DATA FLOW ANALYSIS FOR VERIFYING CORRECTNESS PROPERTIES OF CONCURRENT PROGRAMS

A Dissertation Presented

by

MATTHEW B. DWYER

Submitted to the Graduate School of the University of Massachusetts Amherst in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1995

Department of Computer Science
DATA FLOW ANALYSIS FOR VERIFYING CORRECTNESS PROPERTIES OF CONCURRENT PROGRAMS

A Dissertation Presented
by
MATTHEW B. DWYER

Approved as to style and content by:

Lori A. Clarke, Chair

Leon J. Osterweil, Member

George S. Avrunin, Member

Michal Young, Member

W. Richards Adrion, Member

David Stemple, Department Chair
Department of Computer Science
To

My in-laws for sharing and sacrificing
so that I could pursue this dissertation

My sisters, brothers, and parents for shaping a personality
persistent enough to finish this dissertation

Sue, Sean and Kate for keeping me happy and loved through it all

I love you
ACKNOWLEDGEMENTS

Thanks

To my advisor Lori Clarke for her personal, financial and professional support; she has the courage of her convictions and has taught me many important lessons about being true to oneself.

To Lee Osterweil for many interesting and fruitful discussions, and helping me see the forest and beyond.

To George Avrunin and Jay Corbett for giving me the chance to watch well planned and well executed research carried out to very high standards.

To Michal Young for providing important validation of my ideas, and for being a model of the academic career that I aspire to.

To Rick Adrion who was there at the beginning of my graduate education and was kind enough to come back for the end.

To the many people who have contributed to the development of Arcadia software, in particular Peri Tarr, Sandy Wise, Tim Chamillard, and Gleb Naumovich who contributed directly to the software I built upon.
ABSTRACT

DATA FLOW ANALYSIS FOR VERIFYING CORRECTNESS PROPERTIES OF CONCURRENT PROGRAMS

SEPTEMBER 1995

MATTHEW B. DWYER

B.Sc., UNIVERSITY OF ROCHESTER
M.Sc., UNIVERSITY OF MASSACHUSETTS - BOSTON
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lori A. Clarke

Developers of modern software systems are increasingly employing concurrency to meet demanding system requirements. To deal with the inherent complexity that results from concurrency, developers require cost-effective automated analysis techniques to gain confidence in the quality of their concurrent software.

We present an approach, called FLAVERS, that is able to provide cost-effective analysis of concurrent programs with respect to a rich class of explicitly stated correctness properties. FLAVERS is based on a family of polynomial-time, conservative data flow analysis algorithms. Unlike existing analysis approaches for concurrent software, FLAVERS allows developers to control the tradeoff between analysis cost and the precision of analysis results by incrementally introducing additional information into the analysis. One strength of this approach is the flexibility allowed in choosing and combining a variety of sources of information about the program and property being analyzed, so as to increase precision without making the cost of analysis impractical.
We have extended the theoretical foundations of data flow analysis to support the description and solution of data flow analysis problems for concurrent programs. We have engineered a domain-specific software architecture that greatly reduces the cost of developing data flow analyzers. These advances are widely applicable. We have leveraged off of them to produce an implementation of FLAVERS with which we have been able to successfully validate the feasibility of the analysis approach.
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Artifact Sizes for header-packet Analyses</td>
<td>155</td>
</tr>
<tr>
<td>5.2</td>
<td>Analysis Cost for header-packet Analyses</td>
<td>156</td>
</tr>
<tr>
<td>5.3</td>
<td>Artifact Sizes for exclusive-read-write Analyses</td>
<td>165</td>
</tr>
<tr>
<td>5.4</td>
<td>Analysis Cost for exclusive-read-write Analyses</td>
<td>166</td>
</tr>
<tr>
<td>5.5</td>
<td>Artifact Sizes for no-read-upon-write Analyses</td>
<td>168</td>
</tr>
<tr>
<td>5.6</td>
<td>Analysis Cost for no-read-upon-write Analyses</td>
<td>169</td>
</tr>
<tr>
<td>5.7</td>
<td>Artifact Sizes for write-first Analyses</td>
<td>170</td>
</tr>
<tr>
<td>5.8</td>
<td>Analysis Cost for write-first Analyses</td>
<td>171</td>
</tr>
<tr>
<td>5.9</td>
<td>Artifact Sizes for neighbors-think Analyses</td>
<td>174</td>
</tr>
<tr>
<td>5.10</td>
<td>Analysis Cost for neighbors-think Analyses</td>
<td>174</td>
</tr>
<tr>
<td>5.11</td>
<td>Artifact Sizes for one-per-pump Analyses</td>
<td>178</td>
</tr>
<tr>
<td>5.12</td>
<td>Analysis Cost for one-per-pump Analyses</td>
<td>179</td>
</tr>
<tr>
<td>5.13</td>
<td>Raw Data for no-premature-go</td>
<td>182</td>
</tr>
<tr>
<td>B.1</td>
<td>Raw Data for header-packet</td>
<td>242</td>
</tr>
<tr>
<td>B.2</td>
<td>Raw Data for Multiple header-packet</td>
<td>244</td>
</tr>
<tr>
<td>B.3</td>
<td>Raw Data for Composite header-packet</td>
<td>246</td>
</tr>
<tr>
<td>B.4</td>
<td>Raw Data for no-orphans</td>
<td>247</td>
</tr>
<tr>
<td>B.5</td>
<td>Raw Data for exclusive-read-write</td>
<td>250</td>
</tr>
<tr>
<td>B.6</td>
<td>Raw Data for write-first</td>
<td>252</td>
</tr>
<tr>
<td>B.7</td>
<td>Raw Data for no-read-upon-write</td>
<td>253</td>
</tr>
<tr>
<td>B.8</td>
<td>Raw Data for Single neighbors-think</td>
<td>255</td>
</tr>
</tbody>
</table>
B.9 Raw Data for Multiple neighboring-think .......................... 257
B.10 Raw Data for Alpha Refined one-per-pump ...................... 260
B.11 Raw Data for Alpha+CI Refined one-per-pump .............. 261
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Approximate Analyses</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Typical Sequential and Concurrent Flow Graphs</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Components of FLAVERS</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Ada Tasks for Semaphore Example</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Trace Flow Graph for Semaphore Example</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Semaphore TFG with Node 19's MIP Edges</td>
<td>79</td>
</tr>
<tr>
<td>4.5</td>
<td>Phase One of TFG Construction</td>
<td>80</td>
</tr>
<tr>
<td>4.6</td>
<td>Phase Two of TFG Construction</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Phase Three of TFG Construction</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>QRE Syntax</td>
<td>90</td>
</tr>
<tr>
<td>4.9</td>
<td>QRE Semantics</td>
<td>91</td>
</tr>
<tr>
<td>4.10</td>
<td>Property Automaton for Response Property</td>
<td>93</td>
</tr>
<tr>
<td>4.11</td>
<td>P-equivalent Paths</td>
<td>97</td>
</tr>
<tr>
<td>4.12</td>
<td>State Propagation at Synchronization Points</td>
<td>100</td>
</tr>
<tr>
<td>4.13</td>
<td>Splitting TFG Nodes</td>
<td>102</td>
</tr>
<tr>
<td>4.14</td>
<td>MIP Paths</td>
<td>111</td>
</tr>
<tr>
<td>4.15</td>
<td>Communication Interval Example</td>
<td>119</td>
</tr>
<tr>
<td>4.16</td>
<td>Multi-caller Interval Example</td>
<td>127</td>
</tr>
<tr>
<td>4.17</td>
<td>Disjoint Interval Example</td>
<td>129</td>
</tr>
<tr>
<td>4.18</td>
<td>Example TFG and TA</td>
<td>133</td>
</tr>
<tr>
<td>4.19</td>
<td>VA for Boolean Variable</td>
<td>137</td>
</tr>
</tbody>
</table>
4.20 TFG Fragment with VA Symbols ........................................ 138
5.1 Analysis Cost Versus CPA Size .......................................... 157
5.2 Total Analysis Time for header-packet .................................. 158
5.3 Total Analysis Time for Multiple header-packet ....................... 159
5.4 Total Analysis Time for Composite header-packet .................. 160
5.5 Total Analysis Time for Composite and Multiple header-packet .... 161
5.6 Total Analysis Time for no-orphan-packets ............................. 162
5.7 Total Analysis Time for exclusive-read-write .......................... 167
5.8 Total Analysis Time for no-read-upon-write ........................... 170
5.9 Total Analysis Time for write-first ...................................... 172
5.10 Total Analysis Time for neighbors-think .............................. 175
5.11 Total Analysis Time for Multiple neighbors-think ................... 176
5.12 Total Analysis Time for Alpha one-per-pump ......................... 180
5.13 Total Analysis Time for Alpha+CI one-per-pump .................... 181
6.1 Architecture of Data Flow Analyzer ..................................... 199
6.2 A Library of Components .................................................. 203
6.3 Boolean Variable Lattice ................................................... 204
6.4 Example Function Space ................................................... 214
6.5 Example Sparse Representation .......................................... 215
6.6 Example Analyzer Generation ............................................ 216
A.1 FLAVERS/Ada Type Hierarchy ........................................... 231
A.2 FLAVERS/Ada Tool Orderings ............................................ 234
B.1 Refined TFG Edges for Protocol ......................................... 242
B.2 Individual Cost Components for Single header-packet ............... 243
B.3 State Propagation for Single header-packet ............................ 243
B.4 Individual Cost Components for Multiple header-packet .......... 245
B.5 State Propagation for Multiple header-packet ........................ 245
CHAPTER 1
INTRODUCTION

Users of computers ask two practical, and fundamental, questions about every piece of software: "is it correct?" and "is it fast?". Different people may ask these questions in different orders, but, ultimately they are both important. Very fast software is useless if it computes an incorrect result. Correct software is useless if it cannot produce its results in a reasonable amount of time. The quality, or degree of correctness, of software is becoming increasingly important as the criticality of applications rises to new levels. Software is being used, increasingly, to control safety critical systems in diverse application domains ranging from medicine to avionics. The problem of determining software quality is exacerbated by its increasing size and complexity; multi-million line programs are not uncommon. Increasing the performance of software has led to the movement of concurrent programming technology from special purpose database and operating systems into the programming mainstream. Enormous communicating distributed software systems are in use every day. While concurrency may be the road to high-performance it is a road littered with obstacles to high-quality. Concurrent applications have a new, and fundamental, source of complexity that derives from the ability of components of the software to execute at the same time. To overcome this complexity, software developers require cost-effective techniques to gain confidence in the quality of their concurrent applications.

The demand for high-quality production software has fueled a wealth of research on a variety of techniques for gaining high-confidence in software [62]. This research covers a spectrum from manual techniques, such as proof construction [31, 50] and
fault analysis [48], static techniques, such as anomaly detection [39] and verification approaches [4, 11], and dynamic techniques, such as testing [1] and run-time monitoring [35, 66]. We believe that no single approach will meet all the needs of software developers for practical assurance of software quality. Together, however, these techniques can provide a wide variety of different kinds of information about program behavior. These techniques can be mutually supportive in a variety of ways. For example, results produced by automatic verification techniques can be used in manual proof techniques, or information from fault analysis or anomaly detection can be used to focus efforts in testing. As approaches to assuring software quality mature, more effort will need to be exerted in exploring the cross-fertilization of individual approaches. This unification will be possible only after a thorough understanding of the practical utility of individual techniques is gained.

This thesis makes contributions to the specification, development and evaluation of data flow analyses for concurrent programs. The four main contributions of this dissertation are in:

- generalizing the theory of data flow analysis
- developing a novel flexible program analysis technique
- empirically evaluating that technique
- engineering analyzers to solve a data flow analysis problem

**Generalizing the Theory of Data Flow Analysis** Conceptually, data flow analysis is a process for uncovering facts about executable program behavior without actually running the program. With applications to compiler optimization, program testing, validation, and verification, data flow analyses are an important component of a variety of software development activities. The constant desire for faster programs has spurred tremendous theoretical and practical developments in program analysis
and optimization techniques. Unfortunately, the bulk of this work has been for sequential programs. We generalize existing results on monotone data flow analysis frameworks to what we call complete-lattice monotone data flow analysis frameworks. Complete-lattice frameworks can be used to describe data flow analysis problems for concurrent programs; the solutions for such problems can be significantly more precise than using existing techniques. We have adapted existing solution algorithms to data flow analysis problems formulated as complete-lattice frameworks.

Development of a Flexible Program Analysis Technique FLAVERS (FLow Analysis for VERifying Specifications) is a static program analysis approach that has the potential to provide cost-effective analysis of concurrent programs with respect to explicitly stated correctness properties. With FLAVERS, developers define a set of program events that they want to reason about and specify patterns of those program events. They choose whether the analysis should attempt to verify that all or no program executions satisfy the given pattern of events.

FLAVERS is fundamentally an approximate analysis technique. It is pessimistic in the sense that it will only report that a program satisfies the specified pattern of events if that is actually the case. To overcome the fundamental imprecision of static analysis, we have developed a range of techniques for improving the precision of analysis results. Prior to analyzing executable program behavior with respect to the specified pattern of events, we can perform preliminary analyses to attempt to improve the precision of the program model on which analyses are based. To increase the precision of analysis results we can constrain the analysis to avoid considering portions of the model that are not truly executable. Once analysis is complete we can combine the results of independent, but related, analyses to improve the overall precision of the combined analysis results.
One of the strengths of FLAVERS is its flexibility. A rich class of properties specified as patterns of program events can be analyzed. FLAVERS provides the ability to control the tradeoff between the precision of analysis results and the execution time of the analysis. This has two significant implications. By choosing to encode more information into a FLAVERS analysis higher levels of precision can be obtained. The sources of information that can be encoded into a FLAVERS analysis cover a wide range and by wisely selecting from these sources we can reduce analysis cost and increase precision.

**Empirical Evaluation of FLAVERS**  In contrast to most existing static concurrency analysis techniques, FLAVERS is based on algorithms with low-order polynomial bounds on their running time. While this suggests that FLAVERS may be relatively more cost-effective than other techniques, an understanding of the actual cost of FLAVERS analyses as applied to real programs can only be gained empirically. We have constructed a toolset that enables the analysis of Ada programs with respect to specified patterns of program events. This toolset has been applied to the analysis of a collection of concurrent programs. The results indicate that FLAVERS can provide analysis results of sufficient precision to verify interesting behaviors of concurrent programs, and that the cost of such an analysis grows as a low-order polynomial in the size of the program.

**Engineering of Data Flow Analyzers**  Data flow frameworks help us specify a data flow analysis problem and to reason about how much it would cost to solve such a problem in the abstract. They do not help with the construction of an implementation of a solver for a given analysis problem. We have developed a domain-specific architecture that facilitates rapid prototyping of data flow analyzers. The architecture defines a set of interoperating standard interfaces for components of data flow analyzers. With this architecture, a developer chooses from a collection of pre-existing
components or, using high-level component generators, constructs new components and combines them to produce a data flow analyzer. Component generators capture common functionality and facilitate creation of components specialized for the problem at hand. Finally, the interfaces defined by the architecture allow existing and generated components to be reused across data flow analyzers. Thus, the cost of building, or generating, a component can be amortized over a number of analyzers. We envision that this architecture will be especially useful in the early stages of development of data flow analyzers. Prototype analyzers can be developed quickly for a variety of formulations of a given data flow analysis problem and evaluated empirically to determine the best formulation in practice.

The remainder of the thesis is organized as follows. Chapter 2 describes related research; we focus on specifying properties of programs and on static program analyses. In Chapter 3 we present complete-lattice data flow analysis frameworks and associated solution algorithms. Chapter 4 describes the details of FLAVERS. We present intuition, examples, complexity arguments and correctness arguments for each part of the analysis. To understand the cost-effectiveness of FLAVERS in practice, we have conducted an empirical study which is reported on in Chapter 5. Chapter 6 outlines the domain-specific architecture for constructing data flow analyzers. Chapter 7 mentions directions for future research. Appendix A provide details of our implementation of FLAVERS for the Ada programming. Appendices B and C are included to provide details of the raw empirical data that is interpreted in Chapter 5 and the source code of the programs in the empirical evaluation.
CHAPTER 2
RELATED WORK

In this Chapter, we provide a broad overview of program specification and analysis techniques and then focus more closely on static program analysis techniques for concurrent programs. Chapters 3, 4, and 6 each have detailed background sections that cover material not included here. We begin by defining terminology that will be used throughout the remainder of the thesis.

2.1 Terminology

In this section, we describe a model of execution for concurrent programs. The model is intended to be general enough to represent a wide range of real concurrent programs. We discuss some fundamental limits on the information that can be captured in such a representation, and how we can construct tractable program models in the face of those limitations.

2.1.1 Models of Program Execution

A concurrent program consists of a number of independently executing threads of control called tasks. Tasks may execute the same or different streams of instructions. They may operate in complete isolation or they may communicate, through various mechanisms, with other tasks.

Task communication mechanisms fall into two major categories: synchronous and asynchronous. Synchronous communication requires communication partners to execute communication statements at the same time. Two popular synchronous mechanisms are remote procedure call (RPC) and rendezvous. RPC is a procedure call
in that the call site is in one task and the body of the procedure is executed in another task. Rendezvous refers to a pair of tasks that synchronize, potentially pass data, and then continue execution. From the perspective of each task the communication statement is synchronous as the task blocks until the partner reaches the rendezvous point. The main difference between RPC and rendezvous is from the perspective of the task receiving communication. The receiver of an RPC is typically a server whose sole function is to process the request and return control to the calling task. In rendezvous style communication the situation is more symmetric where both sender and receiver are independent entities that mix local computation with inter-task communication. Asynchronous communication allows communication partners to independently execute communication statements as long as sends precede receives.

Two popular asynchronous mechanisms are message passing and shared memory. In message passing, tasks may independently send and receive data. Sender and receiver do not synchronize as a result of the data transfer. Shared memory refers to data that is visible to multiple tasks. Communication is effected by defining and referencing shared variables. Shared memory can be viewed as a form of message passing where sends are writes to shared memory and receives are reads from shared memory.

Communication can also take place between collections of tasks. In multicast communication there is a single sender and multiple receivers. The tasks involved in a communication event can be determined statically, where the identity of communication partners is known prior to program execution, or dynamically, where the identity of communication partners need only be computed prior to execution of the communication in which they participate.

In general, there need be no fixed bound on the number of tasks or variables in a program. To reason statically about the executable behavior of program, however, we require that the number of tasks, variables, and the sizes of domains of variables are fixed at the time we construct a program model. This means that we model
particular instances or configurations of a program. For example, we will model a $n$ producer $m$ consumer concurrent program for any given values of $n$ and $m$, but not the original parameterized program itself.

An execution state of a task consists of two components: control state and data state. The control state of a task is its current execution location, which might be represented by a source code line number, for example. The data state of a task consists of the values of all local variables of a task. If a program has global variables then those can be accounted for with an additional data state component. We can talk about a global state of the concurrent program as a whole; such a state consists of an execution state for each task in the program. The set of all global program states that can be entered during some execution of a program is called the program state space. State transitions from one global state to another correspond to a task performing local computation, asynchronous communication, or a pair of tasks performing synchronous communication. Transitions typically involve a single task moving to a new control state and possibly a change in the data state. When a pair of tasks communicate synchronously a transition causes both the sending and receiving tasks to change their control and, potentially, data states.

In this model, we describe a program execution as a sequence of global states, or, equivalently, as a sequence of transitions. This is an interleaving model of execution; two transitions, $a$ and $b$, that may execute concurrently are modeled as the pair of sequences $ab$ and $ba$.

2.1.2 Fundamental Limits on Models and Analyses

Fundamental undecidability results for sequential programs, such as the halting problem, also apply to concurrent programs. This is due to the fact that concurrent programs can be modeled as a non-deterministic choice between any of the possible interleaved executions of the tasks in the concurrent program.
As with sequential programs, in general, the state space of a concurrent program may be infinite\(^1\), making exhaustive exploration impossible. Restricting analysis to finite models helps some, but the determination of many important properties remains intractable\cite{74}. For example, analyzing finite-state programs with synchronous communication to determine the statements that may execute in parallel is NP-hard\cite{74}. These results arise from the fact that the state space of a concurrent program is, in general, exponential in the number of tasks.

### 2.1.3 Tractable Models

We would prefer models that represent all and only the executable behavior of concurrent programs and are tractable to construct and manipulate. In general, this is not possible. To address this, we construct finite state models that necessarily approximate the executable behavior of a program. There are many ways to form this approximation; in the terminology of Young & Taylor \cite{85} a model can sample the behavior of the real program or fold the behavior of the real program. A sampled model includes no unexecutable program behavior, but may fail to include some executable program behavior; such models are useful for fault detection as they can be used to demonstrate the presence of executable program behaviors that fail to satisfy a specification of desired behavior. A folded model may include some unexecutable program behavior, but it includes all executable program behavior; such models are useful for verification as they can be used to demonstrate that all executable program behaviors satisfy a specification of desired behavior. Either of these approaches leads to imprecision in the model. We refer to an analysis as conservative or pessimistic if it reasons about all the behavior in some folded model. We refer to an analysis as

---

\(^1\)We consider a program's execution with respect to the semantics of its programming formalism and ignore the fact that once a program is mapped to a physical execution environment it becomes finite state.
optimistic if it reasons about some or all of the behavior in some sampled model ². We also use the term pessimistic (optimistic) to refer to a folded (sampled) model. So pessimistic (optimistic) analyses require pessimistic (optimistic) models. We refer to a conclusive analysis result for a pessimistic (optimistic) analysis as one that verifies a specification of desired behavior (detects a fault). Failure to verify a specification of desired behavior (detect a fault) in a pessimistic (optimistic) analysis, however, does not mean that program fails to satisfy the specification (program is fault free); we call these results inconclusive.

Given that program executions can be viewed as sequences of state transitions, or atomic program events, we can talk about conditions for such a sequence to be executable or unexecutable. A sufficient condition for a sequence of program events to correspond to a program execution is true, only if the sequence corresponds to a program execution. A necessary condition for a sequence of program events to correspond to a program execution is false, only if the sequence does not correspond to a program execution. Sufficient and necessary conditions are related to optimistic and pessimistic models in the following way: an optimistic model encodes only sequences that satisfy sufficient conditions and a pessimistic model encodes only sequences that satisfy necessary conditions. Intuitively, we can improve the precision of an optimistic model by conjoining sufficient conditions, to produce a new condition that may be satisfiable by additional sequences. We can improve the precision of a pessimistic model by disjoining necessary conditions, to produce a new condition that is unsatisfiable by some sequences that were previously in the model. Figure 2.1 depicts three possible ways we might construct models of executable sequences. We consider each executable sequence as a unique value. In the figure boundary 1 encloses the set of executable sequences. It is contained within boundary 2 which encloses a superset

²A folded (sampled) model is necessary for constructing a pessimistic (optimistic) analysis, but not sufficient. For example, if an analysis over a folded model ignores some of the behavior captured in the model, then we have no guarantee about analysis results; any result from such an analysis would be inconclusive.
of the executable sequences and is therefore a pessimistic model; region 1 represents the imprecision that arises from this pessimism. Boundary 3 encloses a subset of the executable sequences and is therefore an optimistic model; region 2 represents the imprecision that arises from this optimism. Boundary 4 encloses a region that corresponds to neither an optimistic nor pessimistic model; reasoning using such a model would be incapable of producing a conclusive analysis result.

In practice, the degree of imprecision introduced by folding or sampling can vary widely from program to program and is highly dependent on the kind of reasoning we want to perform. For many concurrent programs there are states that may be treated as equivalent when analyzing a particular correctness property. For example, Wolper [83] has identified a class of programs that he calls data independent; these programs may manipulate data from arbitrary domains, but those values are not used in program flow decisions, i.e., the data is just baggage being moved from point to point. In this case, for reasoning about properties of the program that are independent of the baggage data, we can, without loss of precision, reduce the number of data states considerably by folding equivalent baggage values.
2.2 Specifying Program Behavior

What kind of behavior are we interested in reasoning about? The problem of verifying a complete and accurate specification of total program correctness is, in general, undecidable due to the halting problem. Therefore, we focus on describing classes of properties that have exhibited themselves in existing concurrent programs. Historically operating systems and the applications executing under their control represent some of the first concurrent programs. These systems provide examples of a number of anomalies, i.e., undesirable properties, specific to concurrent programs. We refer to a property of a program as a specification of executable program behavior along with an indication as to whether the program should exhibit that behavior on all, some or no program execution.

Techniques for insuring that an application is free from deadlock have long been studied. Operating systems as the granters and controllers of system resources are mostly concerned with resource deadlock. In its simplest form this condition arises when a task, $T_1$, acquires exclusive access to a resource, $R_1$, and blocks waiting to acquire another resource, $R_2$, while another task, $T_2$, having already acquired exclusive access to resource $R_2$ blocks waiting to acquire $R_1$. Informally, they both have what the other wants and neither is willing to give up what they have, thus both tasks are permanently blocked. This condition generalizes to $n$ tasks. A deadlock that involves all tasks in a program is said to be total, otherwise it is partial. Concurrent programs that involve explicit inter-task communication can suffer from a condition called communication deadlock. This is analogous to resource deadlock where the resource in question is a communication interaction with another task.

Even in the absence of deadlocks it is possible for a task to be unable to make progress. A task is said to starve if it continually requests a resource or communication from another unblocked task, but never receives it. Operating system scheduling algorithms can inadvertently starve a task of low priority by choosing to give all
resources to higher priority tasks. Starvation is an instance of a *fairness* property. Informally, the notion of fairness is that if a task continually requests access to a resource eventually it will be granted the resource.

A *race condition* is when two, or more, tasks are executing asynchronously to reach a communication, synchronization, or data sharing point. Races are good as they are one source of parallel speedup. It is only when the result of a race, i.e., which task reaches the point first, can affect the outcome of the program’s computation adversely that they are considered harmful and called *critical races*. *Data races* occur when tasks are involved in a race condition between definitions and uses of shared data. Which ever task reaches the point of data sharing first is said to *win* the race. When tasks all read the value of a shared variable there is no data race as the winner and loser(s) of the race will get the same value. When a task defines a shared variable the values seen by the tasks that use that variable are dependent on whether the defining task wins or loses the race. Operating system device handlers are often involved in race conditions as they transfer data to/from device registers from/to system buffers that are used by other operating system tasks.

For consistent management of internal operating system data structures, it is often the case that a sequence of operations needs to be performed indivisibly. Without special precaution it is possible that the activity of other tasks in the program may interfere with this. This is referred to as the *mutual exclusion* problem.

As will be surveyed in the next section, freedom from deadlock and has been the subject of much of the research on analysis of concurrent programs. This is due to the fact that this problem is universal. We can think of all concurrent programs as having an implicit specification that they should be free of deadlocks. Researchers have concentrated on analysis techniques for these important properties and have more recently generalized to a wider class of properties.
Freedom from deadlock is an example of a \textit{reachability} property. Reachability properties assert that in some program execution it is possible to enter a global state that satisfies some criteria of interest. We refer to the program states that can be entered during some execution of the program as \textit{reachable}. One class of reachability properties are the so called \textit{safety} properties. These properties describe an undesirable program state and assert that such a state is not reachable on any program execution. To refute a safety property we need only find a single program execution in which the undesirable state is reachable. For example, if we view the existence of a program deadlock as undesirable and we can characterize program states that exhibit that property, then specifying that all program executions are free of deadlock is a safety property. Safety properties are often contrasted to \textit{liveness} properties. Liveness properties describe a desirable program state and assert that program executions will eventually and repeatedly reach such a state. Conceptually, these properties specify the existence of cyclic patterns of program behavior that include the desirable program states. Refutation of a liveness property requires that we exhibit a program execution that breaks such a cycle at an undesirable point. For example, if we view the granting of a resource to a requestor as desirable, then freedom from starvation is a liveness property. This property asserts the existence of a cycle in which requests for a resource are eventually followed by the granting of that resource. To refute this property we need to show that the cycle can be broken at a point where there is an outstanding request.

A number of formalisms have been suggested for specifying these kinds of properties of concurrent programs. A variety of temporal logics have been proposed, the most popular being LTL, a linear time propositional logic, and CTL, a branching time propositional logic [12]. For the most part logics adopt a \textit{state} based approach to describing program behavior. A formula describes properties of states that should hold or not at various times during any or all program executions. Most logics that
serve as a basis for automated reasoning are propositional, while predicate logics are commonly used in manual proof construction. The dual of the state based approach considers events, i.e., program actions that cause state changes. TSL[36], ω-regular expressions [47] and Cecil [60] are examples of formalisms for describing sequences of events that should or should not occur in any or all program executions. Algebras of program events [5, 55] have also been used to describe both concurrent programs and properties to be verified of those programs. An event is typically defined as an indivisible program action. The event/state perspective is unimportant in theory, but, in practice, some properties may be more easily expressed in one setting than the other.

2.3 Reasoning About Program Behavior

There has been a wealth of research into techniques for assessing the correctness of software. Analysis techniques including code reviews, testing, dynamic assertion checkers, formal proofs and static analysis techniques can all be viewed as providing information that increases confidence in the correctness of a software system.

Dynamic analyses involve executing a program and reasoning about the behaviors exhibited by those executions. The number of possible program executions is typically prohibitively large; as a consequence, dynamic analyses are limited to approximating the possible behaviors of the program. Dynamic analyses include traditional testing approaches [1], assertion checkers [35, 66], and special purpose property checkers [58, 77]; these approaches are optimistic. They are useful for detecting program faults, but not for providing assurance that a program is free of faults.

Proof systems are well defined axiomatic systems with associated rules of inference [20, 26, 31, 37, 63]. Typically, a proof that a program satisfies a given property is constructed by hand. The process of constructing such a proof requires precision and creativity so that the resulting proof is clear enough to be independently checked for
accuracy. This process is limited only by the truth of the proposition being proven and the skill and effort of the analyst. Studies of the application of formal methods, and their associated proof techniques, suggests that even with automated assistance proof construction for realistic programs requires enormous effort [18, 67].

Efforts to automatically construct proofs are limited in generality by the halting problem. Mechanical proof assistants for limited domains have been developed [44]; such an approach is pessimistic. When presented with a program that satisfies a specified property a proof assistant may fail to indicate its correctness.

An alternative class of techniques for reasoning about program behavior are static analyses. This thesis is about static analysis and as a result we spend the rest of this section discussing a variety of research on static analysis of sequential and concurrent programs.

The term static analysis refers to a wide range of techniques for extracting information about the possible executions of programs from their text, or other descriptions. Many forms of static analysis are able to provide information that can be used to verify properties of a given program. Almost all software undergoes some form of static analysis. Most modern compilers perform syntactic and semantic analysis in order to enforce the syntactic and semantic rules of a programming language. Many go further, warning the user of possible program anomalies. People perform static analysis in the form of code walk-throughs and design reviews. This section presents research on the automated analysis of correctness properties of concurrent programs. We begin with a discussion of a number of common representations and formalisms used to model program behavior.

2.3.1 Program Representations

Reachability graphs represent the possible executable paths, executable events and global states that a program may enter. The graph is rooted at a unique program start
state, edges correspond to state transitions, and nodes are global program states. The information contained in the reachability graph depends on the information content of a program state. It is common in the literature to use reachability graphs that fold all data states and represent only control state information. The size of a reachability graph is, in general, exponential in the number of tasks in a concurrent program, even if data information is folded.

To overcome the exponential nature of reachability graphs, ordered binary decision diagrams (OBDD) have been suggested as a symbolic representation of the state transition relation of a program. OBDDs were developed as a graph representation of boolean functions [6]. The idea is similar to decision trees, but where each branch tests a different input variable and the leaves are function result values. OBDDs can represent many functions compactly by taking advantage of don't care conditions on function input values. This can reduce the depth and branching of the OBDD along certain paths. OBDDs require input variables to be ordered and this order affects the size of the minimal representation. There is no loss of information when using OBDDs in place of enumerating the edges of the reachability graph, but there is no guarantee that it will result in a smaller representation.

It is possible to represent programs as a finite state automata (FSA). As with reachability graphs, when we attempt to construct a FSA that captures the detailed semantics of a concurrent program it grows exponentially in size. For concurrent programs we can represent each individual task as a FSA and the program as a whole as the product of the task automata.

Closely related to finite state automata are flow graph representations. One standard flow graph representation is the control flow graph (CFG). A CFG represents the possible control flow paths through a sub-program. Nodes correspond to program statements and edges correspond to control flow successor relationships. A node can have multiple outgoing edges to represent the fact that the node determines which
control flow path to follow. A CFG typically fold data state information and represent only program control state information.

In either FSAs or CFGs additional information can be incorporated in the representation. For example, the values of program variables. Modeling program variables can improve the precision of the program model and is crucial to successful analysis of many systems.

A number of flow graph representations have been proposed for concurrent programs with the main distinctions being the granularity of states or events and how they represent inter-task communication. The program can be modeled as a disconnected set of task flow graphs, where communication statements are modeled by matching edge labels in the sending and receiving tasks. We call this an implicit representation of communication. When explicit edges are introduced between task flow graphs to represent possible communication, we call this an explicit representation of communication.

Petri Nets (PN) [56] are a widely studied representation that is well suited for modeling concurrent systems. Petri nets are bipartite graphs with node kinds called places and transitions. For the purpose of representing concurrent programs, places represent components of state information and transitions represent system events [25, 69]. Petri nets have a well defined execution semantics. The state of the net is represented by a marking, i.e., an indication of the number of tokens contained in each place. A transition fires when all of its input places have tokens and produces tokens for its output places. We can construct Petri nets whose execution behaviors represent the execution behaviors of a concurrent program.

