(4,3)	COM	(2,3)	$\{1,2\} \cap \{2\} = \{2\}$	$\emptyset$	$States(3) = \emptyset \cup \delta(2, b) = \{2\}$
(2,5)	COM	(4,5)	$\{2\} \cap \{1,2\} = \{2\}$	$\emptyset$	$States(5) = \emptyset \cup \delta(2, b) = \{2\}$
(3,end)	CF	-	$\{2\}$	$\emptyset$	$States(end) = \emptyset \cup \delta(2, \tau) = \{2\}$
(5,6)	CF	-	$\{2\}$	$\emptyset$	$States(6) = \emptyset \cup \delta(2, c) = \{3\}$
(5,3)	MIP	(5,6)	$\{2\}$	$\{2\}$	$States(3) = \{2\} \cup \delta(2, c) = \{2;3\}$
(6,end)	CF	-	$\{3\}$	$\{2\}$	$States(end) = \{2\} \cup \delta(3, \tau) = \{2,3\}$

Figure 6: Example Operation of State Propagation Algorithm

4.2 Feasibility Constraints

In contrast to refinements, feasibility constraints (FC) attempt to improve the accuracy of state propagation with respect to particular TFG paths. Conceptually, a feasibility constraint represents a necessary condition that must be satisfied for a path through the TFG to correspond to a program execution. We have developed feasibility constraints that encode necessary conditions related to the ordering of events local to a task, the ordering of events that are global to the entire program, the relative number of occurrences of related program events in the entire program, and the values of program variables. Each of these can be encoded as a finite automaton, and a collection of these can be combined by the product builder, as described below, to enforce the conjunction of the necessary conditions of the constraints during state propagation analysis. We will see in Section 5 that this can have a great effect on the accuracy of state propagation analysis. We illustrate the approach by discussing the local task event ordering constraint.

Task Automata The presence of MIP edges in the TFG introduces paths that may violate event orderings encoded as control flow edges in the TFG. The refinements of the previous section help improve the precision of the TFG by eliminating unnecessary MIP edges. We have developed a technique for enforcing the control flow orderings for a single task during state propagation by encoding the state and state transitions of an individual task as a finite automaton, called a *task automaton* (TA). During state propagation, the TA restricts the analysis to consider only TFG paths that correspond to the set of legal event sequences for the modeled task.

A TA is constructed from the control flow graph for a task, where the enter and exit nodes of the CFG determines the start and accept states of the TA. In general, symbols in the TFG alphabet do not uniquely label TFG edges. Keeping track of task state in the TA requires that we know when relevant control flow edges in the

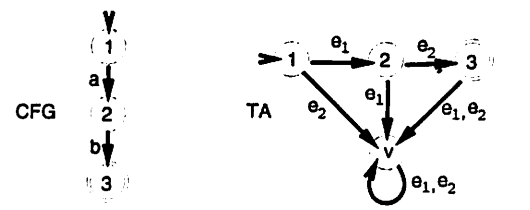

Figure 7: CFG and TA for task T1 of example

TFG are considered rather than the symbols that label the edges. Thus, the alphabet of the TA is a set of unique edge identifiers for the control flow edges of the modeled task. We add a non-accepting *violation* state to represent that a path in the TFG violates the ordering of events in this task. We add transitions from all TA states to the *violation* state for all symbols in the TA alphabet for which the state has no exiting transition. Figure 7 illustrates the CFG and TA for task T1 of the example in Figure 2. Each edge in the task CFG has a corresponding transition in the TA, thus symbol e_1 identifies control flow edge (1,2) and e_2 identifies the control flow edge (2,3). There are four states: violation v and one for each node of T1's control flow graph. Finally, the state associated with the end node of the task CFG, for this example state 3, is the only accepting state.

We note that to enforce the conditions encoded in a TA during state propagation, the TFG alphabet must include the symbols of the TA alphabet. This typically involves converting the TFG to use unique edge identifiers as edge labels and replacing PA transitions on each symbol with a transition for each edge in the TFG that is labeled by such a symbol. In the example, for instance, we would replace transitions on a with transitions on e_1.

Product Builder Given a PA and feasibility constraints encoded as finite automata we can use standard automata theory to construct a product automaton, which we call a *constrained property automaton* (CPA). As we introduce more feasibility constraints, the number of states and the size of the alphabet of this product automaton grows rapidly, with a negative impact on the performance of state propagation. To combat this we have developed techniques reduce the number of states and number of symbols in the alphabet of the CPA.