2.3.2 Reachability Analysis

Reachability analysis involves the enumeration and search of a program's state space for a state that satisfies a specified property; it is a pessimistic technique. Taylor
[75] presents an algorithm for performing such an analysis to determine the presence of deadlock in concurrent programs represented as a collection of CFGs. Young et. al. [86] propose an environment, CATS, for the analysis of concurrent programs that supports a range of program representations and analysis techniques. One analysis component of CATS is a reachability based temporal logic assertion checker. CATS tools construct reachability graphs from a collection of TIGs [49] that represents the program. TIGs yield a compact conservative representation of task communication and synchronization events by abstracting and summarizing all internal sequential task computation. While yielding a significant reduction in the size of reachability graphs over non-abstracted representations, reachability analysis based on TIGs is in the worst-case exponential. Experimental results indicate that the CATS capabilities provide cost-effective analysis for small to medium size programs.

Duri et. al. [23] describe a reachability based deadlock detection system for concurrent Ada programs represented as Petri nets. Prior to construction of the reachability graph Petri nets are subjected to a series of reductions that replace portions of the Petri net with smaller sub-nets that maintain the properties necessary for deadlock detection. This approach is capable of dramatic reductions in the size of the reachability graph for certain problems. This amounts to folding of information that is irrelevant to deadlock detection, thus there is no loss of information for checking deadlock freedom. Perhaps more interesting are the results from this work of combining Petri net reductions with the stubborn and sleep set methods, discussed below. The empirical evidence demonstrates that the techniques are more powerful in combination than in isolation. While these results are encouraging this approach is currently limited to checking freedom from deadlock.

One approach to reducing the state space is to view the events in a program execution as partially rather than totally ordered. The large size of reachability graphs for concurrent programs is due, in part, to the effect of the many interleavings
of independent program events. From any given task's perspective the order in which other tasks perform their computation, whether local or in communication with tasks other than itself, is irrelevant. In this sense the set of executions that correspond to all possible orders of independent events are equivalent and we need only consider a single representative of this set during analysis. Godefroid's sleep sets [28] are a technique for exploiting this structure of state spaces. Sleep sets are used to limit the size of the reachability graph, but preserve enough information to form the basis for efficient verification of deadlock freedom and safety properties of finite state programs. Valmari [79] presents a similar technique that preserves information sufficient to check general linear temporal logic properties. In general, neither of these techniques is capable of reducing the reachability graph to a tractable size, but for selected examples they have produced reachability graphs that are linear in the number of tasks.

2.3.3 Integer Linear Programming

Murata et. al. [57] describe a two level technique for detecting deadlocks. The first level is an optimistic analysis to detect deadlocks that occur on all program executions and, if none are detected, the second level uses a pessimistic analysis detect deadlocks that may occur on some program execution. Both methods require a computation of the set of T-invariants for a Petri net representation of the program. A T-invariant represents the number of times each transition may fire during execution of the Petri net; they are computed by interpreting the adjacency matrix of the Petri net as a system of linear equations and solving that system. Unfortunately, in the worst-case the number of solutions is exponential in the size of the concurrent program. If the set of T-invariants does not cover all Petri net transitions, then there is a potential deadlock on all program executions, however, coverage of all transitions does not imply that the program is deadlock free. Possible deadlocks are found in the Petri net by searching for cycles that satisfy certain necessary conditions for deadlock. If such
a cycle exists, a reachability search, guided by T-invariant information, is made to
determine if the cycle is executable, and if so, a deadlock is reported. Unfortunately,
there has been little experimentation with these techniques so we are unable to judge
the effectiveness of either technique in isolation or in combination.

Integer Necessary Conditions Analysis (INCA) [4] is an automated pessimistic
technique for reasoning about the executions of a concurrent program with respect
to specified patterns of program events. The behavior of each task in the program is
modeled as a finite automaton with a transition alphabet corresponding to program
events. The behavior of the concurrent program as a whole is captured by the task
automata and by constraints, expressions that represent inter-task communication
and synchronization. Conceptually, the goal of the analysis is to see if the language
of this expression corresponds to a legal execution of the program. A string of event
symbols represents an execution of the program, if for each task in the system, the
projection of the string on the alphabet of the task lies in the language of the task
automata, and for each constraint, the projection of the string on the alphabet of
the constraint lies in the language of the constraint. This means that the string repre-
sents legal executions within each task and satisfies the intertask communication
and synchronization constraints. Constrained expressions are capable of expressing
the recursive enumerable languages over event symbols and therefore testing language
containment for constrained expressions is undecidable. The analysis uses a variety
of necessary conditions for a string of event symbols to correspond to a program
execution; obeying the count and order of events within each task and equal num-
bers of send and receive events for each communication channel are examples of such
conditions. The conjunction of these conditions is encoded as a system of linear in-
equalities. The executable behavior to be checked is also specified in the inequalities.
If no solution to this system is found, then the property does not hold. If a solution is
found, a reachability search guided by the solution is conducted to eliminate some of
the potential inconclusive results. The necessary conditions used in the analysis, for example intra-task event ordering and communication count information, appear to be strong as the number of false positive results is small. Empirical evidence demonstrates that for a number of concurrent programs the analysis scales well. Recent results [15] demonstrate that the technique is capable of expressing all properties that can be expressed in linear temporal logic.

2.3.4 Compositional Analyses

Some programs are structured with clusters of tasks that have a high degree of internal interaction but a low degree of external interaction. In fact, if the interface design principles of coupling and cohesion are applied to inter-task communication interfaces in a program such a structure seems quite likely. We can construct the composition of tasks within such clusters and analyze for local properties of interest. In order to analyze the entire program we hide internal interactions of the task cluster by replacing the cluster with a single task that performs the same external interactions in the same order. As long as the composition operation is both associative and commutative we may change the order of composition without affecting the information captured in the program representation. This results in a hierarchy of state spaces that holds the same information as the exact state space, while the sum of the sizes of the component spaces is smaller. Compositional analysis strategies hold the potential for reduction in the size of the state space representation through the simplification of partial task products.

Process algebras are formalisms for describing and reasoning about an abstract representation of concurrent programs. Two well studied algebras are Milner's CCS [55] and Bergstra and Klop's ACP [5]. Process algebras consist of a set of action symbols that represent atomic program events, including inter-task communication events, a set of operators that allow for sequencing, selection and composition of actions and
a set of laws that describe the algebraic properties of the operators. After choosing
a suitable set of actions, the tasks of a concurrent program can be modeled as ex-
pressions built from actions using sequencing and selection operators. The behavior
of the entire program is modeled by connecting these expressions with an explicit
composition operator. Many algebras include a hiding operator that allows specified
action symbols to be replaced by the "silent action", thereby allowing simplification
of expressions involving those actions.

Fundamental to analysis based on process algebras is the notion of equivalence.
Two processes are equivalent if one can replace the other in some context without af-
feting the value of the enclosing process. There have been a number of equivalences
proposed. They range in strength and in complexity of testing for process equi-
ivalence. The strength of an equivalence is an indication how finely it distinguishes
processes. All equivalences can be described in terms of how they partition the set
of syntactically legal processes; a weak equivalence results in a few large classes while
a strong equivalence yields many small classes. In a compositional analysis strategy,
we would like to replace a process, describing a set of program tasks, by a simpler,
equivalent process that represents the externally visible actions of the tasks and hides
the internal actions. To do this, we would like equivalence classes to be as large as
possible to increase the number of choices for replacement processes. On the other
hand the equivalence measure must distinguish between processes that satisfy and
those that fail to satisfy the properties we are interested in reasoning about. In the
extreme, an equivalence measure could judge all processes equivalent, which would
be of little use.

We can specify the intended high-level behavior of our program as a process and
describe the detailed low-level computation of our implementation as a process. Then
the formal notion of process equivalence can be used to check that our implementation
satisfies its specification. Another application of these equivalence measures is to
define an ordering relation between processes, a *preorder*, such that process *A* is less than *B* if *A* has all of the behavior of *B*; note that we allow *A* to have additional behavior. This is useful when a top down refinement strategy is being used for specification and implementation. We can leave parts of the specification out at the highest levels and fill it in as we descend into the details at each step, thereby insuring, by checking preorders, that the more detailed specification satisfies the less detailed one.

One system that supports the specification of programs and properties as algebraic expressions is the Concurrency Workbench [14]. It includes capabilities for testing various equivalences and preorders between processes and verifying properties specified in the propositional mu-calculus.

Yeh and Young [84] make use of fundamental properties of process algebras to control the complexity of reachability analysis. They have built a system, PAL, that allows for specification of concurrent programs in an Ada-like language. These specifications are translated into a transition graph representation that allows for construction of the program’s reachability graph in a compositional fashion, using action hiding and process simplification to reduce the size of intermediate products. In contrast to other process algebra research, this work adopts a programming language perspective that enables them to address one of the major difficulties of compositional approaches, namely choosing the order in which to produce and compose the intermediate reachability graphs. This choice is crucial to cost-effective compositional analysis, as the worst-case leads to exponential complexity. The scoping contours for each program provide likely points at which to decompose the program, since good program design suggests that communication is minimized across such contours. Unfortunately many of the standard programs in the literature exhibit little structure. For these problems, experimental evidence demonstrates that a well chosen analysis order allows PAL’s analysis to scale well with the problem size. The programming
language perspective is desirable since developers need not translate programs to an algebraic formalism to gain its benefits.

2.3.5 Model Checking

If we restrict our interest to concurrent programs that can be modeled as a set of communicating finite state machines we can apply model checking. Clarke et. al. [11] use such a technique for verifying properties specified in a branching time temporal logic, CTL. The goal of model checking is to demonstrate that a logical structure representing the program is a logical model of the CTL formula representing the property of interest. This structure is a finite graph whose nodes represent global system states labeled by the set of logical propositions that are true in the state and whose edges represent the state successor function. A CTL formula expresses truth values of logical propositions associated with the nodes of a path or set of paths through the graph. A CTL formula is translated to a recursive predicate transformer whose maximum fixed point determines the satisfiability of the formula. Early model checking techniques required a structure that is equivalent to the program reachability graph; consequently, model checking was limited by the explosive growth of its inputs.

In practice, a large reduction in the size of the structure is often afforded by using OBDDs to represent the state space symbolically. This has greatly extended the scalability of model checking for a concurrent hardware systems. The breadth of applicability of OBDD based techniques to the analysis of concurrent software remains to be seen.

Recent work by Clarke, Grumberg and Long [13] provides the ability to specify an abstract model of the program that ignores certain details of program behavior, but results in smaller OBDD representations. Abstractions are defined that map a set of global program states to a single state. They are not information preserving and it is intended that an analyst will define appropriate abstractions for the properties
and programs to be verified. A pessimistic representation can be constructed directly from the description of the program and the abstractions without having to construct the unabstracted state space. By going directly to the abstracted state space they avoid the complexity of the full state space, but allow only for a pessimistic analysis. This work has further extended the capacity and applicability of model checking at the expense of user interaction and conservatism in results. It is clear that many programs of interest involve abstractions that need not be viewed as binary functions. Recent work by Hu et. al. [41] provides for a high level description of finite state programs that include structured data, sequential assignment and conditional statements. These programs may then be compiled into OBDDs for verification. Experiments with verifying an industrial cache coherency protocol indicate that the breadth of applicability of model checking techniques is increasing.

2.3.6 Data Flow Analysis

Chapters 3 and 6 give a detailed overview of techniques for specifying and implementing solvers for data flow analysis problems. In this section we focus on application of data flow analysis techniques to fault detection and verification.

Taylor and Osterweil [77] describe data flow computations for verifying a fixed set of properties, including certain forms of deadlock and data races, in concurrent programs. They construct special purpose iterative algorithms based on flow equation formulations of each of the properties to be analyzed. This work represents one of the first applications of data flow analysis to verification of properties of concurrent programs. The programming language considered does not have a rich set of task communication and synchronization capabilities and it is not clear how easily these ideas can adapt to a more rich programming language. Some of the algorithms assume the existence of a predicate that is able to decide whether two statements can be executed in parallel. This problem is now known to be NP-hard [74].
Reif and Smolka [65] present a data flow analysis for concurrent programs that use asynchronous communication. They consider programs with static and dynamic communication. Unlike much of the flow analysis work they formulate a number of traditional data flow computations as lattice frameworks. They describe a mechanism for producing the operator space from the text of the program. When the channel over which a message is sent is determined by the value of a variable, data flow analysis is more complicated. Reif and Smolka use a preliminary data flow computation to determine the possible values of channel variables, overestimating the set of potential communication partners. The analysis continues from this point as in the case where communication channels are determined statically. Unfortunately, this research did not harness the flexibility of the lattice framework to express a wide range of program properties.

Callahan et. al. [8] describe a technique for computing the $B_4$ relation, a conservative approximation to the set of statements of a concurrent program that are guaranteed to execute before a specified statement. This work is set in the context of FORTRAN programs written with explicit parallel case and DO loops. Loop iterations and case arms are treated as separate tasks and contain explicit inter-task synchronization statements. The statement ordering information is used to detect when data-dependences in the program are violated by possible parallel executions of the program. The concurrent program is represented as a flow graph with explicit communication edges. Flow equations are defined that produce information for each flow graph node about the nodes that must precede it in all executions. This approach is tuned to the computation of a single property, the ordering relation, and as such lacks generality. While no empirical evidence is provided, the authors make claims about the utility of the analysis. Its not clear from this work how useful the analysis is for a broader class of programs as the structure of FORTRAN parallel DO
loops may not be indicative of the kinds of control and communication structure that concurrent programs exhibit.

Due to writererror [22] extends the work of Callahan to Ada tasking programs and demonstrates the utility of statement ordering information for data race and static deadlock analysis.

Masticola [52] describes a technique to efficiently compute a set of potential static deadlock cycles for Ada tasking programs. A concurrent program is represented as a flow graph with explicit communication edges; a modified form of the flow graph, with control flow loops unrolled once and back edges removed, is used for deadlock detection. As a pessimistic analysis, this approach is based on necessary conditions. Masticola defines 5 necessary conditions for the class of communication deadlocks the technique is designed detect: the presence of a cycle in the modified flow graph, that cycles must be of a restricted form, that no two nodes on the cycle may attempt communication with each other, that all nodes on a cycle must be executable in the same run of the program and at the same time, and that no other nodes in the flow graph can successfully communicate. The second and third conditions are encoded in the modified flow graph so that detection of a cycle implies that it satisfies those conditions. No tractable way to enforce the last necessary condition has been found. The fourth necessary condition is formulated as a data flow computation. If the conjunction of these 5 conditions is false then there are no static deadlocks in the program, otherwise the results are inconclusive and a set of potential static deadlocks are presented to the user for further analysis.

A range of programs were analyzed for the presence of deadlock by the technique of Masticola. The efficiency of performing the analysis and accuracy of the results were compared to a reachability analysis that folds data state information. Unfortunately, for the non-trivial programs considered no attempt was made to determine whether the reachability analysis was itself producing inconclusive results. Using the results
of another analysis, rather than using the actual program executions, as the baseline against which the precision of analysis results are judged is a useful adjunct to, but not a substitute for, more thorough verification. It should be noted that, unlike most other research in static analysis, this work did attempt a broad empirical evaluation on realistic concurrent programs. This is a good example of how focusing on a single property may allow for increased precision in analysis results.

An important aspect of Masticola and Ryder’s work involves non-concurrency analysis; it consists of a series of data flow analyses designed to determine whether it is never the case that two statements execute concurrently. Note that this analysis computes a sufficient condition for the negation of the fourth necessary condition described above. This analysis has much the same flavor as the deadlock analysis, except that here we have a collection of sufficient conditions for statement concurrency and we disjoin them. The conditions used include general conditions like \( B_4 \) and more program specific conditions such as the existence of critical sections in a program. Without an effective refinement of the flow graph representation by the results of non-concurrency analysis the accuracy of Masticola and Ryder’s deadlock analysis suffers.

Mercouroff [54] presents a data flow analysis that computes a conservative approximation to the set of pairs of send and receive statements that can co-execute to produce an inter-task communication. Programs in a subset of CSP are, for the purpose of our discussion, represented as program flow graphs with communication represented implicitly. Mercouroff uses the concept of the rank of a send or receive statement with respect to the channel over which it communicates. The idea is similar to the event count necessary condition used in constrained expressions analysis. The rank of a send (receive) for a channel, represents the number of communications over that channel in some prefix of a program execution that leads to the send (receive). Rank is represented as a set since there may be many execution paths leading to a
statement and the number of communications performed over each may be different. In fact it is possible that there are an infinite number of rank values. Mercouroff explores a number of lattice frameworks to represent this information pointing out some sources of imprecision in each of them. The most precise framework uses a symbolic equivalence that partitions rank values into equivalence classes of the form \( \{a+bx \mid x \in N\} \) for fixed values of \( a \) and \( b \). Values propagated through the flow graph are sets of pairs \((a,b)\) plus an additional bounded set of individual rank values. The symbolic representation attempts to capture rank patterns for communication statements in loops and the bounded set handles straight line initial and final rank values. For programs with a well defined termination point one can compute the number of communications that follow a statement by performing the analogous backwards data flow analysis. Rank information can be used to estimate the pairs of send and receive statements that can communicate during some program execution. We know that if the intersection of two rank sets is empty there is no execution on which the statements can communicate. No empirical evidence is provided other than the trivial examples used to explain the analysis. One of the main contributions of the work is consideration of a variety of lattice frameworks. If the ranks of communication statements for real concurrent programs can be characterized by a few simple symbolic expressions then this approach holds promise.

Cheung and Kramer [9] present a data flow analysis that is capable of detecting the existence of a class of anomalies in concurrent programs. The analysis computes the set of unreachable program statements. The class of execution behaviors that can be reasoned about using such an analysis is unclear. Concurrent programs are represented as flow graphs with implicit communication. Cheung and Kramer adapt a data flow algorithm from Reif and Smolka [65] to handle synchronous communication. This basic algorithm is refined by information gained by computing the \emph{depend} relation and \emph{history sets}. A node \( x \) depends on another node \( y \), written \( xDy \), if all
program paths leading to node $x$ pass through node $y$. This should not be confused with Callahans $B4$ relation which says $xB4y$ means that all occurrences of $x$ precede all occurrences of $y$; $xDy$ means that some occurrence of $x$ precedes all occurrences of $y$. The history set of node $x$ is the set of nodes that appear in all program paths leading to node $x$. History sets require a number of bits that is quadratic in the number of nodes in the flow graph for storage. The improvement in accuracy arises because for a reachable node it must be the case that its history set contains all nodes on which it depends; the precision of the unreachable node computation is improved by eliminating nodes that violate this condition. A final refinement distinguishes between flow graph nodes that are first reachable, nodes that can appear at most once in a program execution path, and those that are re-reachable. This effectively allows a refinement of the history sets of first reachable nodes by eliminating values that participate in cycles. Those values may reside in the history set by virtue of the fact that the computations enforce only necessary conditions and hence are overestimates. The necessary conditions developed in this work are interesting but the lack of empirical evidence limits the strength of the conclusions that can be drawn about their effectiveness.

We conclude with a data flow analysis technique for sequential programs. Olinger and Osterweil [60, 61] present an approach for analysis of sequential programs with respect to explicitly stated event sequencing properties. Cecil is a language for specifying sequencing constraints as regular expressions over an alphabet of possible program events. More specifically, Cecil allows for the definition of anchored quantified regular expressions (AQRE). These are event regular expressions where analysis may be restricted to a sub-graph of the entire program flow graph (anchoring) and which may be specified as invariance or impossibility properties (quantification). Many traditional data flow anomalies, such as undefined references and dead definitions, can be expressed as AQREs of length two. Cesar is a system for constructing a dis-
tributive lattice framework to check if the set of event sequences generated by the program flow graph satisfy a given AQRE. We discuss Cesar in more detail in Section 4.5. Cecil/Cesar attempts to provide a guaranteed efficient automated analysis, at the expense of some accuracy, and a flexible formalism for specifying useful correctness properties of a program. The analysis system as reported in [61] exhibited poor performance and as a result little experimentation was performed, leaving questions about the practicality of this technique.
CHAPTER 3
DATAFLOW ANALYSIS FRAMEWORKS FOR CONCURRENT PROGRAMS

Data flow analysis is a process for uncovering facts about executable program behavior without actually running the program. With applications to compiler optimization, program testing, validation and verification, data flow analysis is an important technique for a variety of software development activities. The constant desire for faster programs has spurred tremendous theoretical and practical advances in program analysis and optimization techniques. The bulk of this work has been for sequential programs.

The development of data flow frameworks by Kildall [46] revolutionized compiler optimization. Setting data flow analysis on a sound formal foundation spurred a wealth of research into both generalizing, e.g., [17, 29, 43, 72] and specializing Kildall's result, e.g., [42, 72, 80]. This resulted in formalisms for specifying and classifying data flow analysis problems, and the development of algorithms for solving any problem in a given class. This meant that compiler developers no longer had to hand-craft a system of equations and a solver for an analysis problem. Instead, the formal machinery of data flow frameworks could be used to describe a data flow problem and to guide the selection of an appropriate ready-made solution algorithm for that problem, thereby, reducing the cost of building analyzers.

Compiler optimization research has adapted to the emergence of concurrent and parallel computing into the programming mainstream A large body of work has been developed for using data flow analyses to aid in parallelizing of sequential programs,
e.g., [87]. More recently, data flow analyses and optimizations for explicitly concurrent and parallel programs have been investigated, e.g., [32, 71]. Unfortunately, the theory of data flow frameworks has not been updated to accommodate the new program representations and analyses that concurrency and parallelism have brought.

In the next section, we present a review of data flow frameworks followed by a discussion of the limitations of those frameworks. We then present a generalization of Kam and Ullman's monotone data flow analysis framework [43]; this generalization, called a complete-lattice monotone data flow analysis framework, supports the formulation of data flow analyses over flow graphs for concurrent programs. We present solution algorithms for data flow analysis problems formulated as complete-lattice frameworks and prove results related to their correctness and complexity. To illustrate these ideas, we discuss the formulation of a well-studied data flow analysis problem as a complete-lattice framework.

3.1 Background

We begin with some definitions and terminology. In this section, we present results about data flow analyses formulated as data flow frameworks, using the terminology and definitions in Hecht [34].

Every data flow analysis problem computes a different kind of problem information. This information captures facts about executable program behavior that the analysis is designed to gather; this information is inherently approximate. We can gather more (less) precise approximations with a commensurate increase (decrease) in the cost of analysis.

**Definition 1** A meet semi-lattice is a triple, \( L = (V, \sqsubseteq, \sqcap) \), where \( V \) is a set of values, \( \sqsubseteq \) is a partial-order defined over \( V \), and \( \sqcap \) is binary operation defined over \( V \), such that the following semi-lattice properties hold:
The lattice values encode information about the program that we are interested in collecting. The values in $V$ are partially-ordered by the $\sqsubseteq$ operator; this describes which values contain the information of other values. The meet operator $\sqcap \mid V \times V \to V$ is used for combining values. Intuitively, the bottom value, $\bot$, is less than all other lattice values if we interpret $\sqsubseteq$ as $\lt$. We can optionally have a top value, $\top$, that is greater than all other values. A chain in the lattice is a collection of values that are all ordered with respect to one-another; it is a linear ordering of lattice elements. While $V$ need not be finite, for our purposes we are only interested in lattices that have chains of finite length; note that we allow infinitely many such chains. The height of a lattice is the length of the longest chain. The standard example of a meet-semilattice is a powerset, $\mathcal{P}(S)$, where the values are subsets of a given set, $S_i \subseteq S$, the values are ordered by $\sqsubseteq$ and meet is $\cap$. For this lattice $\bot = \emptyset$, $\top = S$, and its height is the number of elements in $S$.

**Definition 2** A function space $F$ over a meet semi-lattice, is set of functions, $f \mid V \to V$, defined over the lattice values.

The functions in $F$, called transfer functions, are used to capture the local effects of parts of the programs computation. There are a number of function space properties
that can be used to classify a function space with respect to a lattice:

\[ \exists f_{\text{ident}} \in F : \forall v \in V : f_{\text{ident}}(v) = v \]  \hspace{1cm} (3.7)

\[ \forall v \in V : \exists f_v \in F : \forall x \in V : f_v(x) = v \]  \hspace{1cm} (3.8)

\[ \forall f, g \in F : f \circ g \in F \]  \hspace{1cm} (3.9)

\[ \forall f \in F : \forall x, y \in V : f(x \sqcap y) \subseteq f(x) \sqcap f(y) \]  \hspace{1cm} (3.10)

\[ \forall f \in F : \forall x, y \in V : f(x \sqcap y) = f(x) \sqcap f(y) \]  \hspace{1cm} (3.11)

These properties specify the existence of an identity, zeros, closure under composition, monotonicity and distributivity, respectively.

**Definition 3** A monotone function space satisfies properties 3.7-3.10.

**Definition 4** A distributive function space satisfies properties 3.7-3.11.

Combining a lattice of values and a collection of transfer functions yields a (somewhat abstract) description of a class of data flow analysis problems.

**Definition 5** A data flow framework is a pair, \( D = (L, F) \), where \( L \) is a meet semi-lattice and \( F \) is a function space defined over that lattice.

The set of possible program executions is modeled as a rooted directed graph.

**Definition 6** A flow graph is a directed graph, \( G = (N, E, n_0) \), where \( N \) is set of nodes, \( E \) is an edge relation over nodes, and \( n_0 \in N \) is a distinguished start node.

For our purposes we restrict the set of edges such that \( \forall m \in N : (m, n_0) \notin E \). For convenience, we define the set of predecessor, \( \text{Preds}(n) = \{ m : (m, n) \in E \} \), and successor nodes, \( \text{Succs}(n) = \{ m : (n, m) \in E \} \).

Associating the components of a data flow framework with components of a flow graph yields a (more concrete) description of a class of data flow analysis problem.
Definition 7 An instance, $I = (G, M)$ of a data flow framework, $D$, is a binding of transfer functions to flow graph nodes. This binding is accomplished through the definition of a function map, $M|N \to F$.

This mapping associates a function from $F$ with each node; the transfer function captures the effects of that node with respect to the information being gathered and represented as lattice values. For convenience, we will write the function $M(n)$ as $f_n$.

The solution of a data flow analysis problem is an approximation to the problem information at each node in the flow graph. Data flow analysis problems can be formulated directly as equations that give the information for a node in terms of the information at its predecessors and functions that capture the local effects of a node:

$$X[n] = f_n(X[pred_1], \ldots, X[pred_k])$$

Where $X[n]$ is the problem information at node $n$. A data flow framework is a formalism for describing a class of data flow analysis problems. It provides the types for the $X[.]$ variables and the functions $f$. An instance of a framework provides information that tells us which variables and functions to equate; thus, providing a system of related equations:

$$X[n_0] = \bot$$
$$\forall n \in N - \{n_0\} : X[n] = \sqcap_{i \in preds(n)} f_i(X[i])$$

A data flow analysis problem is solved by computing a fixed point of the equation system. This solution associates a lattice value, representing problem information, to each flow graph node. For problems formulated as an instance of a monotone data flow framework, the maximum fixed point (MFP) can be computed efficiently and is
known to be unique. If the framework is distributive, then the MFP solution contains, for each node, the problem information that can be computed by considering every path from the start node to the given node; this is called the meet over paths (MOP) solution. This solution is possible because $\sqcap$ distributes over $f \in F$. We can move the $\sqcap$ operation to the outside and apply it a single time to $f_p(\bot)$, where for a path, $p = n_0, n_1, \ldots, n_k$, the composition of transfer function of the nodes on the path is $f_p(.) = f_{n_k}(\ldots(f_{n_1}(f_{n_0}(.)))) \ldots$. We define the set of paths leading from $n_0$ to node $n$ as $Paths(n) = \{P|P = n_0, \ldots, n_k = n\}$ where, for integer $i$ such that $1 < i < k$ we have $(n_{i-1}, n_i) \in E$. Given this we can write the MOP solution as:

$$A[n] = \sqcap_{p \in Paths(n)} f_p(\bot)$$

### 3.2 Limitations of Semi-lattice Frameworks

Placing data flow analysis on a sound formal foundation has brought many advantages to developers of new analyses. The ability to characterize an analysis framework [51], as rapid, continuous, distributive, monotone, or $k$-bounded for example, allows one to apply existing theoretical results to reason about convergence, bounds on running time, and the precision of analyses. In addition, a wide variety of general and specialized algorithms have been developed for classes of frameworks. Making it easy for developers of new analyses to choose the algorithm that is most appropriate for their data flow framework formulation of a problem.

The application of data flow analysis has been primarily in compiler optimization. Historically, the main thrust of that work has been on producing small/fast executables for programs written in sequential programming languages. The algorithmic and theoretical developments related to data flow analysis have mirrored this work by focusing on sequential programs. There has been some recent work on applying data

---

1 The MOP solution is often referred to as the best possible static solution for a given analysis problem.
flow analysis to concurrent programs. Like the early work on sequential program analysis this newer work has, to a great extent, involved the development of ad-hoc solutions to important practical problems.

As with all static analyses, data flow analyses reason about a model of program execution, i.e., the flow graph. A flow graph model can represent varying amounts of program information; it need only represent all the information that is necessary to support safe or conservative analyses. Given the wide variety of possible flow graph models it is fruitful to examine some common assumptions about these models. We discuss two such assumptions that underly much of the data flow analysis work to date.

A Path Models a Program Execution

Flow graph representations are often constructed so that for each prefix of a program execution that ends at the statement corresponding to node \( n \), there is a path in the graph from \( n_0 \) to \( n \). Typically, flow graphs will contain some paths that do not correspond to feasible executions of the program; this is done in order to represent the large, potentially infinite, number of different program executions in a tractable form.

For sequential programs it is easy to construct a flow graph from the source text. For programs written in high-level programming languages such a flow graph is typically sparse; each node only has a few neighbors. For interacting concurrent programs flow graph is more complicated to construct and often has a higher degree of node-connectedness.

Many models of concurrency allow for communication between processes and support multiple senders and receivers over a given communication channel. A flow graph representation for a program using this model must pessimistically assume that all pairs of senders and receivers for a given channel can communicate; although analysis
may be able to determine the impossibility of certain communications. Thus, the number of nodes in the flow graph will, in the worst-case, be quadratic in the number of program statements rather than linear.

In order for all feasible program executions of a concurrent program to be represented as a flow graph path, we must represent all possible interleavings of the independently executing statements in program processes. Even with exact information about the statements that can execute concurrently, the flow graph would still be very dense, i.e., $|E|$ will approach $|N|^2$. Unfortunately, the problem of computing that information is NP-complete [74].

As a consequence of this second implication, researchers have developed flow graphs that do not explicitly represent statement interleavings; these include the SCG [8], PFG [32], MIG [21], and sync hypergraph [53]. There are also flow graphs that can be viewed with or without explicit interleavings, such as the TFG [24]. Figure 3.1 depicts flow graphs for a sequential and a concurrent program. The sequential flow graph is standard. The concurrent flow graph can be viewed as a collection of

**Figure 3.1** Typical Sequential and Concurrent Flow Graphs
sequential control flow graphs, one for each process in the program (enclosed in a dashed box), consisting of (circular) nodes. Additional (square) nodes represent synchronization or communication points. Note that we are not attempting to define an all inclusive model of flow graphs for concurrent programs, rather we are identifying a number of common structural and semantic modeling features that differentiate them from flow graphs for sequential programs. The specific flow graphs mentioned above are all quite similar to the concurrent flow graph as described here.

These concurrent flow graphs model a program execution as a collection of paths, one for each process sub-graph, where those paths intersect at process interaction points. For such graphs, the MOP is no longer the best possible solution; in fact, it can be quite imprecise. Consider a classic any-path forward-flow data flow analysis problem such as reaching definitions [3]. A definition of variable $v$ at node $d$ is said to reach a use at node $u$ if there exists a path from $d$ to $u$ on which $d$ is not killed, i.e., $v$ is not re-defined. For the concurrent flow graph in Figure 3.1, if $def_1$, $def_2$ and $use$ all refer to the same variable, the MOP solution for this problem would say that $def_1$ reaches $use$. If the successor square node of $def_1$ is a synchronization node, however, then $def_1$ is killed by $def_2$ on all program executions leading to $use$.

**Semantics of flow graph merge points**

The semantics of merge points in such a sequential flow graph are uniform; a merge represents the confluence of two distinct program executions. In a concurrent flow graph nodes and edges model control flow information, synchronization information and communication information. Consequently, merge points can represent the confluence of parts of two distinct program executions or different parts of the same program execution.

For data flow analysis, flow graph merge points serve an important role; they are the points at which data gathered from separate regions of sequential program
execution are combined. In meet semi-lattice data flow frameworks this combining operation is the meet operation. For the concurrent flow graphs described above, it appears that being limited to a single combining operation can lead to significant imprecision in analysis results. Srinivasan [71] has also identified the necessity of multiple combining operations, represented as $\phi$ and $\psi$ nodes, in defining an SSA form for explicitly parallel programs.

### 3.3 Complete-lattice Frameworks

In this section, we generalize semi-lattice frameworks to allow formulation of data flow problems over a complete-lattice. We extend a number of important results for monotone semi-lattice frameworks to complete-lattice frameworks. These new results address the limitations of semi-lattice frameworks and make the advantages of frameworks available to developers of analyses for concurrent programs.

**Definition 8** A complete-lattice is a triple, $L = (V, \sqsubseteq, \cap, \cup)$, where $V$ is a set of values, $\sqsubseteq$ is a partial-order defined over $V$, and $\cap$ and $\cup$ are binary operation defined over $V$. These operations are such that for all subsets $S \subseteq V$, $\cap_{x \in S}$ and $\cup_{x \in S}$ are defined. In addition to the semi-lattice properties (3.1 - 3.6), the components of $L$ must satisfy the following lattice properties:

\[ x \sqcup y \sqsubseteq y \]
\[ x \sqcup y = y \iff x \sqsubseteq y \]
\[ x \sqcup x = x \]
\[ x \sqcup y = y \sqcup x \]
\[ (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \]
\[ x \sqcup (x \sqcap y) = x \]

This is the standard lattice-theoretic definition of a complete lattice [19].
A complete-lattice is essentially a meet semi-lattice with an additional idempotent, associative, commutative operation \( \sqcup \) called *join*. Height and \( \top \) are defined in the same way as for a semi-lattice. The standard example of a complete-lattice is a powerset, \( \mathcal{P}(S) \), where the values are subsets of a given set, \( S \subseteq S \), the values are ordered by \( \subseteq \), meet is \( \cap \), and join is \( \cup \). For this lattice \( \bot = \emptyset \), \( \top = S \), and its height is the number of elements in \( S \).

**Observation 1** If we have a finite collection of lattice values:
\[ x_1, x_2, \ldots, x_k, x_1', x_2', \ldots, x_k' \in V, \]
where for \( i \) such that \( 1 \leq i \leq k \) we have \( x_i \sqsubseteq x_i' \), then:
\[ \cap_{1 \leq i \leq k} x_i \sqsubseteq \cap_{1 \leq i \leq k} x_i' \]
and
\[ \cup_{1 \leq i \leq k} x_i \subseteq \cup_{1 \leq i \leq k} x_i'. \]

This follows from Lattice properties 3.1, 3.18 and the associativity, and commutativity of \( \cap \) and \( \cup \).

We define a function space, \( F \), over the values, \( V \), just as we do for a semi-lattice. Monotonicity and distributivity of the function space requires that function space properties (3.7-3.10) and (3.7-3.11), respectively, be met.

**Definition 9** A *complete-lattice data flow analysis framework*, is a pair, \( D = (L, F) \), where \( L \) is a complete-lattice and \( F \) is a function space defined over that lattice.

**Definition 10** An instance, \( I = (G, N_\cap, N_\cup, M) \), of a *complete-lattice data flow analysis framework*, \( D \), is a quadruple consisting of a flow graph \( G \), subsets of the flow graph nodes \( N_\cap \) and \( N_\cup \), and a function map \( M | N \to F \), where:

\[ n_0 \not\in N_\cap \]
The set $N_{\cap}$ contains the flow graph nodes for which $\cap$ is used to merge predecessor values. The set $N_{\cup}$ contains the flow graph nodes for which $\cup$ is used to merge predecessor values. Intuitively, these sets serve to map merge operations onto flow graph nodes just as $M$ maps transfer functions onto flow graph nodes. A complete-lattice framework describes a class of data flow analysis problems; it induces the following system of simultaneous equations:

\[
X[n_0] = \bot
\]

\[
\forall n \in N_{\cap} : X[n] = \sqcap_{p \in \text{Preds}(n)} f_p(X[p])
\]

\[
\forall n \in N_{\cup} : X[n] = \sqcup_{p \in \text{Preds}(n)} f_p(X[p])
\]

We note that if $N_{\cup} = \emptyset$ then this equation system is equivalent to one derived from the semi-lattice framework embedded in the complete-lattice framework. A data flow analysis problem is solved by computing a fixed point of the equation system, as described above.

**Solving Complete-Lattice Data Flow Problems**

We generalize existing results for monotone semi-lattice frameworks to monotone complete-lattice frameworks by mirroring Kam and Ullman's generalization of Kil-dall's distributive semi-lattice frameworks [43].

In the following, we assume the existence of $\top$. If it does not exist we can introduce a new element and adjust the lattice and functions accordingly, such that:

\[
\forall f \in F : \forall x \in V : \top \sqcap x = x \sqcap \top = x
\]
\[ T \cup x = x \cup T = T \]

\[ f(\top) = \top \]

Algorithm 1 (General Iterative Solver)

\textit{Input:}

An instance \( I = (G, N_\cap, N_\cup, M) \) of \( D = (L, F) \), a monotone complete-lattice framework.

\textit{Output:}

A lattice value for each node, \( \forall n \in N : A[n] \in V \).

\textit{Initialization Step:}

\[ \forall n \in N \quad A[n] = \begin{cases} \bot & \text{if } n = n_0 \\ \top & \text{otherwise} \end{cases} \]

\textit{Iteration Step:}

Visit the nodes, other than \( n_0 \), in order \( n_1, n_2, \ldots \). Nodes can be visited multiple times and the order is not fixed prior to running the algorithm.

A \textit{visit} of a node, \( n \), is the evaluation of one of the assignments:

\[ A[n] = \cap_{p \in \text{Preds}(n)} f_p(A[p]) \quad \text{if } n \in N_\cap \]

\[ A[n] = \cup_{p \in \text{Preds}(n)} f_p(A[p]) \quad \text{if } n \in N_\cup \]

The sequence of nodes visited must satisfy two conditions:

\textbf{No premature termination}

If there exists a node \( n \in N_\cap (n \in N_\cup) \) such that \( A[n] \neq \cap_{p \in \text{Preds}(n)} f_p(A[p]) \)

(\( A[n] \neq \cup_{p \in \text{Preds}(n)} f_p(A[p]) \)) after visiting node \( n_i \) in the sequence,

then there is an integer \( j > i \) such that \( n_j = n \). Intuitively, if we
reach a point in the sequence where some node’s current value has not been updated by newer information at its predecessors then the sequence must continue until that update is performed.

**Termination after stabilization**

If after visiting node \( n_i \), \( A[n] = \cap_{p \in \text{Preds}(n)} f_p(A[p]) \) for all \( n \in N_L \) and \( A[n] = \cup_{p \in \text{Preds}(n)} f_p(A[p]) \) for all \( n \in N_U \), then the sequence will eventually halt. Intuitively, once we have reached a point where all the current node values are up to date it is safe to terminate the algorithm.

In the presentation that follows we will refer to the \( j^{th} \) step of Algorithm 1 to mean the state of the algorithm after visiting the first \( j \) nodes in the node sequence. The value at a node, \( n \), on the \( j^{th} \) step will be denoted \( A^{(j)}[n] \).

The constraints on the node visitation sequence express certain requirements of the algorithm with respect to termination; these requirements, however, do not guarantee termination. The following lemma proves termination.

**Lemma 1 (General Iterative Solver Terminates)**

> Given an instance \( I = (G, N_L, N_U, M) \) of a monotone complete-lattice data flow analysis, \( D = (I, F) \), Algorithm 1 will eventually terminate.

*Proof:* The proof is by induction on \( m \), the number of steps performed by Algorithm 1. We show that for all nodes, \( n \), in \( G \), \( A^{(m+1)}[n] \sqsubseteq A^{(m)}[n] \).

*Base Step:* \( m = 0 \)

At this point the algorithm has completed its initialization phase and visited a single node, \( n_1 \). For the start node, \( n_0 \), we have \( A^{(1)}[n_0] = \perp \sqsubseteq \perp = A^{(0)}[n_0] \). We have yet to visit any of the other nodes, \( n \in N \setminus \{n_0, n_1\} \), and those nodes remain at their initial values, so \( A^{(1)}[n] = \top \sqsubseteq \top = A^{(0)}[n] \).
We visit $n_1$ on this step, but since the value at step 0 at $n_1$ was $\top$ we do not need to bother evaluating $A^{(1)}[n_1]$; by definition all values are $\subseteq \top$.

**Inductive Step**: At step $m + 1$ we have $\forall n \in N : A^{(m)}[n] \subseteq A^{(m-1)}[n]$

At this point in the algorithm we are at step $m + 1$ and have just visited node $n_{m+1}$. We note that at each step in the algorithm the value of at most a single node can change; for step $i$ it is the value at node $n_i$. Thus, for the values at node $n \in N - \{n_{m+1}\}$ will remain unchanged, so:

$$\forall n \in N - \{n_{m+1}\} : A^{(m+1)}[n] = A^{(m)}[n] \Rightarrow A^{(m+1)}[n] \subseteq A^{(m)}[m]$$

Let $l$ be the most recently preceding step at which node $n_{m+1}$ was visited; if $n_{m+1}$ has never been visited then $l = 0$. The sequence of node visits is $\ldots n_l, \ldots, n_m, n_{m+1}, \ldots$, where $n_l = n_{m+1}$.

If for all $i$ such that $l < i < m + 1$, it is the case that $n_i \notin \text{Preds}(n_{m+1})$, then $A^{(m+1)}[n_{m+1}] = A^{(m)}[n_{m+1}] = A^{(l)}[n_{m+1}]$, since none of its predecessors have changed value. In this case the induction trivially holds.

If there exists an $i$ such that $l < i < m + 1$ and $n_i \in \text{Preds}(n_{m+1})$, then some number of predecessors may have changed value. We refer to the set of predecessors of $n_{m+1}$ who have changed value since step $l$ as $\text{Changed}$; we also define $\text{NotChanged} = \text{Preds}(n_{m+1}) - \text{Changed}$. There are two cases to consider in evaluating the new value at $n_{m+1}$:

**case 1**: $n_{m+1} \in N_l$

We write out the values at steps $m$ and $m + 1$ more explicitly to show their relationship.

$$A^{(m)}[n_{m+1}] = A^{(l)}[n_{m+1}]$$

$$= (\cap_{p \in \text{Changed}} f_p(A^{(l)}[p])) \cap (\cap_{p \in \text{NotChanged}} f_p(A^{(l)}[p]))$$
\[ A^{(m+1)}[n_{m+1}] = (\sqcap_{p \in Changed} f_p(A^{(m)}[p])) \sqcap (\sqcap_{p \in NotChanged} f_p(A^{(l)}[p])) \]

Focusing on the right hand side of these two equations, we have:

\[ \forall p \in NotChanged : f_p(A^{(l)}[p]) \sqsubseteq f_p(A^{(l)}[p]) \]
\[ \forall p \in Changed : f_p(A^{(m)}[p]) \sqsubseteq f_p(A^{(l)}[p]) \]

from the definition of \( \sqsubseteq \) and the inductive hypothesis. By Observation 1 the iterated joins of the elements on the right hand sides satisfy the \( \sqsubseteq \) relation. So, \( A^{(m+1)}[n_{m+1}] \sqsubseteq A^{(m)}[n_{m+1}] \) and the induction holds.

**case 2:** \( n_{m+1} \in N_\sqcup \)

We write out the values at steps \( m \) and \( m+1 \) more explicitly to show their relationship.

\[ A^{(m)}[n_{m+1}] = A^{(l)}[n_{m+1}] \]
\[ = (\sqcup_{p \in Changed} f_p(A^{(l)}[p])) \sqcup (\sqcup_{p \in NotChanged} f_p(A^{(l)}[p])) \]
\[ A^{(m+1)}[n_{m+1}] = (\sqcup_{p \in Changed} f_p(A^{(m)}[p])) \sqcup (\sqcup_{p \in NotChanged} f_p(A^{(l)}[p])) \]

Focusing on the right hand side of these two equations, we have:

\[ \forall p \in NotChanged : f_p(A^{(l)}[p]) \sqsubseteq f_p(A^{(l)}[p]) \]
\[ \forall p \in Changed : f_p(A^{(m)}[p]) \sqsubseteq f_p(A^{(l)}[p]) \]

from the definition of \( \sqsubseteq \) and the inductive hypothesis. By Observation 1 the iterated joins of the elements on the right hand sides satisfy the \( \sqsubseteq \) relation. So, \( A^{(m+1)}[n_{m+1}] \sqsubseteq A^{(m)}[n_{m+1}] \) and the induction holds.
We now turn our attention to the conditions for node visitation sequences. After the \(i^{th}\) step we have one of two cases, either the values have stabilized, in which case the sequence will terminate, or there is some value that has yet to be updated and the sequence continues. The premature termination condition guarantees that a node that needs updating will get updated. There are a finite number of nodes in \(G\); for each node there are a bounded number of values that can be taken on in descending through the lattice, since the lattice has finite length chains and a \(\perp\) element. Consequently, node updates can occur only a finite number of times. Therefore, the sequence is finite and the Algorithm 1 terminates.

\[\Box\]

We now show that the solution computed by this algorithm is, in fact, the maximum fix-point of the system of equations described earlier.

**Theorem 1 (General Iterative Solver Computes MFP)**

*Given an instance, \(I = (G, N_{\Gamma}, N_{\cup}, M)\), of a monotone complete-lattice data flow analysis framework, \(D = (L, F)\), if we apply Algorithm 1 then the solution, the \(A[n]\), is the maximum fix point solution of the following system of equations:*

\[
\begin{align*}
X[n_0] &= \perp \\
\forall n \in N_{\Gamma}: X[n] &= \bigcap_{p \in \text{Preds}(n)} f_p(X[p]) \\
\forall n \in N_{\cup}: X[n] &= \bigcup_{p \in \text{Preds}(n)} f_p(X[p])
\end{align*}
\]

**Proof:** By the definition of the algorithm it is clear that once Algorithm 1 halts, the \(A[n]\) are solutions to the above equations. We need to show that these \(A[n]\) are greater or equal to any other solution to the equations, \(B[n]\). We prove, by induction on \(m\), the number of steps taken in Algorithm 1, that after the \(m^{th}\) step \(\forall n \in N: B[n] \subseteq A^{[m]}[n]\).
The only possible solution at node \( n_0 \) is \( \bot \), so \( B[n_0] = \bot \). At step 0 we have completed initialization so \( A^{(0)}[n_0] = \bot \), and the induction holds. For the other nodes \( n \in N - \{n_0\} \) the initial value is \( \top \), so regardless of its value \( B[n] \sqsubseteq \top = A^{(0)}[n] \).

**Inductive Step:** At step \( m \) we have \( \forall n \in N : B[n] \sqsubseteq A^{(m-1)}[n] \)

At step \( m \) we visit node \( n_m \) to compute a new value; all other nodes keep their same value, \( \forall n \in N - \{n_m\} : A^{(m)}[n] = A^{(m-1)}[n] \). Using the inductive hypothesis we see that the inductive step holds for these nodes.

To compute the new value at node \( n_m \) there are two cases:

\( n_m \in N_\cap \)

By definition the solution we have \( B[n_m] \) and from the proof of Lemma 1 we have:

\[
B[n_m] = \cap_{p \in \text{Preds}(n)} f_p(X[p])
\]

\[
A^{(m)}[n_m] = \cap_{p \in \text{Preds}(n)} f_p(A^{(m-1)}[p])
\]

From the inductive hypothesis we know that

\[
\forall p \in \text{Preds}(n_m) : B[p] \sqsubseteq A^{(m-1)}[p]
\]

The monotonicity of the function space gives us that

\[
\forall p \in \text{Preds}(n_m) : f_pB[p] \sqsubseteq f_p(A^{(m-1)}[p])
\]

From Observation 1, since the elements of the iterated meets in the definitions of \( B[n_m] \) and \( A^{(m)}[n_m] \) are pair-wise ordered so are their
iterated meets. Thus, \( B[n_m] \sqsubseteq A^{(m)}[n_m] \) and the inductive step holds for \( n_m \).

\( n_m \in N \cap \)

By definition the solution we have \( B[n_m] \) and from the proof of Lemma 1 we have:

\[
B[n_m] = \sqcup_{p \in \text{Pre}ds(n)} f_p(X[p])
\]

\[
A^{(m)}[n_m] = \sqcup_{p \in \text{Pre}ds(n)} f_p(A^{(m-1)}[p])
\]

From the inductive hypothesis we know that

\[
\forall p \in \text{Pre}ds(n_m) \ : \ B[p] \sqsubseteq A^{(m-1)}[p]
\]

The monotonicity of function space gives us that

\[
\forall p \in \text{Pre}ds(n_m) \ : \ f_pB[p] \sqsubseteq f_p(A^{(m-1)}[p])
\]

From Observation 1, since the elements of the iterated joins in the definitions of \( B[n_m] \) and \( A^{(m)}[n_m] \) are pair-wise ordered so are their iterated joins. Thus, \( B[n_m] \sqsubseteq A^{(m)}[n_m] \) and the inductive step holds for \( n_m \).

The theorem follows from these results and the fact that Algorithm 1 is guaranteed to terminate by Lemma 1.

\( \square \)

Algorithm 1 is specified as weakly as possible; more refined implementations are free to choose from a wide variety of data structures and node visitation orders. We describe one such implementation here. It is an adaptation of Hecht’s iterative worklist algorithm [34].

**Algorithm 2 (Iterative Worklist Solver)**

*Input:*
An instance $I = (G, N_U, N_L, M)$ of $D = (L, F)$, a monotone complete-lattice framework.

Output:

A lattice value for each node, $\forall n \in N : A[n] \in V$.

We use two auxiliary data structures in this algorithm: $W$ which is an ordered sequence of nodes with at most $|N|$ elements and $v$ a single lattice value.

Initialization:

$$\forall n \in N \quad A[n] = \begin{cases} \perp & \text{if } n = n_0 \\ \top & \text{otherwise} \end{cases}$$

$$W = \{ m : (n_0, m) \in E \}$$

Main Loop:

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

1. $W = n, n_1, n_2 \ldots n_k$ at this point $W = n, n_1, n_2 \ldots n_k$
2. $v = A[n]$  
3. if $n \in N_U$ then
4. $A[n] = \bigcap_{p \in \text{Preds}(n)} f_p(A[p])$

else
5. $A[n] = \bigcup_{p \in \text{Preds}(n)} f_p(A[p])$

end if
6. if $v \neq A[n]$ then
7. $W = n_1, n_2, \ldots, n_k, n_{s_1}, n_{s_2}, \ldots$

---

3We describe the worklist as initially containing $k + 1$ elements; this is a notational convenience and if $k = 0$ then $W = n$. 

52
where \( n_{s_i} \in \text{Succs}(n) \) and for \( 1 \leq l \leq k \quad n_{s_i} \neq n_l \)

end if

We now show that this iterative worklist solution algorithm computes the same results as the general solver algorithm, which was shown in Theorem 1 to be the MFP of the induced system of equations.

**Theorem 2 (Iterative Worklist Solver Correctness)**

*Given an instance \( I = (G, N_{ri}, N_{li}, M) \) of a monotone complete-lattice data flow analysis framework, \( D = (L, F) \), Algorithm 2 computes a solution, the \( A[n] \), that is the maximum fix point solution of the following system of equations:*

\[
\begin{align*}
X[n_0] &= \bot \\
\forall n \in N_{ri}: X[n] &= \cap_{p \in \text{Preds}(n)} f_p(X[p]) \\
\forall n \in N_{li}: X[n] &= \cup_{p \in \text{Preds}(n)} f_p(X[p])
\end{align*}
\]

*Proof:* Recall that Algorithm 1 operated by performing a sequence of node visits, where the sequence had two constraints. We prove this theorem by showing that the computation performed by Algorithm 2 is equivalent to such a sequence of node visits.

The initialization phases of both algorithms are identical. A visit in Algorithm 1 is equivalent to executing statements 3-5 in Algorithm 2. The rest of the statements in Algorithm 2 are used to maintain the worklist, \( W \), which enforces the node visitation sequence. We show that these statements work together to satisfy the conditions on visitation sequences expressed in Algorithm 1. We consider just the case of nodes in \( N_{ri} \); for nodes in \( N_{li} \) we make an identical argument.

**No premature termination** The condition states:
if \( \exists n \in N_n : A[n] \neq \cap_{p \in \text{Preds}(n)} f_p(A[p]) \) after visiting node \( n_i \) in the sequence, then there is an integer \( j > i \) such that \( n_j = n \).

Let the last time we visited node \( n \) be at step \( last \) where \( last < i \), \( n_{\text{last}} = n \). We have \( A^{(last)}[n] = \cap_{p \in \text{Preds}(n)} f_p(A^{(last-1)}[p]) \). The only way that \( \cap_{p \in \text{Preds}(n)} f_p(A[p]) \) can change value is if one of the predecessors, say \( p_1 \), changes value and that requires a visit to node \( p_1 \). Let such a visit occur at step \( new \) where \( last < new \leq i \) and there are no other visits to \( p_1 \) at steps \( last \) through \( new - 1 \). This visit will occur on the \( new^{th} \) iteration of the body of Algorithm 2; we remove \( p_1 \) from the head of \( W \). Line 2 of the algorithm saves the old value of \( A[p_1] \) which is \( A^{(last)}[p_1] \) in our local variable \( v \). Our visit of \( p_1 \), lines 3-5, compute a new value \( A^{(new)}[p_1] \sqsubset A^{(last)}[p_1] \); the values are not equal because of the assumption that the predecessor changes its value. The test at line 6 succeeds and we proceed to add all successors of \( p_1 \) to \( W \) that are not already in \( W \). Thus, we are assured that \( n \) is on the worklist after our visit to \( p_1 \). Since, the worklist contains at most \( |N| \) elements we are guaranteed to visit node \( n \) before step \( i + |N| + 1 \). Thus, the condition is satisfied since there exists a \( j \) such that \( i < j < i + |N| + 1 \).

**Termination after stabilization** The condition states:

If after visiting node \( n_i \), \( A[n] = \cap_{p \in \text{Preds}(n)} f_p(A[p]) \) for all \( n \in N_n \), then the sequence will eventually halt.

After we visit node \( n_i \) we may visit additional nodes. Let \( n \) be the next node we visit at step \( i + 1 \); it is taken off of \( W \) at line 1. We store the current value \( A^{(i+1)}[n] \) in local variable \( v \) at line 2. Our visit to
node \( n \), in lines 3-5, computes \( A^{(i+1)}[n] = \cap_{p \in \text{Prede}(n)} f_p(A^{(i)}[p]) \). The assumption in the condition is that \( A^{(i)}[n] = \cap_{p \in \text{Prede}(n)} f_p(A^{(i)}[p]) \) for all \( n \in N \). So, \( A^{(i+1)}[n] = A^{(i)}[n] = v \) and the test at line 6 fails; we add no nodes to \( W \). Subsequent iterations all behave in the same way because the visits \( i, i+1, \ldots \) cause no change to \( A[n] \) for all \( n \in N \). Since we never execute line 7 in any of these visits, each visit reduces the number of elements on \( W \) by 1. Thus, there can be at most \( |N| - 1 \) more iterations of the body of Algorithm 2 after the \( i^{th} \), and the sequence of node visits halts at \( j \) where \( j < i + |N| \).

Since all phases of Algorithm 2 meet the specification of Algorithm 1 our theorem holds by Theorem 1.

\[ \square \]

We can express a bound on the running time of this algorithm in terms of the height, \( k \), of \( L \).

**Theorem 3 (Iterative Worklist Solver Complexity)**

Given an instance \( I = (G, N, N_\text{li}, M) \) of a monotone complete-lattice data flow analysis, \( D = (L, F) \), where \( L \) has height \( k \), Algorithm 2 it will terminate in \( O(k|N|^2) \) time.

**Proof:**

The initialization phase requires \( O(|N|) \) operations to assign values to each node and \( O(|N|) \) operations to initialize \( W \).

We use \( W \) as a queue that contains at most a single instance of each node. Lines 1 and 7 require standard enqueue/ dequeue operations. Querying whether \( W \) has a given successor node in line 7 requires a containment test. Both of these requirements can be handled efficiently by implementing \( W \) as a collection of nodes that is threaded as a queue and
maintaining a bit-vector representation of the set of nodes in $W$. In this case, lines 1-3 and 6 all require $O(1)$ operations.

The cost of lines 4 and 5 are equal, so we consider only one of them. Line 4 is the iterated meet over all predecessors, of the predecessors transfer function applied to the predecessors current value. A standard, and reasonable, assumption is that transfer functions, $f_n$, require $O(1)$ operations to evaluate. Thus, line 4 requires $O(|N|)$ operations since in the worst-case a node may have all other nodes as predecessors.

Line 7 requires that for each successor, we check for containment in $W$; if not contained we append the successor to $W$. The containment check requires $O(1)$ operations, a bit-vector test, and the append requires $O(1)$ operations, threading the node on the tail of the queue and setting a bit in the bit-vector. Thus, line 7 requires $O(|N|)$ operations since in the worst-case a node may have all other nodes as successors.

In total the body of the main loop requires $O(|N|)$ operations.

The number of iterations of the body, or visits to a node, can be bounded by making use of the conditions on the node visitation sequence from Algorithm 1. It was shown in the proof of Theorem 2 that the sequence of nodes taken from the $W$ satisfies those conditions. Recall, from the proof of Theorem 1, that the values at all nodes in the graph are descending monotonically through the lattice from $T$; the lowest they can go is $\bot$. In the worst case, each iteration of the main loop causes a single node to descend a single value in the lattice; multiple nodes descending multiple values in a single iteration is quite possible but only makes the algorithm converge faster. At least one node will descend otherwise the

**termination after stabilization** condition guarantees that Algorithm

\footnote{Evaluating a binary-tree of meets would require only $O(\log |N|)$ operations. In this analysis we don't bother doing that because the cost of this step is dominated by the cost of line 7.}
we will terminate in at most $|N|$ steps. At most a node can descend $k$, the height of the lattice, times before it hits $\bot$. Since there are $|N|$ nodes in the graph there are at most $k|N|$ descending iterations. Thus, there are $O(k|N|)$ iterations of the main loop.

The theorem holds since $O(k|N|)$ iterations of $O(|N|)$ operations is $O(k|N|^2)$.

Algorithm 2 is a practical improvement over Algorithm 1 because of the node visitation order it enforces. Nodes are only visited when they have the potential to change value. If the sequence of nodes removed from the worklist is $\ldots n_i, n_{i+1}, \ldots, n_j \ldots$ and $n_i = n_j = n$ then there is an integer $k$ such that $i < k < j$ and $n_k \in Preds(n)$; intuitively, we only put a node on the worklist if one of its predecessors changes values.

Careful consideration of Algorithm 2 reveals a number of additional opportunities for eliminating unnecessary computation. As described, for each node the algorithm stores a value that does not reflect the effects of the transfer function at that node; this is called the in value for a node. We can easily modify Algorithm 2 to keep track of the value that reflects the effects of a node’s transfer function; this is called the out value for a node. We can perform the comparison $out(n) \not > in(s)$ where $s \in Sucess(n)$ 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 $in(n) = in(n) \cap (\cap_{p \in Changed} out(p))$ rather than considering all predecessors. These optimizations do not improve the worst-case time bound for Algorithm 2, but for dense graphs and graphs whose nodes have high fan-in or fan-out they can yield significant practical speedup.
3.4 An Example

To illustrate the utility of complete-lattice data flow frameworks, we consider an example data flow analysis problem, which computes flow graph dominators. Intuitively, for a sequential program represented by a statement flow graph domination captures information about statements that must precede the execution of other statements on any program executions in which they both occur; we call this statement precedence information. Formally, $\text{Dom}(m) = n$ if all paths from $n_0$ to $m$ in $G$ pass through $n$. This is a safe approximation to statement precedence information in the sense that $\text{Dom}(m) = n$ implies that statement $n$ precedes statement $m$ on all program executions in which both occur.

Hecht presents a formulation of it as a distributive semi-lattice data flow analysis framework [34]:

\[
L = (\mathcal{P}(N), \subseteq, \cap)
\]

\[
F = \{f_{\text{idem}}(S) = S : S \subseteq \mathcal{P}(N)\} \cup \{f_n(S) = S \cup n : n \in N \land S \subseteq \mathcal{P}(N)\} \cup \{f_{\text{misc}}(X) = Y : X, Y \subseteq \mathcal{P}(N)\}
\]

Where the $f_{\text{misc}}$ are included to make $F$ closed under composition. Note that, for this problem, $\top = N$ and $\bot = \emptyset$. An instance of this framework is defined as $(G, M)$ where $M(n) = f_n$.

For concurrent programs we are also interested in gathering statement precedence information [8, 9, 22, 32, 52]. Unfortunately, using $\text{Dom}$ information often yields an overly pessimistic safe approximation. Recall the example concurrent flow graph in Figure 3.1. The statement $\text{end}$ is clearly preceded by the statement $\text{def}_1$, since all program executions that lead to the last statement of the process must pass through the first statement of the process. The existence of paths through the other processes
crossing to the middle process via one of the square nodes means that this precedence
relationship will not be captured by the Dom information.

Intuitively, we want to treat control flow merge points differently from communication/synchronization merge points. At a control flow merge point we know that one, but not all, of the predecessor statements has executed immediately prior to the merge. In this case, only statements that precede all of those predecessors are guaranteed to precede the merge point. At a synchronization merge point we know that all of the predecessor statements have executed immediately prior to the merge. In this case, the statements that precede any of those predecessors are guaranteed to precede the merge point. This problem can be formulated as a complete-lattice framework problem as follows:

\[
L = (\mathcal{P}(N), \subseteq, \cap, \cup) \\
F = \{ f_{\text{id}}(S) = S : S \subseteq \mathcal{P}(N) \} \cup \\
\{ f_n(S) = S \cup n : n \in N \land S \subseteq \mathcal{P}(N) \} \cup \\
\{ f_{\text{misc}}(X) = Y : X, Y \subseteq \mathcal{P}(N) \}
\]

Where the \( f_{\text{misc}} \) are included to make \( F \) closed under composition. Note that, for this problem, \( T = N \) and \( \bot = \emptyset \). An instance of this framework is defined as \((G, N_{\text{c}}, N_{\text{s}}, M)\). \( G \) is a concurrent flow graph whose control flow merge points and synchronization merge points are \( N_{\text{c}} \) and \( N_{\text{s}} \) respectively. The function map is defined as in the semi-lattice case \( M(n) = f_n \). Since the lattice is the powerset of the set of flow graph nodes its height is \(|N|\). Thus, applying Algorithm 2 gives an \( O(|N|^3) \) solution procedure for the problem. The bound of this algorithm is equal to the bound of algorithms developed by Callahan [8] and Grunwald [32] for the statement precedence problem.
3.5 Summary

We have extended the theoretical foundations of data flow analysis to accommodate a number of natural analysis problems for concurrent programs. The formulation of complete-lattice frameworks is compatible with previous semi-lattice frameworks. Thus, a general implementation for the complete-lattice frameworks, such as Algorithm 2, is applicable to semi-lattice frameworks as well. We have illustrated, by way of example, the ease with which analysis problems for concurrent programs can be formulated as a complete-lattice framework. On the theoretical front we intend to look at the extent to which distributivity, continuity and rapidity [51] can be exploited in complete-lattice frameworks. We intend to investigate a wide variety of standard and special purpose data flow analysis problems to understand the extent to which they are accommodated in the new theory. Given that complete-lattice frameworks promise improved precision for data flow analysis of concurrent programs, we intend to empirically investigate the extent to which this occurs in practice.
CHAPTER 4
FLAVERS

The application of 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. Along with increased performance, incorporation of concurrency in software greatly complicates the problem of reasoning about the correctness of an application. To combat this, developers require cost-effective analysis techniques to gain confidence in the quality of their concurrent software.

In this chapter, we present an analysis approach, called FLAVERS, that is based on data flow analysis and has the potential to provide cost-effective analysis of concurrent programs with respect to explicitly stated correctness properties.

Although FLAVERS is applicable to a wide range of concurrency and communication models, in this dissertation we restrict our discussion to 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 choose whether the analysis should attempt 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 imprecision of static analysis, we have developed a range of techniques for improving the precision of the results. Prior to analysis,
refinements to the flow graph representation of the program, which are based on program and property specific information, increase the efficiency and precision 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 precision of the results. After analysis is performed we can combine the results of independent analysis runs to improve the overall precision of the combined analysis results. One strength of our approach is the flexibility allowed in choosing and combining these techniques so as to increase precision without making analysis time impractical.

The next section 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 in Section 4.2. Following that, Sections 4.3 through 4.6 describe the basic analysis components and techniques for improving the precision and efficiency of the basic approach.

4.1 Background

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 [39, 70, 86]. Complexity results for reachability analysis [74] imply that, in general, the size of the program state space and consequently the cost of analysis increases rapidly 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 [23, 28, 79], building and analyzing the state space compositionally [14, 84], and using a symbolic representation of the state space[7]. Each of these techniques has been successfully applied to selected programs to increase the size of programs that can be analyzed.
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 [4] and sufficient [57] conditions. The necessary condition technique has been successfully applied to a number of programs and allows for practical analysis of very large versions of those programs.

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 [51]. For data flow analysis of sequential programs, it is often the case that near linear-time analysis algorithms are applicable; this is due to the nature of structured flow graphs for sequential programs.

Data flow analysis of concurrent programs requires that inter-task communication be represented. Early model checking approaches [11] 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 [8, 32, 52] or label nodes representing communication statements so that they can be matched during analysis [9, 54]. 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 [53], although the resulting flow graphs are invariably irreducible. Our approach also employs a flow graph that represents communication explicitly.

\(^1\)The question *do all executions satisfy the property?* is the dual of *is it false that no executions satisfy the negation of the property?*.
Masticola and Ryder [52] describe an analysis approach for checking deadlock freedom in Ada tasking programs that uses data flow analyses [53] to improve the precision of the analysis results. These data flow analyses are in the spirit of the refinements described in Section 4.6.1. 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 precision through the use of feasibility constraints, and support analysis of a rich class of properties as opposed to just deadlock.