One such technique reduces the size of the CPA alphabet by recognizing that rather than using the cross-product of the PA and FC alphabets, we can use unique identifiers for the control flow and communication edges of the TFG. Prior to state propagation, we can easily transform the TFG labels, PA transition symbols, and FC transition symbols to this common edge alphabet. This can significantly reduce the complexity of δ, the CPA transition function.

We have developed a technique that collapses collections of equivalent CPA states into a single state. As mentioned above, 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. Therefore, we can collapse all CPA states that represent a violation state in any of the FC automata into a single violation state without losing accuracy.

CPA states that are unreachable do not contribute to the time performance of state propagation but they do require unnecessary space to represent δ. When transitions in multiple FC automata or the PA share a label, the CPA may contain states that are unreachable. These states are easily detected and removed from the CPA.

A final improvement to the CPA does not reduce its size but it does improve the accuracy of analysis. We apply refinement information to increase the accuracy of the CPA that are built from one or more TA by directing transitions on edges that are unexecutable at a given TA state to the violation state.

These techniques resulted in a dramatic reduction in the size of the CPAs used in state propagation for the examples in Section 5.

Interpreting the results of state propagation of a CPA, as opposed to a PA, has one slight difference from the description in Section 3.4. 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 of TFG paths that violate any of the conditions enforced by the FCs encoded in the CPA.

5 Empirical Evaluation

The two main goals of our empirical evaluation are to demonstrate the feasibility of the analysis approach and to provide feedback on the effectiveness of the analysis components. As discussed in Section 3.1, in the worst case the number of TFG edges is cubic in the number of nodes. One goal is to assess whether this bound is approached in practice. In addition, we want to validate that accurate analysis results can be obtained without excessive use of feasibility constraints. Note that if we use a TA for each task in the program, then state propagation of the resulting CPA will be as accurate, and as costly, as global reachability analysis.

Although detailed characterization of the execution performance of our analysis was not an explicit goal of this evaluation, we are able to present some preliminary data on the execution performance of state propagation analysis.

Prototype Implementation

We have developed prototype implementations of a number of components of the analysis architecture as depicted in Figure 1. In particular, the program translator and a simple version of the state propagation algorithm are fully automated, and detection of communication intervals is partially automated. We have well defined algorithms for all the other components, including an algorithm for constructing an automaton that enforces constraints based on variable access called a *variable automaton* (VA). The algorithms that are not automated were applied manually for the experiments discussed in this section.

Empirical Results

We consider three examples: the simple example in Figure 2, the controller task from a readers/writers problem [ABC+91], and a simple protocol [Cor92]. We measure the cost of executing the state propagation algorithm, described in Section 3.3, in terms of the number of TFG edges taken off of the worklist and the number of calls to δ, the CPA transition function. Figure 8 summarizes data for the experiments discussed below [4]. The experiments included communication interval refinement (CI), alphabet refinement (A), variable automata (VA), task automata (TA) and refinement of the CPA based on communication intervals (CPAR). The analysis results indicate that either *all* executions satisfy the property, *no* executions satisfy the property, or the results are *in*conclusive.

Example Experiment We analyzed 2 $\exists$ and 2 $\forall$ properties on TFGs representing the simple example introduced in Section 3. We attempted to verify the response property described in Section 3.2 and illustrated in Figure 4. For this simple example there is a single executable program trace over the alphabet $\{a, b, c\}$, namely *abc*. We attempted to verify that all the execution traces represented in the TFG exhibit this lone executable trace. We also attempted to verify that two unexecutable traces were exhibited by no trace in the

[4] Some of the QREs are too large to fit in the table and are included in the text with reference numbers.

#	Program	Tasks	Refinements Constraints	QRE	Result	TFG Nodes	TFG Edges	CPA States	State Propagation Edges	State Propagation δ Evals
1	example	2		{a,b,c}∃ac	inc	8	16	3	24	50
2	example	2		{a,b,c}∃abac	no	8	16	6	24	43
3	example	2		Figure 4	all	8	16	3	19	24
4	example	2		{a,b,c}∀abc	all	8	16	4	24	55
5	example	2	CI	{a,b,c}∃ac	no	8	12	3	10	10
6	example	2	CI	{a,b,c}∃abac	no	8	12	6	11	12
7	example	2	CI	Figure 4	all	8	12	3	11	12
8	example	2	CI	{a,b,c}∀abc	all	8	12	4	11	12
9	control	1		QRE 1	inc	22	29	3	67	125
10	control	1	A	QRE 1	inc	6	9	3	13	25
11	control	1	A,VA	QRE 1	all	11	14	8	32	57
12	protocol 2	4	CI	QRE 2	inc	14	78	3	158	275
13	protocol 2	4	CI,TA	QRE 2	inc	14	78	5	129	340
14	protocol 2	4	CI,TA,CPAR	QRE 2	all	14	78	5	105	221
15	protocol 4	6	CI,TA,CPAR	QRE 2	all	24	232	5	444	747
16	protocol 6	8	CI,TA,CPAR	QRE 2	all	34	466	5	862	1812
17	protocol 8	10	CI,TA,CPAR	QRE 2	all	44	780	5	1616	2802
18	protocol 8	10	CI,TA,CPAR	QRE 3 with m=0	no	44	780	5	1761	3574
19	protocol 8	10	CI,TA,CPAR	QRE 3 with m=2	no	44	780	13	4076	16371
20	protocol 8	10	CI,TA,CPAR	QRE 3 with m=4	no	44	780	21	6318	37893
21	protocol 8	10	CI,TA,CPAR	QRE 3 with m=6	no	44	780	29	8560	68023

Figure 8: Data for Experiments

TFG. We performed state propagation analysis on the two versions of the TFG illustrated in Figure 3.

The data in Figure 8 for experiments 1 through 8 indicates that in all cases analysis of the refined TFG was considerably less costly than for the unrefined TFG. For the property {a,b,c}∃ac, analysis of the unrefined TFG was inconclusive; communication interval refinement of the TFG improved the accuracy of the analysis results so that the property could be verified.

Controller Experiment Many programs encode state information in variables that is crucial to the success of static analysis. The controller task from the readers/writers example enforces the order in which it accepts entry calls on its 4 entries by setting and testing the state of two local variables. The code for this example is given in appendix A with communication events and variable access events indicated.

We specified the property that every bW, representing a communication over the `Start_Write` entry, is followed by an eW, representing a communication over the `Stop_Write` entry, without an intervening bW as the QRE:

(1) $\{bW, eW, bR, eR\}\forall$

$$(\Sigma - bW)^*(bW; (\Sigma - \{bW, eW\})^*; eW; (\Sigma - bW)^*)^*$$

The PA for this QRE has 3 states including a trap state. We analyzed this property for three TFGs for this task: an unrefined TFG representing all communication statements and variable accesses, a TFG refined to have an alphabet of just the communication events, and a TFG refined to have an alphabet of communication events and accesses to the `WriterPresent` boolean variable. As mentioned in Section 4.1, the alphabet refinement has no effect on the accuracy of analysis results, but it did have an appreciable impact on the cost of analysis. Unfortunately, the results were still inconclusive. Increasing the TFG alphabet and incorporating a VA provided sufficient accuracy to verify the property.

Experiments 9 through 11 illustrate how the combination of TFG refinement and feasibility constraints can provide increased accuracy while decreasing the execution cost of the state propagation algorithm. These experiments also illustrate how our analysis approach supports reasoning about tasks and groups of tasks independent of the rest of the program.

Protocol Experiment The code for the protocol example with 2 clients is given in appendix A. The property that every header from client 1, $h1$, was followed by a packet from client 1, $p1$, without any intervening header sent was specified as the QRE:

(2) $\{h1, p1, \ldots, hn, pn\}\forall$

$$(\Sigma - h1)^*(h1; (\Sigma - \{p1, h1, h2, \ldots, hn\})^*; p1; (\Sigma - h1)^*)^*$$

Where there are n clients, each with unique h and p events. The PA for this QRE has 3 states including a trap state.

We analyzed this property for three TFGs representing a 2 client version of the protocol program. The first TFG has an alphabet of communication events and was refined by communication intervals. The second incorporated a TA for `Client1` into the CPA, resulting in 2 additional states. This required converting the TFG to the appropriate edge alphabet. The final version of the 2 client program applied the results of communication interval refinement to the CPA. This introduces no new CPA states, although it does change a number of transitions. The TFG for all of these experiments was the same size but only the communication interval refined CPA version provided sufficient accuracy to verify the property.

Experiments 12 through 14 illustrate that accuracy can be gained by using feasibility constraints while decreasing execution cost. In contrast to the the controller example, refinements were not solely responsible for the decrease in execution cost. The ability of the CPA product building algorithm to add constraints without greatly increasing the number of states in the CPA was an important factor in keeping analysis cost low.