Olender and Osterweil [61] have developed a necessary conditions based analysis for sequential programs that uses a simple version of the state propagation algorithm described in Section 4.5. In addition to the increase in complexity associated with the analysis of concurrent programs, our approach incorporates a variety of mechanisms to increase the precision 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 automata [28, 47, 60]. At a high level, the set of executable program paths are represented as strings generated by the program automaton and the set of program paths that satisfy the property are strings accepted by the 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 4.5 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 [64] and regular expression based formalisms [4, 47, 60]. 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, we will analyze properties 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 [82].

4.2 Overview

FLAVERS is an architecture and collection of analysis components that enables the construction of static analyzers. These analyzers can be tailored to the program and the class of execution behaviors being analyzed. The resulting analyzer provides information that can be used to determine whether a program exhibits a specified execution behavior. The flexible composition of analysis components allows tailoring of analyses to make them more cost-effective for a given analysis problem; this includes both reducing the cost of performing analysis and increasing the precision of analysis results. Figure 4.1 provides a high-level view of the major components and connections in the analysis architecture. Rectangular boxes represent basic analysis components. Ellipses represent optional components that are used to tailor the
analysis. Incorporation of information from these optional components is depicted by \(\oplus\).

All FLAVERS analyzers perform the basic analysis steps of:

- Extracting a model of executable behavior from a textual description of the program.
- Constructing a representation of the class of execution behaviors to be checked.
- Running the state propagation algorithm with the model and behavior representation as inputs.
- Presenting the results of that algorithm to the user.

The analysis results may demonstrate conclusively that the behavior to be checked 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 desired behaviors 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 optional analysis components can be helpful.

There are two kinds of optional analysis components: feasibility constraints and refinements. Both are designed to increase the precision of analysis results. The user can select components that incorporate partial information about the set of feasible program executions. Components can be added one at a time allowing for an incremental growth in the amount of program information on which analysis is based. This flexibility in composing components means that we need only add as much information as is necessary for isolating a program fault or eliminating imprecision for a given analysis problem.
A View of Program Execution

Users may wish to reason about the execution of an application from a variety of perspectives. For example, we may be interested in determining whether some statement in a program uses a variable whose value is uninitialized; in this case we are interested in the definitions and uses of program variables. We may be interested in determining that whenever we call a read or write operation on a file abstraction that we have previously opened the file and will subsequently close the file; in this case we are interested in calls on the operations of a file abstraction. The information users are interested in can vary from fine sub-statement level details to course super-statement details. To address this variation, FLAVERS allows users to define a set of program activities as program events of interest; these events will be used both in the construction of the model of executable program behavior and in specifying patterns of behavior to be reasoned about. FLAVERS assumes that all events are atomic; non-atomic activities can be modeled by a pair of start and end events. These events are collected into an alphabet that fixes the scope of reasoning to be performed.

Definition 11 An event alphabet, \( \Sigma = \{ \tau, a, b, c, \ldots \} \), is a set of symbols, where each symbol, \( a, \ldots \), in \( \Sigma \) represents a unique program event.

The symbol \( \tau \) is an element of all event alphabets; it stands for the occurrence of an unmodeled program event. The user defines a mapping, \( \text{Events: stmts} \rightarrow \Sigma^* \), from program statements to sequences of event symbols. There are a variety of mechanisms for defining this mapping, including source code annotations, having a fixed set of events that are automatically mapped to a given program, or using a special purpose language for describing classes of mappings. FLAVERS uses a combination of these techniques. All inter-task communication events are implicitly included in \( \Sigma \). Additional symbols, \( \text{sym}_i \), are defined by adding a comment of the form "\( -- \text{QREa}[\text{sym}_1, \text{sym}_2, \ldots] \)" to the end of the line of the statement to be modeled.
Such a comment indicates that the statement at which the comment occurs should be modeled as the sequence of events $sym_1, sym_2, \ldots$.

**Modeling Program Execution**

In order to reason about program execution behavior we need a semantically well-founded model of that behavior. This model need not represent all details of program execution; it must, however, represent sufficient information to support the desired analyses. FLAVERS' conservative analyses imply that this model must represent all sequences of program events, drawn from $\Sigma$, that correspond to feasible program executions. In a sense this model *overestimates* the executable behavior of the program; in doing so, it may include sequences of program events that do not correspond to actual program executions.

**Specifying Patterns of Execution Behavior**

Our program model represents sequences of program events. It seems natural, therefore, to specify the execution behavior we wish to reason about, also called a *property*, as sequences of that same set of events. A collection of such sequences is specified as a regular language over $\Sigma$. In practice, regular expressions and finite state automata are used. 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 specified using these formalisms. In addition to specifying a *pattern* of events, we need to indicate whether to check that this pattern characterizes all program executions or, alternately, whether it characterizes no program execution.
State Propagation Analysis

The execution behavior of the program and the execution behavior that we want to reason about are both expressed as finite sequences of events over Σ. We can easily construct a finite-state acceptor for the language of the property to be checked. One approach to reasoning about whether the specified property is consistent with the executions of the program is to enumerate the event sequences of the program and submit them to the acceptor. If all such sequences are accepted then the property holds for all program executions. If no such sequence is accepted then the property holds on no program execution. This approach is impractical. While the sequences are finite they may be arbitrarily long and consequently there are infinitely many of them. Restricting consideration to fixed length sequences doesn't help much as the nature of concurrent execution implies that, in general, the number of executions will grow exponentially with the number of tasks in the program.

FLAVERS casts the analysis question of whether program behavior is consistent with a specified property as a data flow analysis problem called state propagation analysis. We have adapted this algorithm to apply it to concurrent programs and have enhanced it to improve its accuracy. This algorithm avoids enumerating sequences of program events by collapsing them into equivalence classes based on the structure of the property being analyzed. The result is an efficient algorithm 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 that guides the user to the source of program faults or imprecision in the analysis.

Increasing Precision and Reducing Cost

The state propagation algorithm trades a reduction in precision for a reduction in the cost of analysis. The result, as shown in Section 4.5, is an analysis that grows as a
polynomial in the number of program statements. The initial tradeoff is made prior to consideration of a specific analysis problem. A distinguishing feature of FLAVERS, and one of its major strengths, is the ability to modify this tradeoff. This allows FLAVERS to be tailored to the program and property under analysis.

There are three kinds of mechanisms for controlling this tradeoff:

**Refining the Model** The basic model of program execution contains many event sequences that do not correspond to actual program executions. Through preliminary analysis of the basic model, we can remove some (sub)sequences of program events that do not correspond to actual program executions.

In addition, the basic model represents all program events in $\Sigma$. We may be interested in reasoning about a property specified over a subset of those events, in this case the other events represented by the model are irrelevant and can be removed.

**Constraining the Algorithm** To gain efficiency the basic state propagation algorithm does not consider all of the statically determinable information about which event sequences are executable. We can selectively incorporate partial information about the set of executable event sequences. This allows the state propagation algorithm to avoid considering some sequences of events that do not correspond to actual program executions.

**Combining separate analyses** Analyses of different properties for the same program are performed independently, but the information they provide can be combined in a variety of ways. As a simple example, we can consider conclusive analysis results for a pair of properties as equivalent to a conclusive result for the conjunction of the two.

In the following sections we discuss each of the basic and optional analysis components in greater detail. While the FLAVERS approach is language independent we
present the details of the approach as it applies to concurrent programs written in Ada. We refer to this instance of the FLAVERS architecture as FLAVERS/Ada.

4.3 A Model of Program Executions

All static analysis techniques, including FLAVERS, reason about a model of program execution. The model captures relevant program information to present a precise and practical representation of the execution behavior of the program.

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. Furthermore, analysis over non-finite-state models is impractical. In order to enable analysis of complex software we construct a finite-state model of program execution behavior in such a way that information that is relevant to a desired class of analysis is preserved in the model and such that the model is tractable. An information preserving model allows us to construct an analysis that can produce precise analysis results. A tractable model gives us the opportunity to perform analysis in a practical amount of time.

The analysis embodied in FLAVERS is truth-preserving with respect to checking a pattern of behavior over all program executions and falseness-preserving with respect to checking a pattern of behavior over some program execution. FLAVERS supports such conservative analyses by reasoning using a model of program execution that represents all possible executable sequences of program events; we note that this model may represent some sequences of program events that do not correspond to possible program executions.

In this section, we present the trace flow graph (TFG) model for concurrent software. We demonstrate that the model is small and easy to construct, that it incorporates information to support a wide variety of analyses and that it is suitable for the conservative analyses described in Section 4.5.
4.3.1 Trace Flow Graph

A control flow graph (CFG) for a task represents each statement in the task and the possible execution orderings of those statements with respect to one another. It is a directed graph with distinguished initial and final nodes, representing the first and last statements of the task respectively. Standard compiler techniques can be used to construct a collection of CFGs for the tasks of a program. The TFG can be viewed as a forest of control flow graphs (CFG), one for each task in the program with additional nodes and edges used to represent inter-task synchronization and communication. Formally,

**Definition 12** A *trace flow graph* is a labeled directed graph with distinguished entry and exit nodes, $(N, E, n_{\text{initial}}, n_{\text{final}}, L)$, where:

\[
\begin{align*}
N & = \{n_1, n_2, \ldots, n_k\} \\
E & \subseteq \{(n_i, n_j) | n_i \in N \land n_j \in N\} \\
n_{\text{initial}}, n_{\text{final}} & \in N \\
L & : N \rightarrow \Sigma
\end{align*}
\]

A node $n \in N$ represents an instance of some program event that is captured in the model. The labeling function $L$ maps a node to the associated program event symbol; $L$ is many-to-one. Distinguished nodes $n_{\text{initial}}$ and $n_{\text{final}}$ represent the initial and final program events. Nodes have a number of attributes that provide additional information:

**kind** There are two kinds of nodes: LOCAL and COM. LOCAL nodes represent events that are internal to a single task. COM nodes represent events that are shared by a number of tasks. Of the LOCAL nodes we distinguish LAST, SEND and RECEIVE nodes; these correspond to local statements that represent a τ
labeled node with a non-\( \tau \) labeled successor and requests for communication. Of the COM nodes we distinguish INITIAL, FINAL, and COM_END. The first two correspond to implicit synchronizations of all tasks at program activation and termination, respectively. The COM_END is used to represent the end of a compound communication interaction. An example of a compound communication is a remote procedure call.

**task** It is often convenient consider only the nodes and edges that are related to a particular program task. For example, we can formulate a problem over a task’s CFG by eliding all nodes and edges not in the task.

**cfg-node** Relating a TFG node back to the CFG from which it is derived allows us to ultimately map a TFG node back to the source code for the program.

**partner** For compound communications we indicate the pair of COM and COM_END nodes as partners of each other.

Edges \( E \) represent possible execution orderings of program events. An edge \((n_i, n_j)\) represents the fact that the program event \( L(n_i) \) may precede event \( L(n_j) \); the direction of the edge indicates the order of the events. In the TFG there are two kinds of edges; control flow (CF) and may immediately precede (MIP) edges. Control flow edges represent intra-task orderings of program events; they are analogous to edges in a CFG. May immediately precede edges represent inter-task orderings of program events; for nodes in different tasks, \( n_i \in Task_1 \) and \( n_j \in Task_2 \), we have a MIP edge \((n_i, n_j)\) to represent the fact that in some program execution, events \( L(n_i) \) and \( L(n_j) \) may be consecutive. As with nodes, edges have a number of additional attributes that provide information that is useful in analysis.

**kind** There are two kinds of edges, CF and MIP.
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; -- QREa[work1]
      Semaphore.V;
    end loop;
  end Client1;

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

Figure 4.2 Ada Tasks for Semaphore Example

task It is often convenient elide all nodes and edges that are not related to a particular program task. For example, we can formulate a problem over a tasks CFG by eliding nodes and edges not in the task.

An Example

In this section we describe an example and illustrate the TFG constructed for it; the example is necessarily simple as TFGs quickly grow beyond a size that is conducive to visualization.

The example in Figure 4.2 consists of a task that operates as a semaphore; it presents two communication entries P and V. The state of semaphore is maintained by the control state of the task. 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.

Nodes

TFG nodes represent program events that we are interested in modeling. We can also view a node as representing the local control state of a task either just before or after execution of that event. For the semaphore example there are four
program events of interest: Semaphore.P, Semaphore.V, work1, and work2. The TFG for the example is illustrated in Figure 4.3 with MIP edges elided to make the figure more readable. Diamonds represent program events that are performed by more than one task jointly; they represent communication, synchronization, task activation and termination. Ellipses represent task start and end events. Rectangles represent program events that are local to a task.

Every TFG has designated initial and final nodes; in the semaphore example they are diamonds 25 and 26. These are points at which all tasks in the system synchronize their execution.

The other four diamonds in our example represent instances of inter-task communication. An instance of a communication event is where a pair of send and receive  

\footnote{All other TFG nodes will have a \( \tau \) label}
statements execute to engage in communication over a common channel\textsuperscript{3}. In general, the number of communication nodes related to a channel is the product of the number of send and receive statements over that channel. In our example, nodes 21 and 23 are instances of the \texttt{Semaphore.P} communication event.

An explicit representation of each communication instance provides the opportunity to perform a number of optimizations with respect to the size and precision of the TFG. For example we can specialize the representation of a communication based on the participating send and receive statements. If a send passes a constant parameter to the receiving task we can use the value of that parameter to statically refine its representation by performing a kind of inter-procedural constant propagation.

TFG nodes can represent events that are local to a single task. In the semaphore example, the event \texttt{work1} labels node \texttt{client1:2}. A TFG can include $\tau$ (in the figure \texttt{tau}) labeled nodes that do not contribute to representation of program executions; they are used to maintain certain desirable structural properties of the TFG. For example, node 8 is the loop header node for task \texttt{client1}.

We note that each node has a single event that labels it, however, the same event may label many nodes.

Edges

A TFG edge represents an ordering on a pair of instances of program events; the event labeling the source node immediately precedes the event labeling the destination node. These orderings can arise from a number of sources including: the order of statements in the program, synchronization of independent tasks, and asynchronous execution of independent events in independent tasks.

The ordering of statements in the program is reflected in the structure of the CFGs for the tasks in the program; the edges of a CFG represent these orderings. In the

\textsuperscript{3}We use both the general term \textit{channel} and the Ada specific term \textit{entry} to refer to points at which tasks may interact.
TFG, we have control flow edges that capture ordering information that is equivalent to statement ordering. Since the TFG represents program events and those events can model sub/super/statement-level program activity, TFG control flow edges capture event orderings. In addition, there may be many program statements that are not modeled by program events and for those any statement ordering information is ignored. In Figure 4.3 we illustrate the TFG for the semaphore example with only CF edges shown. The standard control structures, sequencing, branching, and looping, are represented. Event sequences are illustrated by Semaphore.P immediately preceding work1 by virtue of the edge (21, 12). Branching structures are illustrated by outgoing edges of node Client1: 9. Control flow loops are illustrated by 8 → 9 → 10 → 21 → 12 → 13 → 22 → 8, where each of these nodes is local to task 1.

Note that for each communication node there are two incoming and two outgoing edges; for each task participating in the communication there is a single incoming and outgoing control flow edge.

In some cases, detailed ordering information that is necessary for execution of a concurrent program, but is well below the programming language abstraction, is not represented in the TFG. When multiple tasks synchronize execution, whether at program activation, termination, or at a synchronous communication statement, we model the simultaneity of this activity by a COM node. We do not represent send and receive communication statements explicitly in a TFG and the execution ordering of these statements is ignored. In Figure 4.3, nodes Semaphore:6 and Client2:20 are placeholders for accept and call statements over the V entry; these statements have been combined into a representation of their simultaneous execution in the form of the Semaphore.V event labeling node 24.

Unlike other concurrent flow graph representations, TFGs explicitly capture the potential interleaving of pairs of asynchronously executing program events. We rep-
resent pairs of nodes that may interleave execution order by introducing additional edges into the TFG. A control flow edge indicates that the event labeling the source of the edge may immediately precede the event labeling the destination of the edge; a \textit{may immediately precede} (MIP) edge is used to express the analogous inter-task event ordering. An overly pessimistic set of MIP edges contains, for each node that is not labeled \( \tau \), a MIP edge from that node to all other nodes. This would clearly represent all feasible inter-task orderings. However, a few simple observations allow us to significantly reduce the number of MIP edges and consequently improve the precision of subsequent analysis.

1. MIP edges between nodes in the same task are unnecessary, because all event orderings local to a task are captured by CF edges.

2. If we have a sequence of \( \tau \) labeled nodes then we need only construct MIP edges whose destination is the final node in the sequence. This is an optimization based on the fact that we are never interested in reasoning about sequences of \( \tau \)s.

The first is true because CF edges already capture the necessary intra-task orderings. The second is true because such a sequence is equivalent to a single \( \tau \) labeled node in terms of its ability to represent sequences of program events, \( \tau \rightarrow \tau = \tau \).

Applying these principles reduces the number of MIP edges required in the TFG for the semaphore example from 676, for the overly pessimistic set of edges, to 42. To illustrate MIP edges we show those that originate at node 19 in Figure 4.4. Note that there are no MIP edges to other nodes in task \texttt{Client2}. Furthermore, in task \texttt{Client1} only the final nodes in the sequences of \( \tau \) nodes \( 8 \rightarrow 9 \rightarrow 10 \) and \( 8 \rightarrow 9 \rightarrow 11 \) have incident MIP edges. Using the principles we are able to reduce the number of MIP edges required to make the TFG a conservative representation for our analyses.
4.3.2 Constructing A TFG

A TFG is constructed from a collection of CFGs that represent the tasks in the program. These CFGs must represent all of the program events that we are interested in reasoning about. We define a function, \( GetEvents : N \rightarrow e_1, e_2, \ldots e_k \), where \( e_i \in \Sigma \), that produces the sequence of program events of interest that represents the given CFG node \(^4\). This function is built from the user defined statement to event mapping information. The construction algorithm is as follows:

**Algorithm 3 (TFG Construction)**

*Input:*

- a set of task CFGs

---

\(^4\)We simplify the presentation here, but in our complete algorithm an analogous function \( GetEvents : E \rightarrow e_1, e_2, \ldots e_k \) is defined for CFG edges and the construction algorithm is modified to process edge related events.
Output:

a TFG

The algorithm is broken into three phases. The first, in Figure 4.5, phase creates local nodes and control flow edges. The second, in Figure 4.6, phase creates communication nodes and edges. The third, in Figure 4.7, phase creates MIP edges.

for each CFG
    do a depth-first walk of the CFG
        call current CFG node n
        if GetEvents(n) ≠ ∅ then
            -- Build up a sequence of TFG nodes to represent the CFG node
            GetEvents(n) is (e₁,e₂,...,eₖ)
            build a sequence of TFG nodes connected by TFG edges
            (n₁,n₂,...,nₖ) such that for i ∈ 1...k, L(nᵢ) = eᵢ and (nᵢ,nᵢ₊₁) ∈ E

            -- Build edges headed TO the representation of this CFG node
            for each TFG node, i, from which an edge to TFG node for n is desired loop
                Create TFG edge (i,nᵢ)
            end loop

            -- Build edges headed FROM the representation of this CFG node
            for each outgoing CFG edge of n, (n,m) ∈ E_c loop
                if TFG nodes, (m₁,m₂,...), for CFG node m have not been created then
                    record need for TFG edge from nᵢ to TFG nodes for m
                else
                    Create TFG edge (nᵢ,mᵢ)
                end if
            end loop
        else -- no events to be modeled from CFG node n
            if there are edges, (p,n₁) to be created for TFG nodes of n then
                for each CFG successor, s, of n loop
                    record need for TFG edge (p,s)
                end loop
            end if
        end if
    end depth-first traversal
end loop

Figure 4.5 Phase One of TFG Construction
for each TFG node, \( n \), where \( \text{kind}(n) = \text{CALL} \) loop
for each TFG node, \( m \), where \( \text{kind}(m) = \text{RCV} \) and \( L(n) = L(m) \) loop
   -- if the RCV has a body \( \text{partner}(m) \) is its end
      if \( \text{partner}(m) \neq \emptyset \) then
         Create TFG node \( \text{com}_{\text{start}} \)
         Create TFG node \( \text{com}_{\text{end}} \)
   -- there exists a unique edge \( (m, x) \) to start of RCV body
   -- there exists a unique edge \( (y, \text{partner}(m)) \) from end of RCV body
         Create TFG edge \( (\text{com}_{\text{start}}, x) \)
         Create TFG edge \( (y, \text{com}_{\text{end}}) \)
         Create TFG edges \( (n, \text{com}_{\text{start}}), (m, \text{com}_{\text{start}}) \)
         replace TFG edge \( (n, x) \) with \( (\text{com}_{\text{end}}, x) \)
         replace TFG edge \( (\text{partner}(m), y) \) with \( (\text{com}_{\text{end}}, y) \)
      else
         Create TFG node \( \text{com} \)
         Create TFG edges \( (n, \text{com}), (m, \text{com}) \)
         replace TFG edge \( (n, x) \) with \( (\text{com}, x) \)
         replace TFG edge \( (m, y) \) with \( (\text{com}, y) \)
      end if
   end loop
end loop

Figure 4.6 Phase Two of TFG Construction

4.3.3 Conservativeness of the TFG Model

In this section, we prove that the TFG represents all program executions over a specified set of program events.

We can construct a finite-state automaton that models all control and data states that a given program may enter \(^5\). To do so we may have to fold the potentially infinite set of program data states into a finite representation. Such an automaton will include a state for all reachable global program states and transitions on each state for each possible next global program state; such transitions represent the execution of a program statement. Paths in this automaton represent possible system executions as a sequence of program statements. We define a computation automaton (CA) as:

\(^5\)We are considering a fixed configuration of the system.
for each TFG node, $n$, loop 
    for each TFG node, $m$, loop 
        if $(m,n) \notin E$ and $(n,m) \notin E$ and $m \notin \text{Task}(n)$ and 
            $(L(m) \neq \tau$ or $\text{kind}(m) = \text{LAST})$ then 
                Create TFG MIP edges $(m,n)$, $(n,m)$ 
        end if 
    end loop 
end loop

Figure 4.7 Phase Three of TFG Construction

Definition 13 A computation automaton is a deterministic finite state automaton $(S, Q, \delta, s_{\text{start}}, A)$. Elements of $S$ represent global states of the program and are represented as tuples $(d_1, d_2, \ldots, d_m, t_1, t_2, \ldots, t_n)$ with $m$ components of data state and $n$ components of control state information. $s_{\text{start}} \in S$ is a designated start state; it corresponds to the initial program state. $F \subseteq S$ are the set of designated accepting states; these correspond to terminal program states. $A$ is an alphabet of events; these correspond to program statements. $\delta: (S, a) \rightarrow S$ is the state transition relation; each transition represents the occurrence of a program event that causes some component(s) of the current state to change value.

We define a number of relationships between components of the CA and TFG.

In Section 4.2 we defined a map from program statements onto the program events that we are interested in reasoning about. Using this map we can define the events associated with a CA transition and extend that definition to paths in the CA. For each $a \in A$ we define $\text{Event}(a) \in \Sigma^*$. For each CA path $p = s_1 \rightarrow s_2 \rightarrow \ldots s_k$, $\text{String}(p) = \text{Event}(s_1 \rightarrow s_2)\text{Event}(s_2 \rightarrow s_3)\ldots\text{Event}(s_{k-1} \rightarrow s_k)$.

As mentioned above CA states have a control component for each task in the system; each such control component corresponds to a TFG node. Each control component of a state, $t_i$, represents a locus of control in a single task in the program. Similarly, each non-communication node, $n$, in the TFG correspond to a locus of control in a single task in the program.
Definition 14 The control component node map, $\text{Node}(t_i)$, is defined as $\text{Node}(t_i) = e$ where $e$ is a TFG node and $t_i$ is the control state of the task containing $e$ after the event represented by $e$ executes.

States of the CA are related to collections of nodes in the TFG.

Definition 15 The state node map, $\text{Nodes}(s)$, is defined for each $s \in S$ as $\text{Nodes}(s) = \{ n \in N : n = \text{Node}(t_i) \text{ for some } t_i \text{ in } s \}$.

We can now state and prove the main theorem of this section. It states that a path in the CA is equivalent to a collection of paths in the TFG.

Theorem 4 (Equivalence of CA paths and collections of TFG paths)

For each path, $p = s_{\text{start}} \overset{s}{\rightarrow} s$, in the CA, for each node, $n \in \text{Nodes}(s)$, corresponding to the final state on that path, there exists a path, $p' = n_{\text{start}} \overset{*}{\rightarrow} n$, in the TFG from the start node to that node, such that the two paths correspond to identical strings over the program events of interest, $\text{String}(p) = \text{String}(p')$.

Proof: By induction on the length of a CA path.

Base Step: Length 0

A path, $p$, of length zero in the CA starts at $s_{\text{start}}$ and ends at $s_{\text{start}}$. Therefore we need to consider each $n \in \text{Nodes}(s_{\text{start}})$.

$\text{Node}(t_i)$ is defined to be the node after whose execution task $i$ will be in local control state $t_i$; thus $\text{Node}(s_{\text{start}}) = \{ n_{\text{initial}} \}$.

Since $\text{String}(p) = \epsilon$ we must produce a path $p' = n_{\text{initial}} \overset{*}{\rightarrow} n_{\text{initial}}$; such a path exists, trivially, and furthermore

$\text{String}(p') = \text{String}(\tau) = \epsilon$.

Inductive Step: Length $l - 1$
Assume that the theorem holds for all paths of length \( l - 1 \).

Choose any path of length \( l \), \( p = s_{\text{start}} \rightarrow s_{l-1} \rightarrow s_l \), in the CA.

There are two cases to consider; the last transition \( s_{l-1} \rightarrow s_l \) corresponds

1. to an event local to a task

2. to an joint event shared by a set of tasks

**Case 1:**

The program statement, \( stmt \), that caused the transition is local to some task, \( i \). In this case, we have:

\[
s_{l-1} = (d_1, d_2, \ldots, d_m, t_1, t_2, \ldots, t_l, \ldots, t_n)
\]

and

\[
s_l = (d_1, d_2, \ldots, d_m', t_1, t_2, \ldots, t_l', \ldots, t_n)
\]

where \( t_i \) is the control state of task \( i \) before execution of \( stmt \) and \( t_i' \) is the control state of task \( i \) afterwards. We allow for the possibility that \( stmt \) can change the value of some data state component as well, \( d_m \) becomes \( d_m' \).

We know that the theorem holds for the prefix of length \( l - 1 \). For our length \( l \) path we can reuse the collection of TFG paths that satisfied the inductive hypothesis.

Each of these TFG paths is extended:

**in task** \( i \) by a control flow path, \( p_{cf} = Node(t_i) \rightarrow n_{cf} \), that is local to task \( i \) that passes through all and only the program events modeling \( stmt \).

**in task** \( j \neq i \) by a path, \( Node(t_j) \rightarrow p_{cf} \rightarrow Node(t_j) \), that links the local control flow path in task \( i \) with MIP edges to the node in task \( j \).
Case 2:

The program statement, $stmt$, that caused the transition is shared by a set of tasks $i, i+1, \ldots, j$. In this case, we have $s_{l-1} = (d_1, d_2, \ldots, d_m, t_1, t_2, \ldots, t_i, \ldots, t_n)$ and $s_l = (d_1, d_2, \ldots, d_m', t_1, t_2, \ldots, t_i', t_{i+1}', \ldots, t_j', \ldots, t_n)$, where $t_i$ is the control state of task $i$ before execution of $stmt$ and $t_i'$ is the control state of task $i$ afterwards.

We allow for the possibility that $stmt$ can change the value of some data state component as well, $d_m$ becomes $d_m'$.

We know that the theorem holds for the prefix of length $l - 1$. For our length $l$ path we can reuse the collection of TFG paths that satisfied the inductive hypothesis.

Each of these TFG paths is extended:

- **in tasks** $i, i+1, \ldots, j$ by a control flow path, $p_{cf} = Node(t_i) \rightarrow n_{joint}$, that is local to task $i$ that passes through all and only the program events modeling $stmt$ and has a final node labeled with the joint event.

- **in task** $k < i$ or $k > j$ by paths, for each path $p_{cf} = Node(t_k) \xrightarrow{MIP} p_{cf} \xrightarrow{MIP} Node(t_k)$, that link the local control flow paths in task $i, i+1, \ldots j$ with MIP edges to the node in task $k$.

The path equivalence theorem implies that each path in the CA is represented in the TFG; in general, the reverse is not true. There exist collections of paths in the
TFG that do not correspond to the CA. We can construct the CA to be as precise a conservative representation of all feasible program executions as is statically possible. Since the TFG is conservative with respect to CA paths, then it is conservative with respect to all feasible executions.

4.3.4 Complexity of the TFG Model

In this 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 send and receive statements. The TFG, however, is dependent on the designated set of program events. The number of events we are interested in for a given program may vary widely depending on the patterns of execution behavior 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 5 (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 number of events per program statement, \( S \) the number of statements, \( C \) the number of communication send and \( R \) the number of communication receive statements. The number of nodes in the TFG is \( O(2 + c(S - (C + R)) + c(C \times R)) \). This is the sum of the initial/final nodes, the nodes from modeled CFG nodes and the COM 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 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 this upper bound on the number of nodes is weak. The product of send and receives is more accurately computed on a per channel basis and then summed over the set of channels; this value can be significantly less than the product of all send and receive statements. In addition, communication statements are typically a small percentage of statements in a given program. We expect that the number of TFG nodes is much closer to linear in the number of statements in practice. We also note that the number of TFG edges is dominated by the number of MIP edges in the graph. In Chapter 5 we provide empirical evidence that, in our experience, the number of nodes and edges in TFG do not begin to approach these worst case bounds.

**Theorem 6 (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:*

We reason about each phase of the algorithm separately and sum the results.

Phase 1:

For each CFG we perform a depth-first walk of the nodes in the CFG resulting in a factor of $S$. For each such node we construct the set of events modeling the node, resulting in a factor of $c$, then we process each successor node in the CFG, in the worst-case when there is a single CFG and a fully connected graph there are $S$ successors. Thus, phase 1 requires $O(S^2)$ operations.
Phase 2:

For each pair of call and receive statements we perform at most 7 operations. In the worst-case the number of calls is $C = S/2$ and the number of receives is $R = S/2$. Thus, phase 2 requires $O(7RC) = O(S^2)$ operations.

Phase 3:

For each pair of TFG nodes we potentially create a MIP edge. From Theorem 5 the number of TFG nodes is $O(S^3)$. Thus, phase 3 requires $O(S^4)$ operations.

Clearly, the worst-case cost of constructing a TFG is dominated by phase 3 and the bound on the execution time of the construction algorithm is $O(S^4)$.

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 class of execution behaviors. Furthermore, as mentioned above, in practice communication statements are a small minority of program statements. Together these factors frequently make the cost of constructing a TFG sub-quadratic in the number of program statements. In Chapter 5 we provide empirical evidence to support this claim.

4.4 Describing Executable Behavior

All automated analysis techniques, including FLAVERS, require a specification of intended behavior to which the executable behavior of a program is compared. In some analysis approaches the intended behavior is implicit, as in many analyses
that attempt to demonstrate freedom from deadlock. In other approaches, including FLAVERS, the intended behavior is specified explicitly by a user. While it is possible to completely specify the intended behavior of a program, in practice, it is often more desirable write small specifications of important properties of the program. Large monolithic specifications may be as difficult to reason about as the implementation itself. With an analysis approach like FLAVERS, breaking a large specification into small self-contained pieces allows the analysis to be tailored for each piece in turn. This can effectively reduce the amount of information required for producing precise analysis results. FLAVERS supports specification of and reasoning about such self-contained properties of concurrent systems as finite-state automata.

Olender and Osterweil developed the Cecil formalism for describing event sequencing constraints of sequential programs. Cecil introduces anchored quantified regular expressions (AQRE)s to specify regular expressions over (sub) control flow graph representations of a program. We adopt quantified regular expressions (QRE) for our purposes. We use standard algorithms for converting a QRE into a finite state automaton and reducing the size of that automaton; this property automaton (PA) is used as input to the state propagation algorithm. In this section, we define the syntax and semantics of QREs, we describe a syntactic style for writing QREs that makes writing specifications easier, and define the PA and its construction.

4.4.1 Syntax of QREs

Figure 4.8 gives the syntax for QREs. For convenience we have defined a few higher-level operators in the syntax. In particular, the "[-...]

---

6 An implementation may be considered such a specification.

7 Anchors serve to specify a sub-graph of the program CFG and are a means of restricting the scope of analysis, potentially increasing the precision and reducing the cost of analysis. For conservative analysis, anchoring requires that the start anchor be reachable from the program start state. In a sequential CFG this is a reasonable assumption and we can gain additional useful information from anchoring a QRE. In concurrent flow graph the assumption that a syntactic state is reachable is not reasonable. In fact, the number of reachable states is often a small minority of the number of syntactic states.
id : [a-zA-Z][a-zA-Z0-9_:=<]*
pos : [1-9][0-9]*
ALL : "all"
NONE : "none"

qre : alphabet quantifier expression

alphabet : "{" idlist "}"

quantifier : ALL | NONE

equation : |
  .
  id
  ":[" "-" idlist "]"  
  ":[" idlist "]"  
  expression;expression  
  expression ";" expression  
  "(" expression ")"  
  expression ";" pos  
  expression ";"  
  expression ";"  
  expression

idlist : id
  id "," idlist

Figure 4.8 QRE Syntax

describing sub-strings that exclude specified events; this operator is used extensively in the QRE style described below.