We conducted experiments to get a sense of how analysis cost scales with the size of the TFG and CPA. In Figure 8, experiments 14 through 17 are identical except for increasing the number of clients. We note that the `LockManager` task scales in complexity with additional clients. All of these experiments successfully verified the property described above. In Figure 8, experiments 18 through 21 are identical except for increasing the number of CPA states. We attempted to show that the QRE

(3) $\{h1, p1, \ldots, hn, pn\}\exists$

$$(\Sigma - h1)^{*}(h1; (\Sigma - \{p1, h1, h2, \ldots, hn\})^{*}; p1; (\Sigma - h1)^{*})^{m}; h1$$

was satisfied by no program execution, where m is a parameter used to increase the size of the CPA. All of these experiments successfully verified the property described above.

Experimental Observations

Although the limited data presented here is insufficient for drawing statistically valid conclusions about the rate of growth of state propagation analysis, we note a number of trends in the data. For all of the experiments in this section the number of TFG edges is considerably less than the square of the number of nodes. For the protocol example with increasing numbers of clients, the measures of execution cost for the state propagation algorithm appears to grow as the square of the number of nodes and linearly in the number of edges. The number of edges taken off the worklist appears to grow linearly with the number of CPA states and the δ evaluations appears to be grow faster than linear but less than quadratic in the number of CPA states. This last point is due to the fact that the state propagation algorithm currently implemented in our prototype is the simple version described in Section 3.3. Using the more efficient algorithm will reduce this cost. Thus, it appears that the bounds on the number of edges in the TFG and the bound on execution time of the state propagation algorithm are not indicative of the practical performance of the analysis.

6 Conclusion

We have presented an analysis approach based on polynomial-time data flow analysis algorithms that analyze whether a concurrent program satisfies an explicitly stated correctness property.

Our initial prototype implementation and algorithms have provided valuable information on the contributions of refinements and feasibility constraints to the efficiency and accuracy of state propagation analysis. For example, communication interval refinements have been generalized from their initial incarnation based on feedback from our experiments. The accuracy of the state propagation algorithm was improved by enforcing task synchronization at communication points. In addition, the set of TFG edges that need to be considered during state propagation has been reduced without loss of information. Also, we developed a faster version of the state propagation algorithm, and the CPA construction algorithm has been improved from its initial incarnation by adding violation state collapsing and dead state elimination.

In contrast to most of the methods described in Section 2, our analysis approach is based on algorithms with low-order polynomial bounds on the running time. Our preliminary empirical evaluations suggests that accuracy sufficient for verifying correctness properties of concurrent programs may be obtained in an analysis whose cost is bounded by a low-order polynomial in the size of the program. Although our results are preliminary, the accuracy and efficiency of state propagation analysis for the examples presented in this paper are encouraging.

A rich class of properties specified as patterns of program events is supported. Specification of some global properties however, such as deadlock, can require very large PAs and the cost of analysis may be impractical for these properties. One of our goals is to study a number of properties of concurrent programs to understand how the cost-effectiveness of our analysis varies with the property under analysis.

One of the strengths of our approach is its flexibility. Users have the ability to control the tradeoff between accuracy of the analysis results and the execution time of the analysis. The analysis architecture allows for a wide variety of TFG refinements and feasibility constraints to be incorporated into the analysis. One of our goals is to study the relative effectiveness of existing and new refinement strategies with respect to increasing the accuracy of analysis results and decreasing analysis cost. While this paper describes FCs only as finite automata, recent work has extended the analysis algorithms and architecture to allow a more general class of data flow problems to be solved in conjunction with state propagation to eliminate unexecutable program traces from consideration. We intend to use this increased flexibility to explore a wider range of tradeoffs between analysis cost and accuracy than was previously possible.

Programs can scale in a number of dimensions, such as the number of tasks, complexity of control and data structures, and communication structures. The examples considered in this paper are relatively simple. We are in the process of fully automating all of the analysis components. Using these tools we plan on performing a number of case studies of large, complex concurrent programs to investigate the extent to which our analysis can be applied to realistic software analysis problems.

It has been suggested that no single analysis technique is suitable for analysis of all properties of all concurrent programs. We intend to compare the cost-effectiveness of our analysis approach with a number of existing analysis techniques. This comparison will be carried out by attempting to characterize the effectiveness of these analyses on programs and properties found in the literature. In this way we hope to contribute to an understanding of the relative strengths and weaknesses of each analysis technique so that ultimately software developers will be able to choose the most appropriate technique for the analysis problem at hand.