4.4.2 Semantics of QREs

There are three components to a QRE: the alphabet, quantifier and regular expression. The alphabet only establishes a set of symbols for the expression and has no semantics. We first consider the expression part of a QRE. This is a standard regular expression over the set of symbols in the specified alphabet, which denoted \( \Sigma_{property} \). Figure 4.9 defines the semantics of an expression in terms of other expressions or the
\( \mathcal{L}(\text{tau}) = \{\} \)
\( \mathcal{L}(\text{x}) = \{x\} \)
\( \mathcal{L}(\text{.}) = \{x | x \in \Sigma_{\text{property}}\} \)
\( \mathcal{L}([-\text{idlist}]) = \{x | x \in \Sigma_{\text{property}} \setminus \{\text{idlist}\}\} \)
\( \mathcal{L}([\text{idlist}]) = \{x | x \in \{\text{idlist}\}\} \)
\( \mathcal{L}(e_1;e_2) = \{xy | x \in \mathcal{L}(e_1) \land y \in \mathcal{L}(e_2)\} \)
\( \mathcal{L}(e_1|e_2) = \{x | x \in \mathcal{L}(e_1) \lor x \in \mathcal{L}(e_2)\} \)
\( \mathcal{L}((e)) = \mathcal{L}(e) \)
\( \mathcal{L}(e^*k) = \mathcal{L}(e)_1;\mathcal{L}(e)_2;\ldots;\mathcal{L}(e)_k \)
\( \mathcal{L}(e^*) = \{\} \cup \bigcup_{i=1}^{\infty} \mathcal{L}(e^{*i}) \)
\( \mathcal{L}(e+) = \bigcup_{i=1}^{\infty} \mathcal{L}(e^{*i}) \)
\( \mathcal{L}(e?) = \mathcal{L}(e) \cup \{\} \)

**Figure 4.9 QRE Semantics**

language of symbols that it describes. 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 a 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 an empty language intersection test.

### 4.4.3 A QRE Style

QREs allow for patterns of events to be described as arbitrary regular expressions. We have found that by writing QREs in a particular style they are both easier to read and write. The style consists of expressions constructed out of two types of sub-expressions: intervals that exclude a set of events, of the form \([-\text{a}]\), and required events, of the form \(\text{a}\). The idea of intervals that require and exclude events is derived from Corbett’s \(\omega\)-starless expressions [15].

QREs in this style begin with an outer excluding interval that is iterated; this serves to guard against unintended entry into the pattern. The initial interval is followed by an iterated sequence of alternating required and excluding intervals; in-
tuitively, this alternating sequence captures the pattern of events that is required of a satisfying execution.

\[-\text{start}]*(\text{start;}[-\text{excluded},\text{next}]*;\text{next}; \ldots;[-\text{start}]*)*

Data races, mutual exclusion, general forms of invariance, response and precedence properties can be 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\ \text{all} \ [-a]*;(a;[-c]*;c;[-a]*)*

4.4.4 Property Automata

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 sequences over the event alphabet that correspond to the property of interest. Formally,

**Definition 16** A **property automaton** is a deterministic finite-state automaton \((S, \delta, A, s, \Sigma_{\text{property}})\), where:

\[
S = \{s_1, s_2, \ldots, s_k\} \\
\delta \mid S \times \Sigma_{\text{property}} \rightarrow S \\
A \subseteq S \\
s \in S \\
\Sigma_{\text{property}} \subseteq \Sigma
\]
PA states, $S$, represent equivalence classes of prefixes of strings over $\Sigma_{property}$. The state transition function, $\delta$, maps a state and a symbol from the properties event alphabet to a next state. The set of accepting states, $A$, are reached by strings that satisfy the property. The unique start state, $s$, is the initial state for all string recognition. The property alphabet, $\Sigma_{property}$, is a subset of the program's event alphabet, $\Sigma$.

Figure 4.10 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 violated the property in such a way that no extension of the string can possibly satisfy the property; we call these states trap states.

**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 "$|", ",", \text{ and } "+"$. In the worst case, the number of symbols in the new expression will be $|\Sigma_{property}|$, the number of alphabet symbols, times larger. For any program, we can bound the size of the alphabet of any property to be checked at the size of the program event alphabet, i.e., for any property $|\Sigma_{property}| < |\Sigma|$; in this case the blow up in the number of symbols from QREs to canonical expressions is bounded by $|\Sigma|$ which is a constant for a given program.
We construct the PA directly from the regular expression of the QRE using algorithm 3.5 from Aho, Sethi and Ullman [3], thereby avoiding the construction of an intermediate non-deterministic finite-state automaton. The construction algorithm consists of three steps, which we describe at a very high-level:

1. Parse regular expression into a syntax tree.

2. Perform a depth-first traversal of tree to gather information.


It is well known [3] that there exist regular expressions for which the smallest DFSA is exponential in the number of symbols in the expression. Thus, the worst-case cost of the third step of the algorithm dominates the cost of the first two. This algorithm is $O(2^{s_{\text{expr}}})$ where $s_{\text{expr}}$ is the number of symbols in the QRE. The expressions that exhibit this worst-case behavior typically involve some kind of counting; we have encountered very few natural properties of concurrent programs that require this type of expression.

PA Minimization

In Section 4.5 we will see that the cost of performing state propagation analysis is strongly dependent on the number of states in the property automaton. It is well known [3] that for every DFSA there is a DFSA that accepts the same language and has a minimal number of states. We use algorithm 3.6 from [3] to perform PA minimization; the algorithm is $O(|S| \log(|S|))$.

Minimization would seem to always be desirable. Surprisingly, however, we have found examples for which there is a net increase in total analysis time when minimization is performed.


4.5 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 for performing this comparison. We have defined a flow graph, the TFG, that represents all sequences of program events that correspond to actual program executions. We have defined a finite-state automaton, the PA, that represents all sequences of program events that satisfy a user defined property; it is associated with an indication as to whether the property should hold on all or no program executions. As described in Section 4.2 we could enumerate all paths in the TFG with each path yielding a sequence of program events. We could submit each event sequence to the PA. If the PA accepts every such sequence the property is verified to hold on all program executions. If the PA accepts no such sequence the property is verified to hold on no program executions. This exhaustive strategy is impractical since the number of TFG paths is, in general, infinite.

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 [40] and later Olender and Osterweil [61] 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 represented as a TFG.

4.5.1 Background

Data flow analysis is a process of uncovering facts about executable program behavior without actually running the program. A data flow analysis problem can be characterized as a system of simultaneous equations; the variables on the LHS of
these equations hold facts that are true at certain points in the computation and the expressions on the RHS describe the effects of a portion of the program with respect to the information gathered. Solving the system of equations allows us to gather global program wide information from this inherently local description of program behavior.

There are a variety of ways one can go about describing a data flow analysis. We use data flow analysis frameworks since a mature body of algorithms and theory is available for problems formulated in this way. In Chapter 3 we described data flow frameworks and extend existing theory and algorithms to make them applicable to data flow analyses on concurrent programs.

Cesar

Olender and Osterweil [61] present a distributive data flow framework for solving the state propagation analysis problem. We review their results and describe the intuition behind the algorithm.

Recall, we have a property automaton, $P$, that accepts a sequence of program events. We have a flow graph, $G$, whose paths include all executable sequences of program events. As describe in Section 4.4, we check the consistency of property and program by checking either language containment, $L(G) \subseteq L(P)$, or empty language intersection, $L(G) \cap L(P) = \emptyset$.

To simplify the discussion consider just the language containment test. Ultimately, we aren’t 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 Olender and Osterweil does exactly that. Let $\delta^*$ be the extension of $\delta$, $P$’s state transition function, from symbols in $\Sigma$ to sequences of
symbols in $\Sigma$. Two paths, $p_1 = n_{\text{start}} \rightarrow \ldots \rightarrow n_1$ and $p_2 = n_{\text{start}} \rightarrow \ldots \rightarrow n_2$, are $P$-equivalent if they cause $P$ to transition to the same state, $\delta^*(\text{String}(p_1)) = \delta^*(\text{String}(p_2))$. Figure 4.11 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; $\delta^*(\text{String}(1 \rightarrow 4)) = \delta^*(ba) = 1$ and $\delta^*(\text{String}(1 \rightarrow 2 \rightarrow 1 \rightarrow 2)) = \delta^*(baba) = 1$. The strength of this collapsing lies in the fact that it is based on the structure of the property automaton that describes the intended sequences of program events we are reasoning about. The reduction in cost is related to the interaction of the structure of the PA and the structure of the TFG. This collapsing can dramatically reduce the cost of performing the language containment test. As we will see in Chapter 5 this collapsing comes at a cost; the algorithm loses information that might be useful in detecting the source of a program fault or cause of a spurious results.

Formally, state propagation analysis computes for each node the set of PA states that can be reached by any path to that node from the start node of the graph, $\text{States}(n) = \{\delta^*(p) : p = n_{\text{initial}} \rightarrow \ldots \rightarrow n\}$. We define a meet-lattice over the power-set of PA states with union for meet and super-set as the ordering relation, $L_{sp} = (\mathcal{P}(S), \supseteq, \cup)$; in this lattice $\bot = S$, the set of all PA states, and $\top = \emptyset$. The function space is defined by extending all total functions over PA states to the power-set of PA states, $F_{sp} = \{F : F(X) = \cup_{x \in X} f(x)\}$, where $X \in \mathcal{P}(S)$. Theorem
5.1 in [59] implies that $D_{sp} = (L_{sp}, F_{sp})$ is a distributive framework. The framework is instantiated for a flow graph by adding an additional initial node, $n_{init}$. The function map $M_{sp}$ is defined such that for each node, $n$, to each value flowing into that node, the PA state transition function is applied to produce the value at the node, $f_n(X) = \{\delta^P(L(n), X)\}$, where $\delta^P$ is the element-wise extension of $\delta$ to a set of PA states. To satisfy the frameworks initial conditions, namely that the value of initial node is $\bot$, we map the constant function that returns the PA start state to the initial flow graph node, $f_{n_{init}}(X) = \{s_{start}\}$. Theorem 5.2 in [59] tells us that the MOP solution of an instance of $D_{sp}, (G, M_{sp})$, is $States^8$.

4.5.2 Adapting State Propagation to Concurrent Programs

In this section we discuss how we adapted Olender and Osterweil’s algorithm to concurrent programs represented as TFGs. We address the applicability of $D_{sp}$ to flow graphs for concurrent programs and the precision of the analysis results that can be obtained.

Applicability

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 the MOP solution contains conservative values for $States$ at each flow graph node. From Theorem 4 we know that the TFG represents, for each node, all executable sequences of events from the initial node to that node. In addition, the TFG already contains an initial node, $n_{initial}$. Using the same function map as above, we construct $I_{tfg-sp} = (TFG, M_{sp})$ as an instance of $D_{sp}$. The MOP solution at the final TFG node, $States(n_{final})$, can be compared to the set of accepting states of the PA, $A$, to produce a conservative

---

8This proof is done for edge labeled flow graphs. The proof carries over directly to node labeled flow graphs.
version of either a language containment, \( States(n_{\text{final}}) \subseteq A \), or empty intersection test, \( States(n_{\text{final}}) \cap A = \emptyset \).

As discussed in Section 4.3 a program execution is modeled as a collection of TFG paths, one in each task sub-flow graph, that share nodes at inter-task communication or synchronization points.

\[
States_{\text{TFG}}(n) = \begin{cases} 
\bigcap_{p \in \text{Preds}(n) \land \text{kind}([p,n]) = \text{MIP}} \delta^P(States_{\text{TFG}}(p)) \cup \\
\bigcup_{p \in \text{Preds}(n) \land \text{kind}([p,n]) \neq \text{MIP}} \delta^P(States_{\text{TFG}}(p)) & \text{if kind}(n) = \text{COM} \mid \text{FINAL} \\
\bigcup_{p \in \text{Preds}(n)} \delta^P(States_{\text{TFG}}(p)) & \text{otherwise}
\end{cases}
\]

In other words, we intersect values at the nodes where the separate paths intersect.

**Precision**

While \( D_p \) is applicable to concurrent programs the solution is imprecise. We can improve the precision of state propagation analysis by incorporating the execution semantics of concurrent programs into our data flow framework. Intuitively, we take advantage of the fact that at certain points during program execution two, or more, tasks may synchronize their execution. This occurs trivially when all tasks terminate at the completion of the program. There are a number of languages, including Ada, in which inter-task communication is accomplished by sender and receiver synchronizing to exchange data. If we can identify these communication points we can take advantage of the semantics of synchronization. The TFG makes such communication points explicit for Ada programs; a COM node represents the joint, synchronized, state of both the sender and receiver. In addition, the initial and final TFG nodes represent program states at which all program tasks are synchronized before execution begins and after it completes. Figure 4.12 illustrates a TFG fragment. This TFG is constructed to represent the event sequences \( \ldots pabcd \ldots \) and \( \ldots pbacd \ldots \); the events \( a \) and \( b \) may swap execution order\(^9\). To illustrate the potential imprecision

---

\(^9\)In general, this might involve the interleaving of event sequences from multiple tasks. In this example, we have the, almost trivial, interleavings of \( a \) and \( b \), namely \( ab \) and \( ba \).
of an analysis based on $D_{sp}$ for a TFG consider the computation of the set of PA states at node 27 in Figure 4.12, States(27). The meet operator in $L_{sp}$ is $\cup$, thus we will compute the values flowing in to node 27 based on all paths leading to nodes 4 and 7. The event sequences leading to node 4 include: \ldots pabc, \ldots pbac, \ldots ac, and \ldots abc. The event sequences leading to node 7 include: \ldots pabc, \ldots pbac, \ldots pb, and \ldots pab. Theorem 4 tells us that the set of paths leading to node 4 include all executable sequences of events that leave task 1 in the state corresponding to node 4, similarly for node 7 in task 2. For any execution of the program that reaches a communication event, it must be the case that the communicating tasks have reached send and receive statements. The state propagation algorithm will compute separate information about the sequences of program events that can lead to the send statement and the receive statement. For synchronous communication, we know that any program execution leading to the send must also lead to the the receive, and vice versa. Consequently, we can eliminate any event sequences that are not at both the send and receive nodes, because they cannot represent feasible program executions. For our example, by intersecting event sequences we get a result that includes the executable event sequences \ldots pabcd\ldots and \ldots pbacd\ldots and excludes unexecutable sequences \ldots ac, \ldots abc, \ldots pb, and \ldots pab. It appears that we should use $\cap$ as our
meet operator in this context. In fact, what we want to do is to select the combining function to use at merge points based on the kind of flow graph node we have.

The Improved State Propagation Framework

We construct an improved state propagation analysis by reformulating $D_{sp}$ as a complete-lattice framework.

We extend the semi-lattice from $D_{sp}$ with a join operator to get $L_{improved} = (\mathcal{P}(S), \supseteq, \cup, \cap)$; in this lattice $\bot = S$, the set of all PA states, and $\top = \emptyset$. As in $D_{sp}$ the function space is defined by extending all total functions over PA states to the power-set of PA states, $F_{sp} = \{f : f(X) = \{f(x) : x \in X\} : X \in \mathcal{P}(S)\}$.

**Theorem 7**

$D_{improved} = (L_{improved}, F_{sp})$ is a distributive complete-lattice data flow analysis framework.

Proof:

Both $\cup$ and $\cap$ are idempotent, associative and commutative. The rest of the complete-lattice properties are as follows:

\[
x \cup y \supseteq y
\]
\[
x \cap y \subseteq y
\]
\[
x \cup y = x \iff x \supseteq y
\]
\[
x \cap y = y \iff x \supseteq y
\]
\[
\forall x \in \mathcal{P}(S) : S \supseteq x
\]
\[
x \cap (x \cup y) = x
\]
\[
x \cup (x \cap y) = x
\]

Furthermore, since all subsets of $\mathcal{P}(S)$ are finite then all iterated meets and joins are finite and they exist.
Figure 4.13 Splitting TFG Nodes

$F_{sp}$ was defined above to distribute over the semi-lattice $L_{sp}$ embedded in $L_{improved}$, thus it distributes over $L_{improved}$’s ordering and meet operators.

An instance of $D_{improved}$ is created by providing a flow graph, merge and function maps. The flow graph is a TFG where the COM nodes have been split into a $\tau$ labeled part with incident control flow edges only, called a $CF$-COM node, and a part with incident MIP edges, called a $MIP$-COM node. A COM node may have both communication and control flow successors. In order to use $\cap$ to combine values at synchronizing communication points and $\cup$ to combine values at control flow merge points we split COM nodes. Figure 4.13 illustrates the node splitting operation. Let $N_c$ be the $CF$-COM nodes and $n_{final}$. Let $N_\cup = N - N_c - N_{initial}$.

We reuse the function map $M_{sp}$ from the formulation of $D_{sp}$. It is defined such that for each node, $n$, to each value flowing into that node the PA state transition function is applied to produce the value at the node, $f_n(X) = \{\delta^P(L(n), x) : x \in X\}$. To meet the framework’s initial definition, namely that the value of initial node is $\perp$, we map the constant function that returns the PA start state to the initial TFG node, $f_{n_{initial}}(X) = \{\delta_{start}\}$.
Theorem 8 (Conservativeness of Improved State Propagation)

The maximum fixed point solution of an instance of \(D_{\text{improved}}\) is a conservative estimate of \(S_{\text{TFG}}(n)\) for each node in the TFG.

Proof:

From Theorem 1 we have that the MFP solution, \(X\), of an instance of \(D_{\text{improved}}\) is given as:

\[
X[n_{\text{initial}}] = S
\]

\[
\forall n \in N_{\cup} : X[n] = \bigcup_{p \in \text{preds}(n)} \delta^p(X[p])
\]

\[
\forall n \in N_{\cap} : X[n] = \bigcap_{p \in \text{preds}(n)} \delta^p(X[p])
\]

By definition the \(N_{\cap}\) as the set of CF-COM nodes and \(n_{\text{final}}\). The cumulative effect of consecutive CF-COM and MIP-COM nodes, one in \(N_{\cap}\) and the successor in \(N_{\cup}\), is the same as the computation for non-split COM nodes in the definition of \(S_{\text{TFG}}\). This follows from the fact that \(f_{\text{ident}}\) is mapped to all CF-COM nodes and that \(\cup\) is commutative. Thus, the solution \(X\) satisfies the definition of \(S_{\text{TFG}}\) and the theorem holds. \(\square\)

4.5.3 Evaluating Solution Methods

As discussed in Chapter 3 there is a large body of research and practice relating to solution algorithms for and formulations of data flow analysis problems. Choosing the appropriate algorithm and formulation requires a careful assessment of the problem to be solved. We start by enumerating and discussing features of the TFG and PA.

Reducibility Due to the presence of MIP edges TFGs are irreducible.

Node Connectedness Nodes in a TFG are typically highly connected due to the presence of MIP edges.
Fan In/Out  Nodes in a TFG have a large number of predecessors and successors due to the presence of MIP edges.

PA States  PAs can have a large number of states.

We briefly discuss the rationale behind our selection of a solution algorithm for and the details of our formulation of an instance of $D_{improved}$.

Selecting a Solution Algorithm

There are three classes of data flow analysis solution methods[51]: path-algebra, elimination and iteration.

Path-algebra methods construct a single transfer function that represents the effects of paths in the flow graph that lead to a node. These path transfer function are the composition of transfer functions for the nodes on the paths. To compute the value at a flow graph node we need only evaluate its path transfer function on the initial lattice value. Intuitively, we have replaced the propagation of lattice values and evaluation of individual transfer function with this single path transfer function. This method is effective when we are interested in values at only a few nodes in the graph, when the cost of computing path transfer functions is low, and when the cost of evaluating the path transfer function is low. For state propagation analysis, we are interested primarily in the values of the final nodes of each task; since this is a small minority of all TFG nodes a path-algebra method may be appropriate. Unfortunately, computing and evaluating path transfer functions will be expensive. To illustrate, imagine a path in the TFG that does not contain flow graph merges; for such a path, call it $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_k$ we compute the path transfer function, $T_p(x)$, as the composition of node transfer functions $\delta(L(n_k)) \ldots \delta(L(n_2)), \delta(L(n_1), x)$. Since we know the node labels we can compute $T_p$ in $O(k|\Sigma_{property}|)$. At flow graph merges we must incorporate the appropriate confluence operator, union in the case of the basic state propagation algorithm. Unlike merge-free paths, there is no compact
encoding of the composition of path functions and the confluence operator. If the TFG contained long paths without merges then there may be an opportunity for significant savings through evaluating $T_p(x)$ for the path. Unfortunately, the structure of TFGs implies that merge-free paths will be very uncommon due to the presence of MIP edges. Consequently, the cost of evaluating a path transfer function will be essentially the same as evaluating each node transfer function along that path in the proper sequence; this negative result combined with the significant overhead involved in computing and storing the path transfer functions leads us to conclude that path-algebra methods are not appropriate for state propagation analysis.

Elimination methods derive their name from Gaussian elimination [72]. They rely on the ability to decompose the input flow graph into sub-graphs that have a high-degree of internal connectivity and a relatively small number of external connections; common decompositions are based on the notions of intervals [3, 51] or regions [3, 51]. Intervals correspond to natural loops in the flow graph, while regions correspond to groups of nodes dominated by a single node; flow graph domination plays a key role in the identification of sub-graphs that are intervals or regions. TFG sub-graphs that correspond to each program task are well structured and amenable to such decomposition, however, the communication nodes and MIP edges in a TFG cut across those structured regions; the result is an irreducible flow graph that is unsuitable for interval or region based analysis. Techniques such as node splitting [3] have been proposed as a means for eliminating irreducibilities in flow graphs. These techniques are intended to handle graphs that are almost completely reducible. TFGs will have many irreducible sub-graphs and consequently node splitting might well encounter its worst-case exponential behavior if applied to such graphs. For these reasons we believe that elimination methods are not appropriate for state propagation analysis.

Iteration is the most general of the solution methods. It can be applied to irreducible flow graphs and for bounded monotone frameworks we can derive low-order
polynomial bounds on the solvers. Iterative methods can involve significant redundant or useless intermediate computation. One approach to address this issue and improve the performance of iterative solution methods is to process flow graph nodes in a specific order; this order can have a significant impact on both the practical and theoretical performance of an algorithm. For structured flow graphs we can compute an order, such as $RPOSTORDER$ [34], prior to analysis. Worklist algorithms dynamically compute an order and are applicable when the flow graph is less structured. Because TFGs are relatively unstructured we use an iterative worklist algorithm for the state propagation algorithm. An important pragmatic benefit of iterative algorithms is that they are typically much easier to implement and have smaller overhead costs than path-algebra or elimination methods.

Given our formulation of $D_{improved}$ as a distributive complete-lattice framework we know that an iterative algorithm will compute the MFP solution for state propagation. In order to select a particular algorithm and reason about its execution bound we need to first reason about the cost of lattice operations and transfer function evaluations.

**Formulating the Data Flow Framework**

Many data flow analysis problems deal with sets of values and are formulated as data flow frameworks using a power-set lattice. For problems in which the set is known prior to analysis, one can use a bit-vector representation of the sets. These *bit-vector problems* are well studied and widely applicable [45].

Now that we have settled on an iterative worklist algorithm we need to evaluate our options for defining the components of $L_{improved}$ and $\delta^P$.

First consider the lattice values, ordering, meet and join operations. The number of PA states is fixed throughout the solution of the state propagation algorithm. Thus, we can consider both a bit-vector and linked-list representation of an element of $\mathcal{P}(S)$. 
**bit-vector** There are two primary advantages of this representation: operations are very fast and the memory footprint of a value can be small. Operations such as computing ordering, meet and join are "almost constant time"; more precisely they are $O(|S|/wsize)$ where $wsize$ is the word size on the machine. The memory required to store each value is the same $O(|S|)$. When values are dense or $|S|$ is small, bit-vectors are a memory efficient representation.

**linked-list** The chief advantage of this representation, its flexibility, is wasted in this application. We note that for large sparse values this representation can be significantly more memory efficient than bit-vectors.

Now consider the transfer functions which are built out of $\delta^P$.

**bit-vector** The PA transition function $\delta$ operates on a single PA state value at a time, i.e., a single set bit in the vector. The extension of $\delta$ to sets of PA states, namely $\delta^P$, conceptually applies $\delta$ simultaneously to all states represented by the bit-vector. An "almost constant time" implementation of $\delta : S \times A \to S$ is as a $|S|$ by $|A|$ matrix whose elements are in $S$; total space is $|S||A|$. The analogous implementation of $\delta^P : \mathcal{P}(S) \times A \to \mathcal{P}(S)$ would be as a $2^{|S|}$ by $|A|$ matrix elements whose elements are bit-vectors of length $|S|$; total space is $|S||A|2^{|S|}$. For PAs with a large number of states it would be impractical to construct such an implementation of $\delta^P$.

**linked-list** For this representation, the natural implementation of $\delta^P$ is the application of $\delta$ to each element of a given value in $\mathcal{P}(S)$. This requires $O(|S|)$ steps in contrast to "almost constant time" for the bit-vector implementation. There is a significant savings in space with this approach, making it practical for PAs with very large numbers of states.
It appears that we have a classic time-space tradeoff. Rather than choose between a pure bit-vector or linked-list formulation, we have developed a hybrid approach that has many of the advantages of the bit-vectors without the space penalty.

We use a bit-vector representation for $P(S)$ and use the standard bit-vector implementations of $\sqcup$, $\cap$ and $\sqcup$ for lattice ordering, meet and join respectively.

We implement $\delta^P$ as the iterated application of $\delta$, as described above for the linked-list representation. At first glance, the fact that $\delta^P$ is $O(|S|)$ appears to violate the assumption that transfer functions are $O(1)$ and thus, invalidates application of Theorem 3. We argue that the structure of our data flow analysis problem allows application of this theorem.

First a definition. Let $IN[n]$ be the meet/join of the values from predecessor of $n$; it is the value flowing into a node. Let $OUT[n]$ be the value at node $n$ after applying the transfer function to its input. From the monotonicity of the function space we know that once a value $v$ appears as $IN[n]$ then all subsequent values $v'$ appearing at $IN[n]$ are such that $v \subseteq v'$. We can keep track of the most recent value of $IN[n]$ to restrict evaluation of the $\delta$ to only those PA states which have not already been processed on prior visits to node $n$.

$$f_n(X) = OUT[n] \cup \bigcup_{x \in X - IN[n]} \delta(L(n), x)$$

Note that $IN$ and $OUT$ are the values from the previous visit to node $n$; they will be updated at the end of the current visit to the node. The effect of this implementation is to spread the cost of evaluating $\delta^P$ at a node over all visits to that node during execution of the solution algorithm. The cost of considering each possible PA state value at each node in the TFG is inherent in the state propagation problem; we can either pay that cost by visiting a node many times where each node visit is cheap or by making very few node visits where each node visit is more expensive.
We can define the transfer functions so as to allow application of Theorem 3, but that definition requires some bookkeeping. We now argue that the additional bookkeeping cost does not increase the time bound of the solution algorithm.

Every visit of a node in Algorithm 2 requires $O(|N|)$ work. The transfer function defined above requires that we keep 2 $|S|$ bit-vectors for each node. These bit-vectors are referenced and updated on each visit to a node; since these are bit-vectors, the costs of these operations are "almost constant time". In practice, we believe that for most problems $|S| < \text{wsize}|N|$, where \text{wsize} is the implementation machine's word size. If that is not true then the cost of a node visit will be dominated by the bookkeeping costs and the complexity of Algorithm 2 for the state propagation problem will be $O(|N|(|S|^2))$ rather than $O(|S|(|N|^2))$.

**Theorem 9 (Complexity of Improved State Propagation Solver)**

The cost of solving an instance of $D_{improved}$ is $O(|S|(|N|^2))$.

**Proof:**

We have that the cost of lattice ordering, meet, join and the evaluation of a transfer function are all $O(1)$. Since height of lattice $L_{improved}$ is $|S|$, from Theorem 3 there is an $O(|S|(|N|^2))$ iterative worklist solution algorithm for this data flow analysis problem.

4.5.4 Summary

We introduced a state propagation algorithm for sequential programs and argued that it is imprecise when applied to concurrent programs. To address this, we formulate a more precise state propagation algorithm as a distributive complete-lattice data flow analysis framework. The solution to an instance of this framework is shown to be correct. Reasoning about the details of the problem formulation allows us to apply
existing results to obtain a solution algorithm for our improved state propagation algorithm that has polynomial-bound on its running time.

4.6 Increasing Efficiency and Precision

The state propagation analysis algorithm described in Section 4.5 is quadratic in the number of TFG nodes. This bound suggests that the cost of analysis will grow relatively slowly with graph size, however, the cost of analysis for large graphs and complex properties may still be significant. In this section, we discuss a variety of techniques for reducing the size of the TFG and, subsequently, the cost of computing analysis results.

The conservative nature of our model of program execution, the TFG, and the state propagation analysis can lead to analysis results that are inconclusive. In this section, we discuss a variety of techniques for encoding additional information into the analysis to eliminate spurious analysis results.

By construction the TFG includes all event orderings that can occur in some program execution; it may also include orderings that are not executable. There are three major sources of inaccuracy in the TFG representation: unexecutable control flow paths, unexecutable intra-task communication, and unexecutable orderings of asynchronous program events. Imprecision related to asynchronous event orderings is manifested in the TFG by the presence of MIP edges; these edges are included to represent sequences of program events that span multiple program tasks. This is accomplished by using a MIP edge to represent the fact that, on some program execution, an instance of an event in one task immediately precedes an instance of an event in another task. The event symbols in these instances will be adjacent in the sequence of program events that corresponds to that program execution. Unfortunately, adding these MIP edges introduces other paths in the TFG; many of those do not correspond to executable sequences of program events. Figure 4.14 illustrates a TFG.
fragment with some (solid) control flow edges and (dashed) MIP edges. MIP edge \((\text{Task1}: 3, \text{Task2}: 7)\) exists because it is required to represent executable sequences of events of the form \(\ldots ab\ldots\); the reverse direction MIP edge exists to represent sequences of the form \(\ldots ba\ldots\). 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 two MIP edges, however, form a cycle; this introduces arbitrarily long TFG paths containing nodes 3 and 7.

We introduce TFG refinements into the basic analysis illustrated in Figure 4.1 to eliminate sources of imprecision that are independent of particular TFG paths. The example above illustrated MIP edges that are candidates for elimination by refinements. A refinement consists of two phases: an analysis phase that gathers information about the TFG and a transformation phase that modifies the TFG. In Section 4.6.1, we present two refinements and describe the analyses and transformations that they perform.

The state propagation algorithm constrains the set of event sequences considered at task synchronization and program termination nodes. Additional constraints, called feasibility constraints, are incorporated into the basic analysis illustrated in
Figure 4.1. The goal is to eliminate sources of inaccuracy that depend on particular TFG paths. Maintaining full information about program paths is impractical. Feasibility constraints encode only partial information about program paths and have the potential to greatly improve the precision of analysis results with a modest increase in analysis cost. Incorporating feasibility constraints requires two steps: construction of the constraint itself and combining it with the property to be checked. In Section 4.6.2, we present two classes of feasibility constraints and a general algorithm for combining such constraints with a given property automaton.

We conclude this section with a discussion of mechanisms for combining the results of independent analysis to eliminate imprecision across a series of analyses.

4.6.1 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 can reduce the cost of performing state propagation analysis. Eliminating unexecutable behaviors can improve the precision of state propagation analysis results.

State propagation is, asymptotically, inexpensive and fundamentally approximate. Given this, we view cost reduction as an opportunity for increasing precision without a net increase in analysis cost. We can think of the TFG as encoding necessary conditions for a sequence of program events to correspond to a program execution; any sequence that does not satisfy those conditions, i.e., is not represented by the TFG, cannot correspond to a program execution. We call such necessary conditions executability conditions. The process of refinement is the strengthening of the executability conditions that are encoded in the TFG.

All of our refinements are local in the sense that they reason about a sub-graph, or fragment, of a TFG that represents some part of a program execution. Many refinements are concerned with trivial fragments, such as individual nodes or edges.
The analysis phase computes information about fragments that indicates whether a fragment is irrelevant for determining the satisfaction of executability conditions. These fragments can be ignored for the purpose of state propagation analysis.

violates some executability condition. A violation indicates a sequence of unexecutable events. The fragment that represents that sequence can be removed from the TFG to improve the precision of state propagation analysis.

weakly satisfies all executability conditions. A weak condition is poor at distinguishing executable from non-executable sequences of events. Such a fragment can be transformed into a form that more precisely encodes information about executable program event sequences.

We have developed TFG refinements that rely on each of these forms of program information.

Many of these 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 in the analysis phase. The cost of performing refinements is only dependent on the size of the relevant parts of the TFG.

In the remainder of this section we describe alphabet refinement which eliminates irrelevant TFG fragments and communication interval refinement which eliminates violating TFG fragments. For each of these we describe the intuition behind the refinement, present the refinement algorithm, and give correctness and complexity arguments.

4.6.1.1 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, $\Sigma_{\text{property}}$, and the ordering relationships
between those events. We can perform alphabet refinement of a TFG for a given property by relabeling nodes in the TFG whose labels are not in $\Sigma_{property}$. The resulting TFG can be further transformed to eliminate edges and nodes that do not add to the set of event sequences over the reduced event alphabet. A general algorithm can be defined to perform transformations based on the detection of irrelevant TFG fragments. These transformations include: collapsing edges to $\tau$ or non-$\Sigma_{property}$ labeled nodes, eliminating those nodes, and collapsing parallel edges. The resulting TFG will have no paths that are local to a task with consecutive $\tau$ labeled nodes. We note that in order to apply the improved state propagation analysis to a TFG we must maintain the distinction between nodes in $N_\tau$ and nodes in $N_\tau^\perp$. We can merge nodes within these sets but not across the sets. Our approach is to use a well understood partition-refinement algorithm [2, 3].

Algorithm 4 (Alphabet Refinement)

Input:

A TFG and $\Sigma_{property}$, the alphabet of the property of interest.

Output:

A TFG whose alphabet is $\subseteq \Sigma_{property}$.

Initialization:

This algorithm is applied to a view of the TFG that elides MIP edges. We refer to the projection of the labeling function $L$ onto the alphabet $\Sigma_{property}$ as $L_{property}$. We construct an initial partition $\Pi_{init}$ of the set of TFG nodes such that:

- all nodes that are LOCAL to a given task are in the same group of the partition
- nodes in different tasks are in different groups
the INITIAL and FINAL nodes have their own groups

- each COM node has its own group \(^{10}\).

The function that maps a node, \(n\), to its group in a partition, \(\Pi\), is given as \(\text{Group}(\Pi, n)\).

\begin{enumerate}
\item \(G_i, G_n = \emptyset\)
\item \(G_{\text{initial}} = \{n_{\text{initial}}\}\)
\item \(G_{\text{final}} = \{n_{\text{final}}\}\)
\item for each node, \(n\), where \(\text{Kind}(n) = \text{LOCAL loop}\)
\item \(G_i = G_i \cup n\) where \(\text{Task}(n) = T_i\)
\item end loop
\item for each node, \(n\), where \(\text{Kind}(n) = \text{COM loop}\)
\item \(G_n = \{n\}\)
\item end loop
\end{enumerate}

**Main Loop:** \(^{11}\)

\begin{enumerate}
\item \(\Pi = \Pi_{\text{init}}\)
\item \(\Pi_{\text{new}} = \Pi\)
\item for each group, \(G_i\), of \(\Pi\) loop
\item partition \(G\) into \(G_1, G_2, \ldots\) such that
\item \(x, y \in G_i\) iff
\item \(\exists x', y' \in N : \text{L}_{\text{property}}(x') = \text{L}_{\text{property}}(y') \land\)
\item \(\text{Group}(\Pi, x') = \text{Group}(\Pi, y') \land\)
\item \((x, x') \in E \land (y, y') \in E\)
\end{enumerate}

\(^{10}\)This restriction may be relaxed to allow collapsing of consecutive irrelevant communication events. We are working on a careful characterization of when this is profitable and safe.

\(^{11}\)This is a version of Algorithm 3.6 [3] that has been modified to apply to flow graphs rather than deterministic finite state automata.
(5b) \[ \exists x', y' \in N : L_{\text{property}}(x') = L_{\text{property}}(y') \land \\
\text{Group}(\Pi, x') = \text{Group}(\Pi, y') \land \\
(x', x) \in E \land (y', y) \in E \]

(6) replace \( G \) with the \( G_1, G_2, \ldots \) in \( \Pi_{\text{new}} \)

\[ \text{end loop} \]

(7) if \( \Pi \neq \Pi_{\text{new}} \) then

(8) \( \Pi = \Pi_{\text{new}} \)

(9) goto line 3

\[ \text{end if} \]

(10) Construct the new TFG using the groups of \( \Pi \) as nodes

To insure that state propagation analysis remains conservative on an alphabet refined TFG we must demonstrate that the resultant TFG includes all executable event sequences over \( \Sigma_{\text{property}} \).

**Theorem 10 (Correctness of Alphabet Refinement)**

An alphabet refined TFG is a conservative representation of the set of executable program event sequences over \( \Sigma_{\text{property}} \).

**Proof:**

If the input TFG is conservative then it represents all executable program event sequences.

The key point is line 5 which defines the equivalence measure on which partitioning is based. This says that two nodes are equivalent if they have successors(predecessors) with the same label in \( \Sigma_{\text{property}} \) where those successors(predecessors) are themselves equivalent. In other words, two nodes are equivalent, according to the measure of line 5, if for each path leading from one to \( n_{\text{final}} \) there is an equivalent path leading from the other to \( n_{\text{final}} \), and if for each path leading to one from \( n_{\text{initial}} \) there
is an equivalent path leading to the other from $n_{\text{initial}}$. Thus, alphabet refinement preserves equivalent flow graph paths and consequently the set of executable program event sequences over $\Sigma_{\text{property}}$.

Note that the initialization phase of Algorithm 4 insures that node and edge collapsing will be isolated to TFG fragments that represent local control flow within tasks. Consequently, the structural properties of the TFG that allow application of the improved state propagation algorithm are maintained.

**Theorem 11 (Complexity of Alphabet Refinement)**

*Alphabet refinement requires $O(|N|\log(|N|))$ time.*

**Proof:**

Under the assumption that the equivalence measure used in the partition-refinement algorithm requires $O(1)$ steps we know the algorithm as a whole will require $O(|N|\log(|N|))$ [2].

It is not obvious that the computation in line 5 is independent of $|N|$ much less $O(1)$. Let's supply some details about the computation in line 5 so that we can reason about its cost.

We assume that symbols in $\Sigma_{\text{property}}$ can be ordered. For each node, $n$, we compute two arrays, $Pmap(n)$ and $Smap(n)$, of length $|\Sigma_{\text{property}}|$ where each element is a bit-vector of groups. The value of the $i^{\text{th}}$ element of $Pmap(n)$ represents the set of groups that contain the predecessors of $n$ that are labeled by the $i^{\text{th}}$ symbol. The value of the $i^{\text{th}}$ element of $Smap(n)$ represents the set of groups that contain the successors of $n$ that are labeled by the $i^{\text{th}}$ symbol. Two nodes, $n$ and $m$, are equivalent if $Pmap(n) = Pmap(m)$ and $Smap(n) = Smap(m)$. Checking these
equivalences is $O(|\Sigma_{\text{property}}|)$ since we can compare arrays by performing $|\Sigma_{\text{property}}|$ bit-vector equality tests. For any property the alphabet of the property is a subset of the alphabet of the TFG. Thus, all alphabet equality tests for nodes in a TFG are bounded by $O(|\Sigma|)$; since $\Sigma$ is constant for a TFG the equality test is $O(1)$ and the theorem holds.

□

We will see in Chapter 5 that this simple refinement can have a great effect on the size of the TFG and on the cost of state propagation analysis.

4.6.1.2 Communication Interval Refinement

Many programs contain patterns of communication that serve to implement critical sections, transaction-like structures, regions of mutual exclusion, and protocols for acquiring and releasing resources. 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 \(^{12}\).

A communication interval is a TFG fragment involving two program tasks and containing at least two communication nodes. These communication nodes, called the start and end events of the interval, designate the entry to and exit from the fragment. Intuitively, an interval consists of a collection of statements in two tasks, such that, if one task is executing one of the statements in the interval, then the other task must also be executing one of the statements in the interval. Figure 4.15 illustrates an isolated communication interval, where only two tasks are involved $T_1$ and $T_2$. The dotted borders enclose the local nodes in each task. Control flow edges are solid and MIP edges are dashed; note that for clarity we have only illustrated a few of the MIP edges. All of the nodes except for $a_1$, $a_2$, $b_1$, and $b_2$ are in the interval. The COM

\(^{12}\)This refinement depends on intra-task communication being accomplished by task synchronization.
node labeled start is a point at which task $T_1$ and $T_2$ synchronize. The tasks proceed independently executing if, $a, b$ for $T_1$ and $c, d$ for $T_2$. The COM node labeled stop is another point at which the tasks synchronize. The structure of this TFG fragment makes it impossible for either task to avoid engaging in the stop communication once the start communication has occurred; similarly the only way the stop communication can occur is after the start communication. This implies that the portions of program event sequences represented by this fragment are constrained such that an event that occurs within the interval in $T_1$ can neither immediately precede nor immediately succeed an event outside the interval in $T_2$. This is true because either the start or

Figure 4.15 Communication Interval Example
stop communication events is guaranteed to occur *between* any such pair of events. This allows us to remove MIP edges that represent such unexecutable event orderings because they are guaranteed to violate executability conditions. In Figure 4.15 the MIP edges \((b_1, d)\) and \((d, b_1)\) can be removed.

We now give a precise definition of communication intervals and an algorithm for computing them. The notion of graph domination provides information about what nodes *precede* other nodes in a walk from the root of the graph. Dominators in a TFG do not provide us with the precision required to detect TFG fragments that correspond to communication intervals. We define a related notion that gives the desired precision.

**Definition 17** A node, \(n\), **locally dominates** another node, \(m\), if \(\text{Task}(n) = \text{Task}(m)\) and all control flow paths within \(\text{Task}(n)\) from \(n_{\text{initial}}\) to \(m\) pass through \(n\).

**Definition 18** A node, \(n\), **locally post-dominates** another node, \(m\), if \(\text{Task}(n) = \text{Task}(m)\) and all control flow paths within \(\text{Task}(n)\) from \(m\) to \(n_{\text{final}}\) pass through \(n\).

These definitions are the standard notion of dominator and post-dominator applied to task sub-flow-graphs of the TFG. Within a task, a pair of communication statements can form a necessary component of a communication interval if they dominate/post-dominate each other.

**Definition 19** A pair of nodes, \((n, m)\), define a **local interval** if \(n\) locally dominates \(m\) and \(m\) locally post-dominates \(n\) and \(\text{Kind}(n) = \text{SEND or RECEIVE}\) and \(\text{Kind}(m) = \text{SEND or RECEIVE}\).

Now we can formalize the concept of a communication interval.

**Definition 20** A pair of local intervals, \((n_1, m_1)\) and \((n_2, m_2)\), define a **communication interval** if:

\[
\text{Task}(n_1) \neq \text{Task}(n_2)
\]
\[ \exists c_{\text{start}} \in N : \text{Kind}(c_{\text{start}}) = \text{COM} \wedge (n_1, c_{\text{start}}), (n_2, c_{\text{start}}) \in E \]
\[ \wedge L(c_{\text{start}}) \neq \tau \]
\[ \exists c_{\text{end}} \in N : \text{Kind}(c_{\text{end}}) = \text{COM} \wedge (c_{\text{end}}, m_1), (c_{\text{end}}, m_2) \in E \]
\[ \wedge L(c_{\text{end}}) \neq \tau \]

We do not allow \( c_{\text{start}} \) and \( c_{\text{end}} \) to have \( \tau \) labels because it would not be the case that the start and end events were guaranteed to intervene between events inside and outside the interval. Since \( n_{\text{initial}} \) and \( n_{\text{final}} \) represent synchronization points we treat them as implicit communication events for the purpose of detecting communication intervals; thus, every program has intervals bounded by the initial and final nodes in the TFG.

A communication interval is isolated if the only tasks that are involved in the start communication are the ones in the interval, otherwise the interval is multiple. If one of the tasks in an interval has receives for both the start and end communication then that is an accepting interval. We can have both isolated accepting intervals and multiple accepting intervals. Our refinement only requires that we know the identity of the nodes in each local interval of the communication interval and the kind of interval. We compute the set of communication intervals and eliminate unexecutable MIP edges using the following algorithm.

**Algorithm 5 (Communication Interval Refinement)**

*Input:*

A TFG and a pair of tasks \( T_i \) and \( T_j \).

*Output:*

A TFG with some unexecutable MIP edges eliminated.

*Algorithm:*
There are five phases of processing. The first phase computes local dominators and post-dominators; we call these $\text{Dom}(n)$ and $\text{PostDom}(n)$ for a node $n$. These values are represented as bit-vectors of length $|N|$. The second two phases compute local intervals for each task respectively. The fourth phase looks for pairs of local intervals that constitute a communication interval. The last phase eliminates MIP edges that violate the structure of some communication interval.

- **Phase 1**
  
  compute local dominators and post-dominators for $T_i$ and $T_j$

- **Phase 2**
  
  for each SEND or RECEIVE node, $n$, in $T_i$ loop
    
    for each SEND or RECEIVE node, $m$, in $T_i$ loop
      
      if $m \in \text{Dom}(n) \land n \in \text{PostDom}(m)$ then
        
        define a local interval $(n, m)$
        
        end if
      
      end loop
    
  end loop

- **Phase 3**
  
  for each SEND or RECEIVE node, $n$, in $T_j$ loop
    
    for each SEND or RECEIVE node, $m$, in $T_j$ loop
      
      if $m \in \text{Dom}(n) \land n \in \text{PostDom}(m)$ then
        
        define a local interval $(n, m)$
        
        end if
      
      end loop
    
  end loop
– Phase 4

for each local interval, \((n_i, m_i)\), in \(T_i\) loop

for each local interval, \((n_j, m_j)\), in \(T_j\) loop

if \(Succ(n_i) = Succ(n_j) \land Pred(m_i) = Pred(m_j)\) then

if \(L(Succ(n_i)) \neq \tau \land L(Pred(m_i)) \neq \tau\) then

define a communication interval \(((n_i, m_i), (n_j, m_j))\)

if \(n_i\) and \(m_i\) are RECEIVE or \(n_j\) and \(m_j\) are RECEIVE then

mark as an ACCEPTING interval

end if

if there are other SENDERS for the RECEIVE nodes

mark as a MULTIPLE interval

else

mark as a ISOLATED interval

end if

end if

end if

end loop

end loop

– Phase 5

for each communication interval \(((n_i, m_i), (n_j, m_j))\) loop

if (MULTIPLE and ACCEPTING and \(n_i\) is not RECEIVE) or ISOLATED then

for each node \(n \in Dom(n_i) \cap PostDom(m_i)\) loop

eliminate MIP edges \((n, m)\) and \((m, n)\)
where \( m \in N_j \land m \notin \text{Dom}(n_j) \cap \text{PostDom}(m_j) \)

end loop

end if

if (MULTIPLE and ACCEPTING and

\( n_j \) is not RECEIVE) or ISOLATED then

for each node \( n \in \text{Dom}(n_j) \cap \text{PostDom}(m_j) \) loop

eliminate MIP edges \((n, m)\) and \((m, n)\)

where \( m \in N_i \land m \notin \text{Dom}(n_i) \cap \text{PostDom}(m_i) \)

end loop

end if

end loop

Theorem 12 (Correctness of Communication Interval Refinement)

The TFG produced by Algorithm 5 conservatively represents the set of executable program event sequences.

Proof:

The first four phases of the algorithm construct the set of communication intervals for \( T_i \) and \( T_j \) according to Definition 19.

The final phase computes for each communication interval, \(((n_i, m_i), (n_j, m_j))\), the set of nodes, \( In_i \), that are inside the interval in \( T_i \) and the set of nodes, \( Out_j \), that are outside the interval in \( T_j \). All nodes in the local interval are locally dominated by the start node and locally post-dominated by the end node. Consider an executable program event sequence containing \( x \in In_i \) and \( y \in Out_j \); such a sequence is of the form \( \ldots L(x) \ldots L(\text{Pred}(m_i)) \ldots L(y) \ldots \) because there is no way for any statement outside the interval in \( T_j \) to execute until the communication event \( \text{Pred}(m_i) \) happens. Consequently a MIP edge \((x, y)\) is an unexecutable
event ordering; a similar argument holds for MIP edge \((y, x)\). An analogous argument allows elimination of MIP edges between nodes in \(In_j\) and \(Out_i\).

The case of a receive node in a MULTIPLE ACCEPT interval is special. In this case the MIP edges from outside the send interval that lead into the receive interval and vice versa are not created. This is because the receiving task may engage with other sending tasks and consequently be inside of its interval while other senders are outside of their intervals. Note, however, that the send inside to receive outside edges are created. Therefore, Algorithm 5 only eliminates unexecutable edges. If the input TFG is conservative then so is the output and the theorem holds.

\[\square\]

**Theorem 13 (Complexity of Communication Interval Refinement)**

*Algorithm 5 is \(O(|N|^4)\) where \(N\) is the set of TFG nodes.*

*Proof:*

We will refer to the number of SEND and RECEIVE statements in task \(T_i\) as \(C_i\). We use \(N_i\) to refer all LOCAL and COM nodes for \(T_i\).

Using well known algorithms [34] we can compute local dominators and post-dominators in \(O(|N_i|^2)\). Checking for domination or post-domination is a bit-vector operation and is "almost constant time". Consequently, phases 2 and 3 are bounded by the cost of considering each pair of SEND and RECEIVE nodes which is \(O(|C_i|^2)\). There are at most \(O(|C_i|^2)\) local intervals in task \(T_i\) and \(O(|C_j|^2)\) local intervals in task \(T_j\). Since finding predecessors, successors, and labels are all \(O(1)\) phase 4 is \(O(|C_i|^2|C_j|^2)\).
In the worst-case, all pairs of local intervals match to form $O(|C_i|^2|C_j|^2)$ communication intervals. For each interval we consider pairs of nodes from $In_i$ and $Out_j$, or $Out_i$ and $In_j$, and delete a MIP edge if it exists; this requires $O(|N_i||N_j|)$ steps.

In total then phase 5 requires $O(|C_i|^2|C_j|^2|N_i||N_j|)$. By construction of the TFG we know that $|N|$ is greater than $|N_i|$ and $|N_j|$. Since communication nodes are formed from the product of communication statements we know that $|N| > |C_i||C_j|$. Thus, the bound on phase 5, which dominates the cost of Algorithm 5, is $O(|N|^4)$.

We note that this is a weak upper bound. In practice, the number of communication statements is a small fraction of the total number of program statements and, for most concurrent programs, the number of statements in a given task is a small fraction of the total number of program statements. In Chapter 5 we provide evidence that for a variety of programs the cost of communication interval refinement is sub-quadratic in the number of TFG nodes. Given the cost of this refinement, we do not want to apply it to all pairs of tasks in a program. We can use a simple $O(|N|)$ traversal of the TFG COM nodes to determine pairs of tasks that actually communicate and are therefore candidates for communication interval refinement.

Groups of Communication Intervals

We can reason about executable sequences of program events by considering groups of related communication intervals. We have identified two classes of interval groups: multi-caller groups and disjoint groups.

**Definition 21** A **multi-caller group** is a collection of communication intervals:

$$\{(n_i, m_i), (n_a, m_a), (n_{i+1}, m_{i+1}), (n_a, m_a), \ldots\}$$
where $\operatorname{Kind}(n_a) = \text{RECEIVE}$ and $\operatorname{Kind}(m_a) = \text{RECEIVE}$. We refer to the local interval $(n_a, m_a)$ as an accepting interval and the local intervals
$\{(n_i, m_i), (n_{i+1}, m_{i+1}), \ldots\}$ as calling intervals.

Note that we could have multiple calling intervals in the same task. Figure 4.16 illustrates a multi-caller group. The accepting interval is in $T_2$ and the calling intervals are in $T_1$ and $T_3$. Communication interval refinement, as described above, can take place between the pairs of tasks $(T_1, T_2)$ and $(T_2, T_3)$. In addition, we can use the semantics of an accepting interval to remove MIP edges $(x, y)$ where $x \in \mathit{In}_1$ and $y \in \mathit{In}_3$. Such an ordering is unexecutable because the structure of the accepting interval dictates that at most a single calling interval can be active at any time. Consequently, any program execution on which $x$ precedes $y$ is represented by a program

\[\text{This example illustrates why local dominators are used for identifying intervals rather than TFG node dominators. The creation of multiple COM nodes for a single RECEIVE statement, e.g., $b_2$, introduces flow graph paths from $\mathit{initial}$ to $\mathit{stop}_{(2,3)}$ that pass through $\mathit{start}_{(1,2)}$; consequently, $\mathit{start}_{(2,3)}$ does not dominate $\mathit{stop}_{(2,3)}$.}\]
event sequence of the form ... $L(x)$ ... $L(\text{stop}_{1,2})$ ... $L(y)$ .... Thus, $L(x)$ cannot immediately precede $L(y)$. An analogous argument suffices for removing edges $(y, x)$. In the example in Figure 4.16 the MIP edges $(b, e)$ and $(e, b)$ would be eliminated.

We can have arbitrarily many calling intervals in a multi-caller group. Since the $In_i$ values are represented by bit-vectors we can efficiently manipulate their values. Performing multi-caller group refinement requires $O(|N||G|^3)$ operations, where $G$ is the interval group.

**Definition 22** A disjoint group is a collection of communication intervals:

$$\{((n_i, m_i), (n_k, m_k)), ((n_{i+1}, m_{i+1}), (n_{k+1}, m_{k+1})), \ldots \}$$

where:

$$\text{Task}(n_i) \neq \text{Task}(n_{i+1})$$

$$\text{Task}(n_k) = \text{Task}(n_{k+1})$$

$$In(n_k) \cap In(n_{k+1}) = \emptyset$$

We refer to the intervals $(n_k, m_k), (n_{k+1}, m_{k+1}), \ldots$ as common intervals since they are common to a single task.

Figure 4.17 illustrates a disjoint group. The common intervals $(b2-l, a2-l)$ and $(b2-r, a2-r)$ are in $T_2$. These intervals are disjoint because of control flow branching at node $\text{if} - \text{lr}$ and the subsequent merge at $\text{endif}$. Communication interval refinement, as described above, can take place between the pairs of tasks $(T_1, T_2)$ and $(T_2, T_3)$. In addition, we can use the semantics of task execution to remove MIP edges that correspond to unexecutable event orderings. Intuitively, once we have taken a branch at $\text{if} - \text{lr}$ in $T_2$ we are committed to executing one or the other of the common intervals. Once inside a common interval we cannot execute inside another common interval, in the disjoint group, until we have exited the current interval. We can perform a refinement that is similar to the one done between $In$ nodes of calling intervals in the multi-caller group case. For disjoint groups, instead of calling intervals we use the
local intervals that are not common. In addition, we can eliminate MIP edges \((x, y)\) where \(x \in In_{2-1}\) and \(y \in In_3\). Such an ordering is unexecutable because the structure of the task dictates that at most a single common interval can be active at any time. Consequently, any program execution on which \(x\) precedes \(y\) is represented by a program event sequence of the form \(\ldots L(x) \ldots L(stop_{1,2}) \ldots L(y) \ldots\). Thus, \(L(x)\) cannot immediately precede \(L(y)\). An analogous argument suffices for eliminating edges \((y, x)\). In the example in Figure 4.17 the MIP edges \((d, e)\) and \((e, d)\) would be eliminated.

As with multi-caller groups, we can have arbitrarily many communication intervals in a disjoint group. Since the \(In_i\) values are represented by bit-vectors we can efficiently manipulate their values. Disjoint group based refinement is dominated by
the $O(|N||G|^3)$ cost of performing the non-common In node refinement; this is analogous to the multi-caller group refinement. In practice, we expect that $|G| << |T|$ where $|T|$ is the number of tasks in the program.

We will see in Chapter 5 that communication interval refinements can have a great effect on the size of the TFG and the cost and accuracy of state propagation analysis. Chapter 7 outlines a number of ideas for additional TFG refinements.

4.6.2 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 encodes a necessary condition that must be satisfied for a path through the TFG to correspond to an executable sequence of program events. We can think of FCs as unfolding selected control or data information that was previously folded in the TFG; in this way we can recover information that may increase the precision of analysis results.

We have developed feasibility constraints that encode necessary conditions related to the ordering of events local to a task and to the use of program variables; each of these can be encoded as a finite state automaton. A collection of these FC automata can be combined with the property automaton to form a constrained property automaton (CPA) that can be input to the state propagation algorithm. The CPA enforces the conjunction of the necessary conditions encoded in the feasibility constraints during state propagation analysis. A conjunction of necessary conditions is itself a necessary condition that is at least as strong as any of the conjuncts in isolation. In practice, it is often the case that the conjunction is significantly stronger than the conjuncts in isolation. Thus, a CPA that incorporates multiple feasibility constraints has the potential to increase the precision of analysis results.
In the remainder of this section we describe two classes of feasibility constraints in detail: task automata and variable automata. We also discuss the construction of the CPA. This serves to illustrate the intuition and details of feasibility constraints, but, it only begins to touch on the potential benefits to the precision of analysis. There are a wide variety of necessary conditions that could be used to constrain the state propagation algorithm and potentially improve the precision of analysis results. In Chapter 7, we describe a disciplined approach to exploring the possible conditions that may be of use.

We will see in Chapter 5 that incorporating feasibility constraints into FLAVERS can increase precision of state propagation analysis.

4.6.2.1 Task Automata

As illustrated in Figure 4.14 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. The refinements of the previous section help improve the precision of the TFG by eliminating unnecessary MIP edges. An alternate approach is to enforce, throughout the entire flow graph, the control flow orderings for a single task during state propagation. We do this by encoding the state and state transitions of an individual task as a finite state 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 executable event sequences for the modeled task.

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 control flow edges in $T_i$ that are incident upon those nodes. The initial and final nodes of the sub-flow-graph determine the start and accept states of the automaton. Formally,

**Definition 23** A task automaton is a deterministic finite-state automaton $(S_{TA}, \delta_{TA}, A_{TA}, s_{TA}, \Sigma_{TA})$ where:

131
In addition to a state for each node, \( n_i \), in the task sub-flow-graph, we add a non-accepting violation state, \( v \), to represent that a path in the TFG violates the ordering of events in this task. In general, program event symbols do not uniquely label TFG edges. Keeping track of task state in the TA requires that we know when each instance of an event occurs, rather than just recording that some instance occurred. To do this, we construct a TA alphabet that consists of the concatenation of the node label with the unique node id, for each node in the task sub-flow-graph. Transitions are defined for all control flow edges in the task sub-flow-graph. In addition, we add transitions from all TA states to the violation state for all symbols in the TA alphabet for which the state has no defined transition. This causes the TA to enter the violation state if a sequence of program events that violates the control flow structure of this task is encountered. Figure 4.18 illustrates a fragment of a TFG and the TA for task \( T_1 \).

Each node in the task sub-flow-graph of the TFG has a corresponding symbol in the TA; thus symbol \( a : 1 \) represents the instance of event \( a \) at node 1 in the TFG. There are four states: violation, \( v \), and one for each incoming edge to \( T_1 \)'s sub-flow-graph nodes. The start state is associated with the initial node of the task, i.e., the successor of \( n_{\text{initial}} \), for this example it is state 1. The state associated with the end node of the task, i.e., the predecessor of \( n_{\text{final}} \), is state 3; it is the only accepting state.

**Algorithm 6 (Task Automaton Construction)**

*Input:*
A TFG and a task $T_i$.

Output:

A task automaton for $T_i$.

Algorithm:

As we traverse the sub-flow-graph for $T_i$ we create TA states and transitions to correspond to TFG nodes and edges. We record the TA state created for a node, $n$, as $State(n)$. Intuitively, $State(n)$ represents the control state of $T_i$ immediately after execution of the statement at node $n$. Note that an additional state $v$ is introduced as the violation state.

1. for each node, $n$, where Task($n$) = $T_i$, loop
2. $\Sigma_{TA} = \Sigma_{TA} \cup "L(n) : Id(n)"$
   end loop
3. create $|N_i| + 2$ states
4. for each state, $s \in S_{TA}$, loop
5. for each symbol, $a \in \Sigma_{TA}$, loop
\[(6) \quad \delta_{TA}(a, s) = v\]

end loop

\[(7) \quad \delta_{TA}(\tau, s) = s\]

end loop

\[(8) \quad \text{depth-first walk using CF edges in } T_i \text{ from } n_{initial}\]

\[(9) \quad c = \text{current node}\]

\[(10) \quad \text{for each edge, } (c, m) \in E \land Task((c, m)) = T_i, \text{ loop}\]

\[(11) \quad \delta_{TA}("I(m) : Id(m)", \text{State}(c)) = \text{State}(m)\]

end loop

end loop

\[(12) \text{State}(n_{initial}) \text{ is the start state}\]

\[(13) \text{for all nodes, } (n, n_{final}) \in E\text{edge } Task(n) = T_i, \text{ loop}\]

\[(14) \text{State}(n) \text{ is an accepting state}\]

end loop

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. We discuss the introduction of parallel transitions in Section 4.6.2.3.

**Theorem 14 (Correctness of TA)**

*If a sequence of program events leads to a TA violation state then it does not correspond to a program execution.*

*Proof:*
Call the modeled task \( T_i \). Consider any sequence, \( s \), of program events over the symbols \( \Sigma \cup \Sigma_{TA} \). This sequence is of the form, \( s = a_1, a_2, a_3, \ldots, a_k \).

We need to restrict the events in the sequence to be only those that correspond to elements of \( \Sigma_{TA} \). We take the projection of \( s \) onto \( \Sigma_{TA} \).

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 CF sub-flow-graph for \( T_i \). Recall that a symbol \( a \in \Sigma_{TA} \) can be written as the concatenation of two components "\( L(n) : Id(n) \)."

Assume that \( \delta^*_T(s, s_{TA}) = v \).

In this case \( \exists a \in \Sigma_{TA} : s = \alpha_1 a \alpha_2 \land \delta^*_T(\alpha_1, s_{TA}) \neq v \land \delta^*_T(a, \delta^*_T(\alpha_1, s_{TA})) = v \). By construction of the TA this can only happen if the node corresponding to \( \delta^*_T(\alpha_1, s_{TA}) \), i.e., the \( n \) where \( State(n) = \delta^*_T(\alpha_1, s_{TA}) \), 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.

\[ \square \]

To construct a TA we need only consider a small sub-graph of the TFG; consequently, the cost of construction is low.

**Theorem 15 (Complexity of TA Construction)**

Algorithm 6 requires \( O(|N_i|^2) \) steps where \( N_i \in N \) is the set of nodes in \( T_i \).

**Proof:**

It is easy to implement a mechanism for accessing nodes in \( T_i \) without touching all nodes in the TFG. Under this assumption we have the following run-time bounds.

\[ \text{such a sequence corresponds to a path, or collection of paths, from a TFG that has had the labels of nodes in } T_i \text{ transformed as described above.} \]
Statement 12 is $O(1)$. Loop 1 and statements 3 and 14 are $O(|N_i|)$. Loop 4, consisting of statements 4-7, is $O(|N_i||\Sigma_{TA}|)$. In the worst-case, all nodes in $T_i$ have edges to all other nodes in $T_i$; therefore, loop 8, consisting of statements 8-11, is $O(|N_i|^2)$. Since, $\Sigma_{TA}$ is constructed using the node ids of nodes in $T_i$, we have that $\Sigma_{TA} = O(|N_i|)$. Thus the entire algorithm is $O(|N_i|^2)$.

\[ \square \]

In Chapter 5, we will see that constructing TAs is very fast in practice.

4.6.2.2 Variable Automata

Flow graphs do not typically model program variables, however, for many programs accurate analysis depends on modeling some critical program variables. We are most interested in variable that are used to determine control flow branching and in guards for communication statements. Many of these state variables are defined over small finite domains and modified in a disciplined way. Examples include boolean variables to which only constant values are assigned and bounded 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 a finite automaton called a variable automaton (VA). Formally,

**Definition 24** A variable automaton is a deterministic finite-state automaton $(S_{VA}, \delta_{VA}, A_{VA}, s_{VA}, \Sigma_{VA})$, where:

\[
S_{VA} = \{s_{VA}, v_1, v_2, \ldots, v_k, v\}
\]

\[
\delta_{VA} \mid S_{VA} \times \Sigma_{VA} \rightarrow S_{VA}
\]

\[
A_{VA} \subseteq S_{VA}
\]
Figure 4.19 VA for Boolean Variable

\[ s_{VA} \in S_{VA} \]

\[ \Sigma_{VA} = \{ " := u", " := x", " = x?", \ldots \} \]

where \( x \) is a possible value of the modeled variable.

A VA consists of accepting states that model each of the \( k \) possible values, the \( v_i \), that a variable can take on during program execution; this includes a start state which represents that the variable is undefined. We introduce a non-accepting violation state, \( v \), to represent that a path in the TFG violates the semantics of a program branch decision with respect to the current variable state. VA transitions represent modifications of the variable and the results of tests of the variable used in branch decisions. When the result of a test is inconsistent with some value of the variable described by a VA state we introduce a transition to the violation state on that test result. Finally, all states of the VA are accepting except for the violation state.

All boolean variables have VAs that are identical in structure. Figure 4.19 illustrates the a generic VA for a boolean variable. There are four states: undefined \( u \), violation \( v \), true \( t \) and false \( f \). The modifications of the variable are modeled as \( x := t \)
and \( x:=F \); we include \( x:=u \) to indicate that the value of the variable is not known. The test results are \( x=T \) and \( x=F \). 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. For a specific boolean variable we replace \( z \) with the name of the variable. Figure 4.20 illustrates a segment of code and a portion of a TFG that represents the possible executable event sequences exhibited by that code. The TFG only models some of the details of the program statements. To model the

![Diagram](image)

**Figure 4.20** TFG Fragment with VA Symbols

boolean variable \( ok \) we insert separate test result symbols and assignment symbols. The assignment symbols are used to keep track of the current value of the variable.
The test result symbols are used to determine if a test, with the given result, of the current variable value was feasible. The semantics of control flow branching, along with information about the branch condition, allow us to infer values of ok along program paths even when no constants have been assigned ok. Instantiation of a boolean VA for ok and incorporating it in state propagation analysis would allow us to determine that it is never the case that dummy precedes store without an intervening read.