Acknowledgments: The authors would like to thank Jay Corbett, Lee Osterweil, Michal Young and the anonymous reviewers for interesting observations and useful feedback.

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.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[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.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the 4th European Software Engineering Conference*, pages 283–300, Germany, 1993. published in *Lecture Notes in Computer Science 717*, Springer-Verlag.

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposion on Principles and Practice of Parallel Programming*, pages 21–30, March 1990.

[Cor92] J.C. Corbett. Verifying general safety and liveness properties with integer programming. In *Computer Aided Verification, 4th International Workshop*, pages 357–369, Canada, July 1992. published in *Lecture Notes in Computer Science 663*, Springer-Verlag.

[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.

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

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

[GW91] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third 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. North-Holland, 1977.

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

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

[Mer92] N. Mercouroff. An algorithm for analyzing communicating processes. In *Proceedings of Mathematical Foundation of Programming Semantics '91*, Pittsburgh, PA, March 1992. published in *Lecture Notes in Computer Science 598*, Springer-Verlag.

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

[MR91] S.P. Masticola and B.G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 91–102, May 1991. published in ACM *SIGPLAN Notices*, 26(12).

[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*, pages 129–138, May 1993. published in ACM *SIGPLAN Notices*, 28(7).

[MSS89] T. Murata, B. Schenker, and S.M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314–326, March 1989.

[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 curent trends. In *Current Trends in Concurrency*, volume 224, pages 510–584, 1985. published in *Lecture Notes in Computer Science 224*, Springer-Verlag.

[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.

[Tay83] R.N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Computer Aided Verification '90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–41, Providence, RI, 1990. American Mathematical Society.

[YTL+92] M. Young, R.N. Taylor, D.L. Levine, K. Forester, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. SERC Technical Report TR-128-P, Purdue University, October 1992.

[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. ACM Press.

A Code for examples

The code for the version of the read/write control task that supports 2 active readers and where Done is a global variable is as follows:

```
task body Read_Write_Control is
 ActiveReaders : Natural range 0..2;
 WriterPresent : BOOLEAN := FALSE; -- EVENT "WP := F"
 ErrorFlag : BOOLEAN := FALSE;
begin
 accept Start_Write; -- EVENT "bW"
 accept Stop_Write; -- EVENT "eW"

 loop
  select when not WriterPresent =>
   accept Start_Read; -- EVENT "bR"
   -- EVENT "not WP?"
   ActiveReaders := ActiveReaders+1;

  or accept Stop_Read; -- EVENT "eR"
   if WriterPresent and ActiveReaders > 0 then
    ErrorFlag := TRUE;
   end if;
   ActiveReaders := ActiveReaders-1;

  or when ActiveReaders = 0 and not WriterPresent =>
   accept Start_Write; -- EVENT "bW"
   -- EVENT "not WP?"
   WriterPresent := TRUE; -- EVENT "WP := T"

  or accept Stop_Write; -- EVENT "eW"
   if WriterPresent and ActiveReaders > 0 then
    ErrorFlag := True;
   end if;
   WriterPresent := FALSE; -- EVENT "WP := F"

  end select;
```

```
  if Done and (not WriterPresent) and
     (ActiveReaders = 0) then
   -- EVENT "notWP?"
   exit;
  end if;

 end loop;
end Read_Write_Control;
```

The code for the 2 client version of the protocol program where Done is a global variable is as follows:

```
task body LockManager is
begin
 loop
  exit when Done;
  select
   accept Acquire1;
   accept Release1;
  or
   accept Acquire2;
   accept Release2;
  end select;
 end loop;
end LockManager;

task body Channel is
begin
 loop
  exit when Done;
  select
   accept Header(h :  in INTEGER);
  or
   accept Packet(p :  in INTEGER);
  end select;
 end loop;
end Channel;

task body Client1 is
 h, p :  INTEGER := 0;
begin
 loop
  exit when Done;
  LockManager.Acquire1;
  Channel.Header(h); -- EVENT "h1"
  Channel.Packet(p); -- EVENT "p1"
  LockManager.Release1;
 end loop;
end Client1;

task body Client2 is
 h, p :  INTEGER := 0;
begin
 loop
  exit when Done;
  LockManager.Acquire2;
  Channel.Header(h); -- EVENT "h2"
  Channel.Packet(p); -- EVENT "p2"
  LockManager.Release2;
 end loop;
end Client2;
```