We have developed algorithms to construct variable automata for both boolean variables and bounded counter variables. We can view both of these as instantiating a generic template and replacing the variable names in the template with a program variable name.

Algorithm 7 (Boolean Variable Automaton Construction)

\textit{Input:}

A TFG and a variable \( x \).

\textit{Output:}

A variable automaton for \( x \).

\textit{Algorithm:}

1. \( \Sigma_{VA} = \{ \tau, "x:=u", "x:=T", "x:=F", "x=T?", "x=F?" \} \)
2. \textit{create states unknown, true, false, and violation}
3. \( s_{VA} = \text{unknown} \)
4. \( A_{VA} = \{ \text{true, false, unknown} \} \)
5. \textit{create } \delta_{VA} \text{ as depicted in Figure 4.19}

\textbf{Theorem 16 (Correctness of Boolean VA)}

If a sequence of program events leads to a VA violation state then it does not correspond to a program execution.
Proof:

The TFG is conservative with respect to including all executable sequences of events. This overestimating of the sequences of executable events could cause it to include a sequence of variable modification and test result symbols that violate the programming language semantics. This is precisely the situation that variable automata are designed to remedy.

Note that symbols related to a variable in the TFG are conservative, since we label a node with "$x := F$" ("$x := T$") only if we are guaranteed that after executing the node the value of $x$ will be false(true), otherwise we label it with "$x := u$".

There are only two ways that we can get to a VA violation state: being in state true and encountering an "$= F$" test result or being in state false and encountering an "$= T$" test result. If a boolean variable has the value true and we reach a branch point that tests if the variable has the value false the program will take the false branch. Taking the true branch at this point is in violation of program semantics and hence is unexecutable. Thus, entry to a VA violation state corresponds to an unexecutable sequence of program events.

\[ \square \]

Theorem 17 (Complexity of Boolean VA Construction)

Algorithm 7 requires $O(1)$ steps.

Proof:

Immediate, since every line in Algorithm 7 requires $O(1)$ steps.
Bounded counter variable automata are constructed in a similar manner in $O(k)$ steps where $k$ is the maximum value of the counter.

As discussed above, 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 the CPA depends on the ability to identify program statements that cause variable state transitions. In many cases this can be done statically; when that is not possible a conservative, and less precise, approximation must be used. Sophisticated analysis of the usage pattern of program variables may be able to statically discover variable state transitions that are not explicit in the program text. In fact, variables that are defined over very large domains, but only range over a small statically determinable domain can be identified and their states and state transitions encoded in a VA. The extent to which such patterns of access appear in real concurrent programs and provide opportunities to improve the precision of analysis remains to be seen.

4.6.2.3 Constrained Property Automata

Given a PA and a collection of 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. Since the cost of state propagation analysis is linear in the number of states, that cost may also grow rapidly with the number of FCs. We have developed techniques that allow us to reduce the number of states and the number of symbols in the alphabet of the CPA. We begin with the definition of a CPA.
Definition 25 A constrained property automaton is a deterministic finite-state automaton \((S_{CPA}, \delta_{CPA}, A_{CPA}, s_{CPA}, \Sigma_{CPA})\), where:

\[
S_{CPA} \subseteq S_{PA} \times S_{FC_1} \times \ldots \times S_{FC_k}
\]

\[
\delta_{CPA} : S_{CPA} \times \Sigma_{TA} \rightarrow S_{CPA}
\]

\[
A_{CPA} = \{(s_1, s_2, \ldots, s_k) : s_1 \in A_{PA} \land s_2 \in A_{FC_1} \land \ldots \land s_k \in A_{FC_k}\}
\]

\[
s_{CPA} = (s_1, s_2, \ldots, s_k) \text{ where } s_1 = s_{PA} \land s_2 = s_{FC_1} \land \ldots \land s_k = s_{FC_k}
\]

\[
\Sigma_{CPA} = \Sigma_{property} \cup \bigcup_{i=1,\ldots,k} \Sigma_{FC_i}
\]

The states of the CPA are a subset of the cross-product of the PA and FC states. We only require a subset because a number of syntactic states are unreachable or equivalent to other states. CPA states that are unreachable are unnecessary and inflate the representation of \(\delta\). When transitions in multiple FC automata or the PA have a common label the CPA may contain states that are unreachable. These states are easily detected and removed from the CPA via a \(O(|N|)\) sink algorithm. We can collapse 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 precision. More generally, we can apply a partition-refinement state minimization algorithm that in time \(O(|N| \log(|N|))\) would identify and collapse equivalent states. To do this, we need to take care in specifying the initial partition; in particular, all FC violation states should be in a separate partition.

We could use the cross-product of the PA and FC alphabets as the CPA alphabet. This alphabet would grow rapidly with the number of constraints and \(\delta\) would quickly become impractical to implement or evaluate. We can significantly reduce the size
of the CPA alphabet by recognizing that all FC automata alphabets and the PA alphabet are either subset of or variants of $\Sigma$. Thus, symbols found in the PA or one of the FC alphabets are represented by a single symbol, $a$, rather than the element of the cross-product alphabet, e.g., $(a, a, \ldots, a)$. Relative to the cross-product alphabet this reduced alphabet is significantly smaller.

The start and accepting states are defined as the component-wise start and accepting states.

The CPA transition function $\delta_{CPA}$ is the component-wise transition function. The definition of $\Sigma_{CPA}$, however, makes it possible that a symbol not in one of the component alphabets could be submitted to that component's state transition function. To avoid this problem, we extend the component transition functions to include self-loop transitions on all symbols that are not in the component's alphabet. Note that this is only done for the purpose of defining $\delta_{CPA}$ and does not require modification of $\delta_{PA}$ or the $\delta_{FC_i}$.

The fundamentally exponential nature of the CPA is unavoidable. The techniques described above attempt to reduce the size of the CPA; ultimately, they make only a small contribution. More important is the flexibility of the CPA to incorporate a variety of FCs. Using this capability allows FLAVERS to optimize the analysis for both precision and cost, by including only those constraints that are necessary for precise analysis. We discuss this in more detail in Chapter 5.

As mentioned in Section 4.6.2.1 the alphabet of a TA needs to match that of the TFG in order for the constraint to operate correctly. When a TA is included in the CPA the alphabet $\Sigma_{CPA}$ will include all of the symbols in $\Sigma_{TA}$. While this is sufficient for operation of the TA constraint it interferes with the operation of the PA. A TA symbol "$a : 5"$ provides the same information to the PA as the symbol $a$; that an occurrence of program event labeled $a$ has occurred. Unlike the TA, the PA is not concerned with which instances of an event occurs, but, the symbols $a$ and "$a : 5"$
are distinct. We treat them identically in the PA by introducing parallel transitions prior to constructing the CPA.

**Algorithm 8 (Constructing Parallel PA Transitions)**

*Input:*

A PA and a TA.

*Output:*

A PA with parallel transitions for TA symbols and alphabet \( \Sigma_{\text{new}} \).

*Algorithm:*

We use the symbol \# to indicate the node id component of a TA alphabet symbol.

1. \( \Sigma_{\text{new}} = \Sigma_{\text{property}} \cup \Sigma_{\text{TA}} \)
2. *for each symbol, \( a : \# \in \Sigma_{\text{TA}}, \) loop*
3. *for each state, \( s \in S_{\text{PA}}, \) loop*
4. *if \( a \in \Sigma_{\text{property}} \) then*
5. \( \delta(a : \#, s) = \delta(a, s) \)
6. *else*
7. \( \delta(a : \#, s) = s \)
8. *end if*
9. *end loop*
10. *end loop*

Note that if alphabet refinement has been performed then lines 4 and 6 are unnecessary. In this case, the algorithm is readily seen to be \( O(|N_T||\Sigma_{\text{property}}|) \) where the number of nodes, \( |N_T| \), in the task modeled by the TA bounds the size of its alphabet.

In Chapter 5 we show that the techniques described above allow practical construction of CPAs and use of CPAs in state propagation.
Interpreting State Propagation Results

Interpreting the results of state propagation of a CPA, as opposed to a PA, has one slight difference from the description in Section 4.2. 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.

4.6.3 Combining Results

Just as the CPA combines constraints to enforce stronger conditions and improve the precision of state propagation analysis, we can combine the results of separate state propagation analyses to improve precision.

We can construct the reverse of the TFG, PA, FC automata and CPA and reason about reverse program executions. Combining the results of forward and backward analyses has the potential to improve the overall precision of analysis at the cost of doubling analysis time.

For some complex properties we may be able to extract necessary sub-properties, such that, if a program fails to satisfy the sub-property it cannot satisfy the original property. We can perform state propagation analyses of these sub-properties and decide whether to perform the more costly analysis of the complex property based on the preliminary results. For example, we may want to show that "a; b; c" is not possible on any program execution. We can decompose this into two sub-properties a; b and b; c such that if either of these is impossible then the original property is impossible.

We view testing and analysis as an ongoing process for any piece of software. As such we expect that a large collection of properties may need to be checked for a given problem. If two properties are found to have conclusive results that is equivalent to a conclusive result for their conjunction. Another way of taking advantage of existing
analysis results is to use the PA for a property that was satisfied as a feasibility constraint in the checking of another property. This constrains the set of event sequences considered to be those that satisfy the PA. We have found programs where analysis of property $A$ required a PA and some FCs and analysis of property $B$ required a PA and some of the same FCs as were used in the analysis of $A$. By using the PA for $A$ as an FC in the analysis of $B$ we were able to eliminate a number of FCs from that analysis and reduce its cost without loss of precision.

This part of FLAVERS is relatively immature. We discuss our plans for combining analysis results in Chapter 7.

4.7 Summary

In this section, we developed a flow-graph model of execution for concurrent programs, we presented a formalism and representation for describing patterns of executable program behavior and developed an analysis algorithm for checking whether a program satisfies that behavior. This was all done in the context of a flexible architecture called FLAVERS. We discussed the motivation and principles behind a variety of mechanisms for both reducing the cost of analysis and increasing the precision of analysis results. This included detailed description of a number of example mechanisms. For each of these concepts, we presented a rigorous definition, and reasoned about the correctness and complexity of constructing and manipulating the associated artifacts. The next step is to evaluate these concepts in practice.
In Chapter 4 we presented, FLAVERS, a family of polynomial-time analysis techniques for concurrent programs. The polynomial bounds on the running time of these analyses suggests that FLAVERS may scale better than the exponentially bounded approaches described in Chapter 2. The bounds, however, are worst-case and say nothing about what we might expect for the cost of analysis on typical applications. In fact, there are well known best algorithms that are rarely used due to excessive overhead costs that render them impractical for the majority of problems encountered in practice. To develop an understanding of the feasibility and practicality of FLAVERS we need to try it on realistic analysis problems.

In this section, we describe our experience applying the FLAVERS analysis technique to a variety of programs and properties of those programs. The goal of this section is to demonstrate that for selected interesting properties of non-trivial concurrent programs, FLAVERS can produce precise analysis results in a practical amount of time.

There are two important components to this demonstration: cost and precision. We need to evaluate the cost of constructing all of the analysis artifacts and performing the analyses. We gather information about the rate of growth of analysis cost by varying the size of programs that are scalable, i.e., for which we can easily increase the size of the program. We could choose to look at a number of small programs and extrapolate the cost of analysis for medium or large size programs; instead we analyze programs that have a realistic number of tasks. Given that FLAVERS is is fundamen-
tally approximate, as are all conservative analyses, one natural question is whether FLAVERS can produce precise analysis results. FLAVERS, unlike many other static analysis techniques, is designed with the idea that the precision of analysis results can be improved. The evaluation in this section illustrates the mechanisms for this improvement; it also begins to give a sense of the amount of additional information that is required for high-precision and the effect of that information on analysis cost.

This is not a broad experimental study of the cost-effectiveness of FLAVERS. In Section 7.1, we argue that while difficult to design and conduct, such a study is an important component of a thorough evaluation of FLAVERS, or any other analysis technique. Thus, the results of this section should be viewed as preliminary and care should be taken in generalizing them.

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, we follow up with a number of observations about the results of the evaluation and conclude with a distillation of our experience into a strategy for applying FLAVERS to new analysis problems.

5.1 Methodology

Our empirical evaluation was performed using an implementation of FLAVERS that is targeted to Ada tasking programs. We call this FLAVERS/Ada; it is described in Appendix A.

We have selected a number of Ada tasking programs from the concurrency analysis literature. Some of these programs have been well studied and a variety of static analysis techniques have been applied to them; others have been introduced more recently. We have chosen a mixture of programs including scalable programs with lots of replicated tasks, scalable programs with many dissimilar tasks and non-scalable programs with dissimilar tasks. For some of these programs there are well-known
natural properties to be checked; for others we developed specifications by reverse engineering the application. The goal was to come up with properties that might be checked by a developer of the application.

For a given Ada program and a property specified as a QRE we perform the following steps:

1. For scalable programs choose the smallest instance of the program.
2. Determine the truth or falsity of the property with respect to the program.
3. Annotate the program with events of interest.
4. Apply the basic FLAVERS/Ada analysis.
5. If the results are inconclusive, then apply refinements/feasibility constraints until the results are conclusive or we exceed tool capacity.
6. If the results are conclusive then increase the size of program and run the analysis with minimal cost refinements/feasibility constraints.

Step 1 reduces the cost of steps 2-5. Step 2 requires significant human intervention. We reason about the satisfaction of a property by hand, often using exhaustive reasoning techniques. In step 3 we need to mark the program events that are included in the QRE specification of the property; in the current implementation of FLAVERS/Ada this is done by inserting comments into the source code. Step 4 involves: constructing the TFG from the Ada source, constructing the PA from the QRE, and applying state propagation analysis to the TFG and PA. Each of these steps is supported by a set of FLAVERS/Ada tools. In step 5 we have the problem of diagnosing the source of an inconclusive result; we discuss strategies for doing this in Section 5.4. If we determine that the result was spurious we may select a (set of) refinement(s) or feasibility constraint(s) to apply; FLAVERS/Ada provides tool support for alphabet refinement, communication interval refinement, variable automata and task automata. In step 6
we stress FLAVERS by increasing the size of the program under analysis; in some cases there is a concomitant increase in the size of the property to be checked.

We call steps 1-5 an analysis series. In the detailed sections that follow we present data on a number of analysis series. In each of these cases, we provide both analysis cost and precision information. We stop the series when we find an instance of a FLAVERS/Ada analysis that results in no spurious results, i.e., the results are conclusive. This may require combinations of refinements and FCs.

Our primary measure of analysis cost is the sum of user and system time as measured by `/bin/time` on a 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).

Measurements were made on normally loaded public workstations. To simulate the use of FLAVERS/Ada in practice, we ran a series of appropriate tools from TFG construction through state propagation analysis. One implication of this is that the individual costs of each tool may be greater than if we had run each tool a number of times consecutively. We ran each analysis three times and took the average of those times; a more thorough statistical analysis would take into account the effects of timing variation by computing confidence intervals. The run-times include all overhead costs related to the repository and object management infrastructure on which the tools are built. No attempt was made to factor out the cost of infrastructure components.

Additional measurements of the cost of the state propagation algorithm were made by instrumenting the data flow analysis algorithm. The algorithm is an iterative worklist algorithm; we record the number of nodes that are taken off of the worklist and processed, the number of times the meet (or join) operation is applied to compute flow into a node and the number of times that $\delta$, the PA state transition
function, is evaluated. These form a platform independent measure of the cost of state propagation analysis.

To reason about the rate of growth of the cost of analysis we plot the measures of analysis cost 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 cost measure 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. This approach relies on the fact that the data will lie on a line in log-log space. Unfortunately, the data are gathered from real program executions and do not lie on a perfect line. Furthermore, for some of the analyses described in this section both the TFG and the PA increase in size as the program is scaled; in this case, the growth of analysis cost is a function of 2 variables. Our approach, in this case, is to restrict our claims of rate of growth to the data in our experiments. We say that, for example, "for all of the data in a plot the slope between any pair of points is sub-cubic in the number of TFG nodes".

Additional data for each of the analysis problems considered in the rest of this section is given in Appendix B. In particular, for each analysis problem we provide all of the raw empirical data and plots for the rate of growth of each phase of analysis and for the platform independent measures of cost. These last plots illustrate an interesting feature of the data. The points for each of the platform independent measures lie very close to a line in the rate-of-growth plot; analysis time, however, has a slightly cupped shape. This suggests exponential growth in the analysis. The discrepancy between these two could be a result of insufficient data (the cupped shape straightens), or super-polynomial behavior in either the infrastructure components or some uninstrumented component of the FLAVERS/Ada tools. A detailed analysis of the performance of the toolset is warranted to isolate the source of the cost increases.
In the descriptions of our empirical results, we use the following terminology in referring to analysis artifacts and phases of the analysis:

**PA** A property automaton, measured in states.

**TA** A task automaton, measured in states. The phase which constructs a TA from a TFG measured in time.

**VA** A variable automaton, measured in states. The phase which constructs a VA from a description of the variable type measured in time.

**CPA** A constrained property automaton measured in states. The phase which constructs a CPA from a PA, TAs and VAs measured in time.

**TFG** A trace flow graph, measured in nodes and edges.

**LP** The phase which constructs a TFG from an Ada source program, measured in time. This involves parsing, semantic analysis, control flow analysis and finally TFG construction. As mentioned in Appendix A, the TFG is only partially constructed here as we delay the construction of MIP edges until after refinements have been applied.

**QRE** The phase which constructs a PA from a QRE specification, measured in time.

**ALPHA** The phase which performs alphabet refinement, measured in time.

**CI** The phase which performs communication interval refinement, measured in time.

**UNIFY** The phase which insures that CPA and TFG alphabets are consistent, measured in time. This is required when the CPA includes TAs.

**MIP** The phase which computes and adds MIP edges to the CFG, measured in time.

**PROPS** The phase which performs state propagation analysis, measured in time.
One implication of this is that the total cost of constructing the TFG is given by the sum of LP and MIP.

We note that the problem of validating the correctness of FLAVERS/Ada analysis is not discussed in this section. Through a combination of detailed hand analysis of small instances of programs and seeding faults in the programs we have gained a high degree of confidence in the conservativeness of FLAVERS/Ada analyses.

5.2 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 Appendix C. For all of the experiments with scalable programs, we started with the smallest 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 tools were able to handle that size problem.

5.2.1 A Simple Protocol Problem

This simple protocol is a program that illustrates a common mechanism for resource management: exclusive locks. Corbett [15] presents it to illustrate capabilities of the constrained expressions (INCA) toolset. We use a slightly modified version of that code, which was not a legal Ada program.

The system consists of at least 3 tasks; it is scalable. The protocol consists of a collection of client tasks that attempt to gain access to a resource. Once a task holds it, 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 the number of
clients. The 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 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 explore a simple and restricted version of such a specification. The QRE header-packet for client 1 of a 2 client version of the protocol is:

$$\{h1, p1, h2, p2\}
\begin{align*}
\text{all} \\
[-h1]^*;(h1;[-p1,h1,h2]^*;p1;[-h1]^*)^*
\end{align*}
$$

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 apply the steps of our methodology to explore the effects of different refinements and feasibility constraints on the precision and cost of analysis of this property; recall that our goal is to gain high precision for minimal cost.

We perform seven different variations of FLAVERS analyses: basic, ALPHA, ALPHA+CI, ALPHA+TA, ALPHA+2TAs, ALPHA+3TAs, and ALPHA+allTAs. Table 5.2.1 gives data on the sizes of the artifacts produced during these various analyses. The CPAs are produced as follows:

$$CPA1 = PA \ast TA_{C1}$$
$$CPA2 = PA \ast TA_{C1} \ast TA_{C2}$$
$$CPA3 = PA \ast TA_{C1} \ast TA_{C2} \ast TA_{LM}$$
$$CPA4 = PA \ast TA_{C1} \ast TA_{C2} \ast TA_{LM} \ast TA_{Ch}$$
Table 5.1 Artifact Sizes for header-packet Analyses

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Nodes/States</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFG</td>
<td>43</td>
<td>254</td>
</tr>
<tr>
<td>Alpha TFG</td>
<td>23</td>
<td>82</td>
</tr>
<tr>
<td>CI Alpha TFG</td>
<td>26</td>
<td>69</td>
</tr>
<tr>
<td>PA</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Client1 TA</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Client2 TA</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>LockManager TA</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Channel TA</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>CPA1</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>CPA2</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>CPA3</td>
<td>677</td>
<td></td>
</tr>
<tr>
<td>CPA4</td>
<td>677</td>
<td></td>
</tr>
</tbody>
</table>

where $TA_{Ci}$ is the TA for Client$_i$, $TA_{LM}$ is the TA for the LockManager task, and $TA_{Ch}$ is the TA for the Channel task. We note that the Alpha TFG has 3 less states than the CI Alpha TFG; this is because if communication interval refinement is going to be run subsequent to alphabet refinement, then certain structural properties of the TFG must be maintained. This restricts some of the alphabet refinement transformations that can be performed.

In Table 5.2.1 we present information on the execution performance of the seven analyses. Each column represents a phase of analysis; if a column is blank then that phase is not required for the given analysis. It is interesting to note that incorporating the Channel TA did not increase the number of CPA states, however, it changed the structure of the CPA so as to double the cost of state propagation. This underscores the point that the size of an artifact is only part of what determines the cost of analysis, the interaction of the structure of the artifacts also has a significant influence. We note that the cost of computing TAs for Client1 and Client2 should be identical, however, we always found that the cost of constructing the second TA was less than
the first. This is because the FLAVERS/Ada tool that constructs the first TA has been cached when we invoke that same tool on the second TA; consequently, tool startup cost is reduced. There are a variety of ways to obtain a FLAVERS analysis that completely eliminates spurious results for this problem; these analyses vary widely in solution cost. The problem of selecting the cheapest such analysis, before trying them, is difficult. We discuss heuristics for making such selections in Section 5.4. State propagation grows, at worst, linearly with the number of CPA states. In Figure 5.1, we plot the growth of both state propagation and total analysis time, for the analysis in the series, versus the size of the CPA; as expected state propagation has a slope less than 1.

The goal of running this series of analyses is to determine the lowest cost method of obtaining conclusive analysis results. We found three analyses that obtain conclusive results and the cheapest, by an order of magnitude, was Alpha+CI. In analyses of successively larger versions of the protocol program we will apply Alpha+CI as our

<table>
<thead>
<tr>
<th>Phase</th>
<th>Basic</th>
<th>Alpha</th>
<th>Alpha CI</th>
<th>Alpha TA</th>
<th>Alpha 2 TA</th>
<th>Alpha 3 TA</th>
<th>Alpha All TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>17.7</td>
<td>17.8</td>
<td>17.8</td>
<td>17.7</td>
<td>17.7</td>
<td>17.8</td>
<td>17.7</td>
</tr>
<tr>
<td>QRE</td>
<td>0.8</td>
<td>0.8</td>
<td>0.9</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>ALPHA</td>
<td>4.5</td>
<td>4.4</td>
<td>4.4</td>
<td>4.6</td>
<td>4.5</td>
<td>4.5</td>
<td>4.5</td>
</tr>
<tr>
<td>CI</td>
<td></td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1 TA</td>
<td></td>
<td></td>
<td>5.4</td>
<td>5.5</td>
<td>5.5</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>C2 TA</td>
<td></td>
<td></td>
<td></td>
<td>5.1</td>
<td>5.0</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>LM TA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.1</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>Ch TA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>CPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.1</td>
</tr>
<tr>
<td>UNIFY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.2</td>
</tr>
<tr>
<td>MIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.8</td>
</tr>
<tr>
<td>PROPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.7</td>
</tr>
<tr>
<td>Total</td>
<td>29.0</td>
<td>32.1</td>
<td>36.1</td>
<td>48.3</td>
<td>99.1</td>
<td>456.4</td>
<td>1027.5</td>
</tr>
<tr>
<td>Result</td>
<td>inc</td>
<td>inc</td>
<td>con</td>
<td>inc</td>
<td>inc</td>
<td>con</td>
<td>con</td>
</tr>
</tbody>
</table>
Figure 5.1 Analysis Cost Versus CPA Size

set of refinements. We note that all of the results that follow are for analyses that produce conclusive analysis results.

We now consider how the cost of analysis scales as we increase the number of Client tasks in the protocol program. We consider three variants of the header-packet property: the single task version described above, multiple application of single task versions, 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 composite version is an iterated disjunction of each of these individual single task versions.

We start with the single task property. Figure 5.2 plots total analysis time versus the number of nodes in the TFG. This is a rate-of-growth plot; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. We can see that the rate of growth of the Alpha+CI 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 relatively more local properties. 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; only the QRE construction and state propagation algorithms need to be executed for each Client task of the program. Figure 5.3 is the rate-of-growth plot for analysis via multiple header-packet analyses. We include an additional reference line whose slope is $T \times N^3$ 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 $T \times N^3$. 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 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 impractically large for a specification, it turns out that the property is very regular and could easily be produced from a simple macro. In Chapter 7, we discuss mechanisms for making it easier to write QREs. 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 5.4 is the rate-of-growth plot for analysis of the composite property. We include an additional reference line
Figure 5.4 Total Analysis Time for Composite header-packet

whose slope is $T \times N^3$ where $T$ is the number of tasks; this is to account for the fact that the property is growing linearly with the number of program tasks. We see from this plot that the cost of analysis appears to grow faster than $N^3$ but less than $T \times N^3$. 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. In Figure 5.5 we plot rates of growth of the multiple and composite approaches together. While the multiple analysis 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 analysis 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 even against a best case multiple analysis.

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
Figure 5.5 Total Analysis Time for Composite and Multiple header-packet

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:

\[
\{ h1, p1, h2, p2 \} \\
\text{none} \\
[-h1]^* ; p1 ; * 
\]

This is weaker, as conceivably the error could appear only on subsequent packets. Unlike the previous properties, this one specifies that we insure that no program execution satisfies the specified pattern of behavior. Based on the success of Alpha+CI in producing conclusive analysis results for other properties related to header and packet events, we tried that combination of refinements for this property; the analysis results were conclusive. This illustrates one of the ways in which past experience in analyzing a program can benefit new, as yet untried, analyses.
In Figure 5.6 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.

5.2.2 The Readers/Writers Problem

The readers/writers problem captures a common paradigm in concurrent applications. Isolation of shared data and the control of access to that data is a natural extension of sequential abstraction techniques to concurrent programs.

A number of researchers have analyzed versions of the readers/writers problem [4, 23]; Typically, analysis is done for deadlock freedom or observance of mutual exclusion policies. The state space of the program grow exponentially with the number of tasks. A state of the program records the states of the readers and writers, assuming they can only be in one of two states we still end up with $2^{r+w}$, 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 a naive reachability analysis.

The program consists of at least 3 tasks; it is scalable. The readers/writers problem consists of a central data server, also called the control task, and a collection of reader and writer 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 [16] 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 applications, 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 program 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.

We check two properties related to exclusive write semantics. Initially, we intended to first check that writes were exclusive with respect to one another, then extend the analysis to reason about writes excluding reads. Our intuition is that the less a property says the easier it will be to prove; this example illustrates that our intuition is a bit pessimistic.

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 \texttt{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 \texttt{rw.control.start\_write}, \texttt{rw.control.stop\_write}, \texttt{rw.control.start\_read} and \texttt{rw.control.stop\_read}. These are the fully-qualified
Ada names of the control task entries. We use the abbreviations \texttt{wstart, wstop, rstart} and \texttt{rstop}, respectively, to make the QREs more manageable.

The QRE for the \texttt{exclusive-write} property is:

\[
\{ \texttt{wstart, wstop} \} \\
\text{all} \\
[-\texttt{wstart}]^*; (\texttt{wstart};[-\texttt{wstart}, \texttt{wstop}]^*; \texttt{wstop};[-\texttt{wstart}]^*)^*
\]

It says that, if we see the start of a write then no other write start can occur before a stop write occurs. This is the essence of mutual exclusion. Note that this property says nothing about the correspondence between start and stop events; we may want to insure that the stop write comes from the writer that is currently active. This requires a different QRE that is similar to the \texttt{header-packet} QREs for the protocol program.

Very quickly we realized that the specification of an \texttt{exclusive-read-write} property could make use of a trivial modification of the \texttt{exclusive-write} QRE and that the PAs for these specifications would have the same number of states. The QRE is:

\[
\{ \texttt{wstart, wstop, rstart, rstop} \} \\
\text{all} \\
[-\texttt{wstart}]^*; (\texttt{wstart};[-\texttt{rstart}, \texttt{rstop}, \texttt{wstart}, \texttt{wstop}]^*; \texttt{wstop};[-\texttt{wstart}]^*)^*
\]

This QRE excludes the possibility of 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 \texttt{no-read-upon-write} condition:

\[
\{ \texttt{wstart, rstart, rstop} \} \\
\text{none} \\
.*; rstart; [-\texttt{rstop}]^*; \texttt{wstart};.*
\]

The \texttt{.*} expression is used to match any string of symbols.
We apply the steps of our methodology to explore the effects of different refinements and feasibility constraints on the precision and cost of analysis of these properties. We perform five different variations of FLAVERS analysis on the 2 reader 2 writer program for the exclusive-read-write property: basic, ALPHA, ALPHA+CI, ALPHA+TA, and ALPHA+VA. We also include the ALPHA+VA for the exclusive-write property. Table 5.2.2 gives data on the sizes of the artifacts produced during these various analyses. The CPAs are produced as follows:

\[ CPA_1 = PA \ast T_A^{CTL} \]
\[ CPA_2 = PA \ast V_A^{WP} \]

In Table 5.2.2 we present information on the execution performance of the six analyses. Each column represents a phase of analysis; if a column is blank then that phase is not required for the given analysis. It is interesting to note that the TFG was unaffected by performing CI refinement. The readers/writers program does have an interval-like structure to the pattern of communications it engages in. Unlike the protocol program, however, the enforcement of this structure is encoded in both the control and data states of the program. CI refinement is only able to detect intervals that are enforced purely by the control structure of the program. We note that for
Table 5.4 Analysis Cost for exclusive-read-write Analyses

<table>
<thead>
<tr>
<th>Phase</th>
<th>Basic</th>
<th>Alpha</th>
<th>Alpha+CI</th>
<th>Alpha+TA</th>
<th>Alpha+VA Read-Write</th>
<th>Alpha+VA Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>24.8</td>
<td>23.0</td>
<td>22.9</td>
<td>23.0</td>
<td>24.6</td>
<td>24.4</td>
</tr>
<tr>
<td>QRE</td>
<td>1.1</td>
<td>0.7</td>
<td>0.7</td>
<td>1.8</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>ALPHA</td>
<td>5.8</td>
<td>5.7</td>
<td>6.8</td>
<td>6.0</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td></td>
<td></td>
<td></td>
<td>5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control TA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>WP VA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>CPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.3</td>
<td>4.4</td>
</tr>
<tr>
<td>UNIFY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>MIP</td>
<td>6.4</td>
<td>4.3</td>
<td>4.6</td>
<td>4.7</td>
<td>4.8</td>
<td>4.4</td>
</tr>
<tr>
<td>PROPS</td>
<td>8.7</td>
<td>5.4</td>
<td>5.2</td>
<td>8.6</td>
<td>6.3</td>
<td>5.2</td>
</tr>
<tr>
<td>Total</td>
<td>41.0</td>
<td>39.2</td>
<td>44.4</td>
<td>71.0</td>
<td>47.5</td>
<td>44.6</td>
</tr>
<tr>
<td>Result</td>
<td>inc</td>
<td>inc</td>
<td>inc</td>
<td>inc</td>
<td>con</td>
<td>con</td>
</tr>
</tbody>
</table>

a trivial increase in analysis cost we can extend the analysis of the exclusive-write property to analyze the exclusive-read-write property.

The goal of running this series of analyses is to determine the lowest cost method of obtaining conclusive analysis results. We found a single analysis, for the exclusive-read-write property, that obtains conclusive results: Alpha+VA. In analyses of successively larger versions of the protocol program we will apply Alpha+VA as our set of refinements and feasibility constraints. We note that all of the results that follow are for analyses that produce conclusive results.

We now consider how the cost of analysis scales as we increase the number of readers and writer tasks in the program. We begin with two readers and two writers, then increase the size of the program by doubling the number of readers and writers at each step. Figure 5.7 plots total analysis time versus the number of nodes in the TFG. This is a rate-of-growth plot; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. We can see that the rate of growth of the Alpha+VA analysis is sub-cubic for this problem. Total analysis time for the largest example, 32
readers and 32 writers, was approximately 96 minutes.

In analyzing the no-read-upon-write property we start with the assumption that modeling the WriterPresent variable will be necessary; it proved invaluable in gaining precise analysis results for the previous property. We found, unexpectedly, that incorporating a VA for that variable was insufficient for gaining conclusive analysis results for the no-read-upon-write property. In Section 5.4, we discuss our experiences in using the FLAVERS/Ada tools to diagnose the source of the imprecise results and in finding an alternative precise FLAVERS analysis. Our analysis series for this property is short. We perform two different variations of FLAVERS analysis on the 2 reader 2 writer program for the no-read-upon-write property: Alpha+VA and Alpha+2VAs. Table 5.2.2 presents data on the sizes of the artifacts produced during these analyses. Alpha1 is the refinement of the raw TFG based on the alphabet of the specified property and the set of events that reference the variable WriterPresent. Alpha2 is the refinement of the raw TFG based on the alphabet of the specified property and the set of events that reference variables WriterPresent.
Table 5.5 Artifact Sizes for no-read-upon-write Analyses

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Nodes (States)</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFG</td>
<td>65</td>
<td>671</td>
</tr>
<tr>
<td>Alpha1 TFG</td>
<td>40</td>
<td>282</td>
</tr>
<tr>
<td>Alpha2 TFG</td>
<td>42</td>
<td>354</td>
</tr>
<tr>
<td>PA</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>WriterPresent VA</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ActiveReaders VA</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>CPA1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>CPA2</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

and ActiveReaders. 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. In the data presented below we bound the number of readers at 2 for all sizes of the program. This has the effect of fixing the size of the CPA for different program sizes. The CPAs are produced as follows:

\[ CPA1 = PA \times VA_{WP} \]
\[ CPA2 = PA \times VA_{WP} \times VA_{AR} \]

In Table 5.2.2 we present information on the execution performance of the six analyses. Each column represents a phase of analysis. If a column is blank then that phase is not required for the given analysis. The goal of running this series of analyses is to determine the lowest cost method of obtaining conclusive analysis results. We found a single analyses, for the no-read-upon-write property, that obtains conclusive results: Alpha+2VA. In analyses of successively larger versions of the readers/writers program we will apply Alpha+2VA as our set of refinements and feasibility constraints. All of the results that follow are for analyses that produce conclusive analysis results.
Table 5.6 Analysis Cost for no-read-upon-write Analyses

<table>
<thead>
<tr>
<th>Phase</th>
<th>Alpha+VA</th>
<th>Alpha+2VA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>16.7</td>
<td>19.6</td>
</tr>
<tr>
<td>QRE</td>
<td>1.3</td>
<td>1.9</td>
</tr>
<tr>
<td>ALPHA</td>
<td>4.0</td>
<td>4.3</td>
</tr>
<tr>
<td>WriterPresent VA</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>ActiveReaders VA</td>
<td></td>
<td>0.4</td>
</tr>
<tr>
<td>CPA</td>
<td>2.4</td>
<td>8.0</td>
</tr>
<tr>
<td>MIP</td>
<td>2.3</td>
<td>3.0</td>
</tr>
<tr>
<td>PROPS</td>
<td>4.0</td>
<td>7.4</td>
</tr>
<tr>
<td>Total</td>
<td>31.2</td>
<td>45.8</td>
</tr>
<tr>
<td>Result</td>
<td>inc</td>
<td>con</td>
</tr>
</tbody>
</table>

We now consider how the cost of analysis scales as we increase the number of readers and writer tasks in the protocol program. We begin with two readers and two writers, then increase the size of the program by doubling the number of readers and writers at each step. Figure 5.8 plots total analysis time versus the number of nodes in the TFG. This is a rate-of-growth plot; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. We can see that the rate of growth of the Alpha+2VA analysis is sub-cubic for this problem. Total analysis time for the largest example, 32 readers and 32 writers, was approximately 51 minutes.

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:

\[ \{w\text{start}, r\text{start}\} \]

all

\[-r\text{start}^*;w\text{start}^*;.*\]
Figure 5.8 Total Analysis Time for no-read-upon-write

We perform five different variations of FLAVERS analysis on the 2 reader 2 writer program for the write-first property: basic, ALPHA, ALPHA+CI, ALPHA+TA, and ALPHA+VA. Table 5.2.2 gives data on the sizes of the artifacts produced during these various analyses. The CPAs are produced as follows:

\[ CPA1 = PA \times TA_{CTL} \]

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Nodes (States)</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFG</td>
<td>65</td>
<td>671</td>
</tr>
<tr>
<td>Alpha TFG</td>
<td>36</td>
<td>194</td>
</tr>
<tr>
<td>CI Alpha TFG</td>
<td>36</td>
<td>194</td>
</tr>
<tr>
<td>PA</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Control TA</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>WP VA</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CPA1</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>CPA2</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7 Artifact Sizes for write-first Analyses
Table 5.8 Analysis Cost for **write-first** Analyses

<table>
<thead>
<tr>
<th>Phase</th>
<th>Basic</th>
<th>Alpha</th>
<th>Alpha+CI</th>
<th>Alpha+TA</th>
<th>Alpha+VA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>26.1</td>
<td>22.9</td>
<td>23.2</td>
<td>23.0</td>
<td>23.1</td>
</tr>
<tr>
<td>QRE</td>
<td>0.9</td>
<td>0.6</td>
<td>0.6</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>ALPHA</td>
<td>4.7</td>
<td>5.1</td>
<td>6.0</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td></td>
<td></td>
<td></td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>Control TA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11.0</td>
</tr>
<tr>
<td>WP VA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>CPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIFY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td>MIP</td>
<td>6.8</td>
<td>4.3</td>
<td>4.4</td>
<td>4.8</td>
<td>4.2</td>
</tr>
<tr>
<td>PROPS</td>
<td>8.3</td>
<td>4.8</td>
<td>4.7</td>
<td>8.1</td>
<td>5.1</td>
</tr>
<tr>
<td>Total</td>
<td>42.1</td>
<td>37.1</td>
<td>42.9</td>
<td>67.2</td>
<td>43.2</td>
</tr>
<tr>
<td>Result</td>
<td>inc</td>
<td>inc</td>
<td>inc</td>
<td>con</td>
<td>con</td>
</tr>
</tbody>
</table>

\[ CPA2 = PA \times VA_{WP} \]

In Table 5.2.2 we present information on the execution performance of the five analyses. Each column represents a phase of analysis; if a column is blank then that phase is not required for the given analysis. 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. All such prefixes are enforced by 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, Alpha+TA and Alpha+VA. The Alpha+VA analysis was less costly. In analyses of successively larger versions of the program, we will apply Alpha+VA as our set of refinements and feasibility constraints. All of the results that follow are for analyses that produce conclusive analysis results.

We now consider how the cost of analysis scales as we increase the number of readers and writer tasks in the program. We begin with two readers and two writers,
then increase the size of the program by doubling the number of readers and writers at each step. In Figure 5.9 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 readers and 32 writers, was approximately 32 minutes.

5.2.3 The Dining Philosophers Problem

A number of researchers have analyzed versions of the dining philosophers problem [4, 16, 23, 53, 86]. Typically, analysis is performed to insure deadlock freedom. The current FL AVERS/Ada toolset does not support reasoning about deadlock. Instead, we reason about event sequencing properties.

The program has at least 4 tasks; it is scalable. The dining philosophers problem consists of equal numbers of philosopher and fork tasks. 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 on his 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. Other example programs, e.g., readers/writers, gas station, and protocol programs have centralized servers. The DARTES system has a centralized master task.

We wanted to check the property that "adjacent philosophers cannot eat concurrently". 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:

\[
\{ \text{phils}_f, \text{d1}, \text{phils}_f, \text{n1} \} \\
\text{all} \\
[-\text{phils}_f, \text{n1}]^*; \\
( \text{phils}_f, \text{n1}; [-\text{phils}_f, \text{n1}]^*; \\
\text{phils}_f, \text{d1}; [-\text{phils}_f, \text{u1}]^* \\
)*
\]

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 have 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 Alpha refinement; not surprisingly, since the neighbors-thinks property is very local, the resulting FLAVERS analysis produced conclusive analysis results. Table 5.2.3 gives data on the sizes of the artifacts produced during these various analyses. In Table 5.2.3 we present information on the execution performance of the three analyses. Each column represents a phase of analysis; if a column is blank then that phase is not required for the given analysis.

Now we consider how the cost of the Alpha analysis scales as we increase the number of philosophers and fork tasks in the program. We begin with three philosophers and three forks, then increase the size of the program by doubling the number of philosophers and forks at each step. Figure 5.10 plots total analysis time versus
the number of nodes in the TFG for the Alpha analysis. This is a rate-of-growth plot; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. Total Alpha analysis time for the largest example, 48 philosophers and 48 forks, was approximately 7 minutes.

For the multiple neighbors-think analysis we can reuse the unrefined TFG and the PA artifacts. This saves the cost of running the LP and QRE analysis phases. Unfortunately, the alphabets of each of the individual fork QREs is different so we must perform all of the other phases for each fork. Figure 5.11 plots total analysis time versus the number of nodes in the TFG for the Alpha analysis. This is a rate-of-growth plot; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. Total time for all of the multiple Alpha analyses for the largest example, 48 philosophers and 48 forks, was approximately 3 hours and 43 minutes. We note, however, that unlike composite analyses, multiple analyses are amenable to parallel speedup using the current toolset. If we are willing to pay multiple overhead costs for
the LP and QRE phases, not to mention the additional space, we can reduce total analysis time to 58 minutes by using four parallel streams of analysis.

5.2.4 The Gas Station Problem

The gas station problem is a simulation of an automated self-serve gas station [35].

A number of researchers have analyzed versions of the gas station problem; typically, analysis is performed to insure deadlock freedom and to ensure that correct change is given. The program has at least 4 tasks; it is scalable.

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 is 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 which accepts payments and gives change to customers and a number of pump tasks which independently start and stop the pumping of gas. The operator interacts with pumps by enabling them to pump gas after payment has been received and get 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 [16] in that we include explicit program termination code; the operator will only shutdown when there are no active customers and after it has successfully turned the pump off.

Like the readers/writers problem the global state of a gas station program is not completely captured by the control states of the program tasks. The operator task maintains some local state variables that control the pattern of communication events that it engages in and indirectly controls the behavior of the pump. These variables include a bounded counter and a bounded length queue.

One of the properties we attempted to analyze was about the first-come first-serve nature of service provided to clients. We specified the property that "if customer 1 pays first then it will pump before any other customer" as the QRE:

```
{ gas_operator_prepay_1, gas_operator_prepay_2, gas_operator_prepay_3,
  start1, start2, start3}
all
[- gas_operator_prepay_2, gas_operator_prepay_3]*;gas_operator_prepay_1;
[- start2, start3]*;start1;
.*
```

Unfortunately, even modeling all tasks in the system with TAs would not provide conclusive analysis results. FLAVERS, and any other analysis technique, needs to reason about the behavior of the queue in order to provide precise results for this property. Conceptually, FLAVERS can model bounded length queues; the current implementation of FLAVERS/Ada does not yet support VAs for bounded queues.


Table 5.11 Artifact Sizes for one-per-pump Analyses

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Nodes(States)</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFG</td>
<td>101</td>
<td>1500</td>
</tr>
<tr>
<td>Alpha TFG</td>
<td>48</td>
<td>182</td>
</tr>
<tr>
<td>CI Alpha TFG</td>
<td>69</td>
<td>269</td>
</tr>
<tr>
<td>PA</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

We check a property related to the exclusivity of 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:

\[
\begin{align*}
\{ & \text{gas\_pump\_start\_pumping, gas\_pump\_stop\_pumping} \} \\
\text{all} \\
[ & \text{gas\_pump\_start\_pumping}]^*; \\
( & \text{gas\_pump\_start\_pumping}; \\
[ & \text{gas\_pump\_start\_pumping, gas\_pump\_stop\_pumping}]^*; \\
\text{gas\_pump\_stop\_pumping}; \\
[ & \text{gas\_pump\_start\_pumping}]^* \\
)^* 
\end{align*}
\]

We apply the steps of our methodology to explore the effects of different refinements and feasibility constraints on the precision and cost of analysis of these properties. We perform three different variations of FLAVERS analysis on the 3 customer 1 pump program for the one-per-pump property: basic, ALPHA, ALPHA+CI. We stopped here because we found multiple analyses that provided conclusive analysis results. Table 5.2.4 gives data on the sizes of the artifacts produced during these various analyses. In Table 5.2.4 we present information on the execution performance of the three analyses. Each column represents a phase of analysis; if an entry is
Table 5.12 Analysis Cost for one-per-pump Analyses

<table>
<thead>
<tr>
<th>Phase</th>
<th>Basic</th>
<th>Alpha</th>
<th>Alpha+CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>28.5</td>
<td>24.2</td>
<td>28.9</td>
</tr>
<tr>
<td>QRE</td>
<td>2.3</td>
<td>0.8</td>
<td>0.7</td>
</tr>
<tr>
<td>ALPHA</td>
<td>3.6</td>
<td>7.6</td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td></td>
<td></td>
<td>8.0</td>
</tr>
<tr>
<td>MIP</td>
<td>8.7</td>
<td>2.5</td>
<td>4.0</td>
</tr>
<tr>
<td>PROPS</td>
<td>14.4</td>
<td>2.6</td>
<td>4.2</td>
</tr>
<tr>
<td>Total</td>
<td>53.9</td>
<td>33.7</td>
<td>48.5</td>
</tr>
<tr>
<td>Result</td>
<td>inc</td>
<td>con</td>
<td>con</td>
</tr>
</tbody>
</table>

blank then that phase is not required for the given analysis. The goal of running this series of analyses is to determine the lowest cost method of obtaining conclusive analysis results. We found two analyses that obtain conclusive results: Alpha and Alpha+CI. Typically, we would just choose the cheaper of the two and apply the resulting FLAVERS analysis to increasingly larger programs. For the one-per-pump property we were interested in seeing whether the reduction in MIP edges from the CI refinement, and the subsequent state propagation time, compensated for the cost of performing the refinement.

First, we consider how the cost of both the Alpha and Alpha+CI analyses scale as we increase the number of customer tasks in the gas station program. We begin with 3 customers, then increase the size of the program by doubling the number customers at each step. Figures 5.12 and 5.13 plot total analysis time versus the number of nodes in the TFG for Alpha and Alpha+CI analyses, respectively. These are rate-of-growth plots; as such it is on a log-log scale and includes a reference line whose slope is $N^3$. Total Alpha analysis time for the largest example, 48 customers, was approximately 32 minutes. The Alpha+CI analysis could not process the 48 customer program due to tool limitations. We believe that this is because the data structures used in performing the analysis are not optimized for space. This example is causing
the infra-structure on which the tools are based to exceed its space capacity. We are looking to develop light-weight data structures to address this problem.

5.2.5 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, e.g., dining philosophers.

Masticola [52] and Corbett [16] have analyzed versions of the program to check for deadlock freedom. As with other programs, the current FLAVERS/Ada tools do not support analysis 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; it is non-scalable. The simulation is structured such that 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

\[-T.go; *T.go; -T.initialize; *T.initialize; -T.stop; *T.stop \]

The ordering of these events is constrained by the control flow edges in the TFG that are associated with task T; as illustrated in Section 4.6, however, the presence of TFG MIP edges may introduce imprecision into the analysis. 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 causes the current FLAVERS/Ada toolset to exceed its storage capacity. However, incorporating alphabet refinement allows the analysis to easily handle this problem. Table 5.13 presents the data for a single FLAVERS/Ada analysis that incorporates alphabet refinement. For this problem, conclusive analysis results were obtained for this analysis problem in less than 6 minutes.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Edges</th>
<th>PA States</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>QRE</td>
<td>ALPHA</td>
<td>MIP</td>
<td>PROPS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Nodes</td>
</tr>
<tr>
<td>121.2</td>
<td>1.2</td>
<td>24.1</td>
<td>41.2</td>
<td>156.3</td>
<td>2466</td>
</tr>
</tbody>
</table>

Table 5.13 Raw Data for no-premature-go
5.3 Some Observations

We believe that the empirical results of the previous sections clearly demonstrate the feasibility of FLAVERS analyses. In particular, there exist natural properties of non-trivial concurrent 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 property 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 property 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 property 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 property is more costly to analyze than the multiple header-packet property for small instances of the protocol program. For protocols with more than 12 clients, however, the composite property becomes cheaper and the cost of analysis for it grows at a lower rate than for the multiple property.

Intuitively, one of the most important factors that influences the cost of FLAVERS analysis is the degree to which a property is local or global. A local property refers to program events whose only instances are in a small cluster of tasks. A global property 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 property will, in general, require fewer TAs than a more global property. Intuitively, the question is:

How much of the program is relevant in determining the execution behavior of the program with respect to the events referred to in the property?

The properties analyzed in the previous sections vary from being local to a pair of tasks, e.g., no-go-init-stop, to including events from all program tasks, e.g., composite header-packet. FLAVERS can be applied cost-effectively to analyze properties that range from local to global. As we develop more refinements and feasibility constraints that target global behaviors, such as communication interval groups, the degree to which the analysis of global properties are a challenge will diminish.

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 TFG and PA, or CPA, size. 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 static analysis techniques reason about the execution behavior of non-finite state systems by approximating that behavior with a finite model. Data flow analyses have typically used weaker approximations to gain performance at the expense of precision. When applying data flow analysis, and FLAVERS in particular, to the verification problem, one of the key questions is whether the analysis can produce highly precise analysis results. The results of the previous sections illustrate, that by incorporating additional information into FLAVERS analyses, all spurious results can be eliminated; thus, the precision of analysis results is sufficient to verify a given property with respect to a given program. The cost of this increased precision varies with the program and property under analysis.

We would hope to populate FLAVERS with a collection of refinements and feasibility constraints that are widely applicable and effective in reducing analysis cost and increasing the precision of analysis results. Our experiences with the problems in this section indicate that both alphabet and communication interval refinement are broadly applicable. Alphabet refinement is universally applicable and appears to be crucial for cost-effective analysis. The cost of state propagation analysis is highly dependent on the size and density of the TFG. Alphabet refinement can reduce the size of the TFG dramatically, and, when it cannot eliminate a node, it relabels it, leading to the elimination of large number of MIP edges. Communication intervals and interval groups were detected in the protocol, gas station, dining philosophers, and DARTES programs; in all cases they led to the elimination of large numbers of MIP edges.

FLAVERS analyses can obtain the same level of precision as state space enumeration techniques. Incorporating TAs for all program tasks and VAs for variable that are feasible to model with the PA results in a CPA that is the state space of the program. State propagation analysis using this CPA will enforce the ordering of global program states; this will eliminate all sequences of program events that do
not correspond to paths in the state space from consideration. The Alpha+AllTA
analysis of the header-packet property illustrates this point. We have not compared
the performance of FLAVERS/Ada with TAs for each program task to state-space
e数mation methods but we expect the costs to be similar. Fortunately, it is often
the case that we don't need that much information to gain precise analysis results
with FLAVERS.

A small set of refinements and feasibility constraints are implemented in the cur-
rent version of FLAVERS/Ada. The space of choices for combining these refinements
and FCs is large but not beyond practical exploration. Our analysis series are in-
stances of such explorations. As the number of refinements and FCs increases the
choice of an appropriate combination becomes more difficult; there will be signifi-
cantly more options. The practical effectiveness of FLAVERS analyses will depend
on the development of a collection of heuristics that can quickly and effectively guide a
user to a combination of refinements and FCs that will yield precise analysis results.

5.4 An Analysis Strategy

FLAVERS has been designed to take advantage of the fact the information re-
quired for precise analysis varies greatly with the property and program under anal-
ysis. The empirical results of this section bear this point out.

In this section we discuss two important issues that users of FLAVERS will en-
counter: diagnosing the source of an unexpected analysis results and selection of one
of the many possible FLAVERS analyses to apply.

Diagnosis

We can view the process of analysis from two different perspectives: finding faults
or verifying behavior. FLAVERS focuses mostly on verification questions, but when
an analysis fails to provide information that is sufficient for verification, we can use
that information to locate the faults that led to the analysis results. Intuitively, if
intention is to verify a property that we believe holds on a program, then analysis
results that fail verify that property are, in some sense, unexpected.

The raw results produced by a FLAVERS analysis are a set of CPA states for each
node in the TFG. The value at the final flowgraph node, which is the intersection of
the values at each task's exit node, is compared to the set of accepting CPA states
to determine satisfaction of the specified property. Intuitively, a CPA state at a node
represents the fact that some event sequence reaching the node has caused the CPA
to transition to the given state. Thus, if we can identify states at the final node that
are causing the unexpected analysis result we can trace back through the predecessor
nodes of the TFG to find the points at which that state was reached. Once these
points are identified we can map the TFG node back to the source code and based
on the information encoded in the state we can check feasibility.

This sounds nice, but, there are is a significant complicating factor: the state
propagation algorithm collapses P-equivalent paths into a single CPA state value.
Distinguishing the actual TFG path(s) that lead to the CPA state in question can be
difficult. We have successfully applied two techniques to help in this regard.

1. We can incorporate task automata into the analysis. This results in a CPA
for which each state represents more detailed information about a sequence of
program events that represents a program execution. This effectively reduces
the amount of P-equivalent collapsing and can distinguish program executions
that were previously collapsed.

2. We can look at the computation of state values in nodes as the state propagation
proceeds. State propagation considers node in program execution order. In this
way we can see the point at which an offending CPA state is first produced
during analysis. This can give a significantly clearer picture of how the state
was reached because the state values from the, yet to be considered, suffixes of program executions will not be present.

To illustrate, these ideas we consider our experience of checking the no-read-upon-write property of the readers/writers problem. We are attempting to determine that no prefixes of program executions have a start_read before a start_write without an intervening stop_read. We attempt a FLAVERS analysis with Alpha+VA where the modeled variable is the local WriterPresent state variable of the control task. The CPA has 11 states with accepting states 9-11. We expect that there will be no accepting states of the CPA at the final TFG node, unfortunately, this is not the case; CPA state 11 was propagated to the final node.

The question is how did state 11 get to the final node? We trace back through the predecessors of the final node and find that most nodes in the flow graph have values that contain state 11. We apply the second technique described above. Rather than look at the values computed by state propagation analysis at nodes on-line, we drop crumbs of information to a file for off-line processing. The crumbs contain the node id and the new state value computed at the node. The order of the crumbs in the file is the order in which the algorithm produces them; it also corresponds to the execution order of program events1. We use standard editor commands to locate the crumb in which state 11 first occurs. This corresponds to the shortest program event sequence that leads to state 11. The CPA state provides two pieces of information: the PA state and the value of the WriterPresent variable. State 11 corresponds to having seen an accepting prefix, i.e., a start write after a start read without an intervening stop read, and that the value of WriterPresent is false. We then trace back from this crumb to the beginning of the file looking for crumbs that enabled the transition to state 11, i.e., \( z \) where \( \delta(z, a) = 11 \) for some \( a \in \Sigma \). By recording the sequence of crumbs we are recreating a portion of the program execution that leads to state 11. In

\[1\] It is possible to perform state propagation over the reverse of the flow graph with a reversed PA. In this case the crumb order would be the reverse of the execution order.
doing so, we find a branch condition, $\text{ActiveReaders} = 0$ and not $\text{WriterPresent}$, that controls the execution of $\text{start\_write}$ which is modeled pessimistically, i.e., we ignore some of the variable values. It is clear from the semantics of the states and the source lines that the TFG nodes map to that this TFG sub-path is not executable.

At this point we have identified that with respect to at least one of the paths that caused our analysis to produce inconclusive results the result was spurious. Furthermore, this process identified a variable, $\text{ActiveReaders}$, that, if modeled, may increase the precision of analysis. It turns out that adding a feasibility constraint for $\text{ActiveReaders}$ leads to a FLAVERS analysis that produces conclusive analysis results.

We have used the same technique to locate faults that were purposely seeded in programs as a means of validating the analysis tools. In Chapter 7 we discuss how this fault diagnosis can be partially automated.

Analysis Selection

The current FLAVERS/Ada tools includes the basic analysis components and four optional components: alphabet refinement, communication interval refinement, task automata and variable automata. For the feasibility constraints we can incorporate multiple automata into a single CPA. In addition, we can solve separate analyses then use those results to reason about conjunctions and disjunctions of verified properties.

For each of these optional components we describe some heuristics that are useful in determining whether to incorporate the component into a particular FLAVERS analysis.

Alphabet Refinement We have found that, except for very simple programs and properties, it is always cost-effective to apply alphabet refinement. Even in cases when a basic FLAVERS analysis will produce precise results the additional cost of alphabet refinement is small percentage of overall analysis time.
CI Refinement Determining that communication interval refinement can improve the precision of analysis involves identifying pairs of tasks that communicate, computing CI refinement for them, and determining whether any intervals were detected. Determining pairs of communicating tasks in Ada is simple; we need only look at entry calls since they name the task that accepts the call either explicitly or implicitly, via language scope rules. Computing CI refinement is accomplished by running the \texttt{ci_refine} tool. This tool can be configured to output the communication intervals it detects. If no intervals are found then CI refinement cannot be profitable; if some intervals are found, then there is the potential for improved precision and reduced analysis cost.

Variable Automata Intuitively, we model a variable to eliminate sequences of program events represented in the TFG that are infeasible. In particular, sequences that represent modifications and test of the value of the variable that violate programming language semantics are eliminated.

The current FLAVERS/Ada tools are capable of constructing variable automata to model any boolean or bounded counter variable used in the program\(^2\). We have found variable automata to be effective for modeling \textit{state} variables that are used to control the execution of program events in the QRE under analysis. We use the concept of control dependence to identify candidate variables for modeling. If a node \(n\) is control dependent on a branch decision \(m\) where the variable \(v\) is tested at \(m\), and \(v\) is a boolean or bounded counter variable then we model \(v\) with a VA.

Task Automata Intuitively, we model the control flow of a task when that task can influence the execution of some program event that is referred to in the QRE under analysis.

\(^2\)Extensions to the tools are planned to accommodate other variables with small finite domains, such as enumeration types.
The obvious case is when a task contains instances of events in \( \Sigma_{\text{property}} \). It is quite common, however, for a task that does not directly contain such events to influence the execution behavior of tasks that do contain such events. This influence can be due to either inter-task data flow or inter-task control flow; in Ada tasking programs the most common cause is the ordering of events local to a task that is induced by synchronization at inter-task communication points. We define a notion of relevance distance for a QRE and a program. All tasks that contain symbols in \( \Sigma_{\text{property}} \) are at distance 0. If we consider tasks at distance \( i \) then all other tasks whose distance is not less than \( i \), that communicate with a task at distance \( i \) have a relevance distance of \( i + 1 \). We introduce TAs into an analysis by increasing relevance distance. The idea is to first try the tasks that most directly influence the execution behavior that is related to the specified property.

**Decomposing QREs** It is often the case that a global property of a concurrent program can be decomposed into simpler more restricted sub-properties. A typical example is the header-packet property for the protocol problem; we want to specify that the pattern of events \( \forall T_i \in \text{Tasks} : \forall T_j \neq T_i : [\neg h_i]; (h_i; [\neg h_j]; p_i; [\neg h_i])^* \). We refer to this as quantification in a specification. Unfortunately, the QRE formalism, and regular expressions in general, do not provide quantification or predicate like primitives. The patterns are constructed out of atomic proposition-like symbols. We can, however, expand such a property into a conjunction of QREs if we know the set of tasks in the program. Intuitively, we treat the universal quantifiers as a kind of iteration, by unroll these loops and substituting the appropriate values for \( i \) and \( j \) we get the individual QREs. The multiple header-packet and neighbors-think properties are examples of this analysis strategy. These analyses can benefit from having their component analyses executed in parallel; the current FLAVERS/Ada tools support this.
It is also possible that a property naturally decomposes into sub-properties. A good example of this is the combination of \texttt{exclusive-read-write} and \texttt{no-read-upon-write} to specify the intended behavior of the readers/writers problem with respect to exclusivity of write operations. We refer to this as a decomposable specification.

This collection of heuristics is a reflection of our experience to date with applying FLAVERS. We expect that as our experience grows the set of available heuristics will also grow.

\textbf{Putting it all together}

Given the heuristics and strategies described above how do we go about applying FLAVERS to a new analysis problem. There are three steps to the overall analysis strategy:

\begin{verbatim}
if desired property is decomposable or quantified then
  break it up into sub-properties to be analyzed in isolation
end if
apply alphabet refined FLAVERS analysis
while analysis results are inconclusive loop
  apply diagnosis strategies
  if source of unexpected result is a fault then
    analysis complete
  else if source of unexpected result is a spurious result then
    if source of spurious result is related to branch decision
      apply VA heuristics
    if no additional VAs can be applied then
      analysis complete
    end if
  end if
end while
\end{verbatim}
else
    if no additional TAs can be applied then
        analysis complete
    else
        incorporate a TA from next larger relevance distance
    end if
end if
end if
end if
end loop

This approach to producing and applying FLAVERS analyses to a program and property was developed during experimentation with the protocol and readers/writers programs. It was then applied to the analysis of properties of the gas station, dining philosophers, and DARTES programs.
A FLEXIBLE ARCHITECTURE FOR BUILDING DATAFLOW ANALYZERS

Chapter 3 presented a formalism for describing data flow analysis problems. Chapter 4 applied that formalism to develop FLAVERS. In this chapter, we present a domain-specific architecture that leverages off of the inherent generality of data flow frameworks to provide low-cost construction of data flow analyzers. Typically, data flow analyzers are hand-crafted to solve a particular problem. Building analyzers requires a significant effort. Developers must consider, and choose from, a wide variety of alternatives for encoding the data flow analysis problem and then implement the analyzer. While evaluation of some analysis design alternatives can be done analytically, it is often the case that the cost-effectiveness of a particular approach must be judged empirically. In these cases, the significant software development cost of building data flow analyzers is a barrier to exploring the space of analysis design options.

In this chapter, we describe an architecture that facilitates rapid prototyping of data flow analyzers. The architecture defines a set of interoperating standard interfaces for components of data flow analyzers. With this architecture, a developer chooses from a collection of pre-existing components or, using high-level component generators, constructs new components and combines them to produce a data flow analyzer. Component generators capture common functionality and facilitate creation of components specialized for the problem at hand. Finally, the interfaces defined by the architecture allow existing and generated components to be reused across data
flow analyzers. Thus, the cost of building or generating a component can be amortized over a number of analyzers.

We envision that this architecture will be especially useful in the early stages of development of data flow analyzers. Prototype analyzers can be developed quickly for a variety of formulations of a given data flow analysis problem. Although we do not expect prototype analyzers to be as efficient as hand built, finely-tuned analyzers, they will produce results of equal precision. After evaluating and selecting the desired problem formulation a more efficient analyzer can be produced. At this point a developer could decide to hand code a highly optimized version of the analyzer or, alternatively, employ component generators and special purpose components optimized to produce a more efficient analyzer using the architecture. For example, this architecture supports the construction of sparse representations that can significantly reduce the cost of analysis. Another strategy for improving both the efficiency and precision of analysis is combining multiple data flow analysis problems into a single problem. With combined problems, rather than executing multiple analyses independently we run a single analysis. Precision is improved when the constituent problems can use each other's intermediate results; analysis time is reduced by incurring analysis overhead costs a single time rather than once for each run of the constituent analyses. An extension of the architecture allows analyzers to be built for a useful class of combined data flow problems.

We believe this architecture supports a wide variety of analysis problems. We have implemented a library of analyzer components as a collection of interoperating Ada generic packages that provide the interfaces defined by the architecture. This architecture has been used to construct a variety of data flow analyzers that are used in a toolset for analyzing explicitly stated correctness properties of distributed systems [24]. These include analyzers for traditional data flow problems, e.g., dominators, live variables, constant propagation, and for non-traditional problems, e.g.,
for complex reachability problems. These analyzers have been formulated over a variety of program representations, including sequential and concurrent control flow graphs, reachability graphs for concurrent systems, and Petri nets. A number of these analyzers are important components of larger software systems and, as such, they have evolved over time. We have found that modifications to these analyzers have required very little programming effort. Our experience demonstrates that the architecture supports a wide variety of analysis problems and reduces the cost of developing and maintaining data flow analyzers.

The next section describes related work. Sections 6.2 and 6.3 describe the architecture and library of components, respectively. Section 6.4 describes an extended architecture for building analyzers for a class of combined data flow problems. Section 6.5 describes several data flow analyzers that have been built using an implementation of the architecture. We summarize our contributions and plans for future work in the conclusion.

6.1 Background

A typical method for describing data flow analyses is as a system of equations. These equations are derived from the structure of the program being analyzed and based on the information being computed by the analysis. An alternate means of formulating a data flow problem is as a *data flow framework*. A data flow framework is a collection of rigorously defined mathematical objects: a function space and map, a lattice of flow values and a flow graph. Reasoning about these mathematical objects allows one to determine performance characteristics of analyzers for a given data flow problem. A number of theoretical results and algorithms related to specific classes of data flow frameworks have been developed [3, 34, 51].

For decades data flow analyzers have been an integral part of optimizing compilers. Well-engineered compilers often provide standard interfaces to analyzer components;
this can insulate the majority of the compiler from changes to a particular data flow analyzer and may ease the integration of new analyzers in support of new optimizations. Compiler systems do not, however, provide high-level analyzer generator capabilities.

Recent work has exploited the inherent generality of data flow frameworks and attempted to explore some of the issues in supporting a flexible, general approach for constructing data flow analyzers. FIAT [33] is a framework for rapid prototyping of interprocedural analyses and transformations; it provides interfaces for describing a data flow analysis problem as a data flow framework and provides a general iterative solver with which to construct analyzers. Sharlit [78] is a tool for generating compiler optimization phases that incorporate data flow analyzers; it generates a data flow analyzer, based on an iterative solver, from code fragments that specify the components of a data flow framework.

Our work is similar to both FIAT and Sharlit in that it is based on specifying the data flow analysis problem as a data flow framework. Like those systems, our architecture provides interface descriptions of the function space, flow graph, and lattice. Unlike those systems, our approach also treats the solver algorithm as a component of an analyzer and defines an interface for it; this allows different solution algorithms to be used in a data flow analyzer by changing the underlying solver. Our approach also provides a library of pre-existing components that can be used to build analyzers as well as generators for common classes of analyzer components. In addition, we provide support for building analyzers for a class of combined data flow problems.

6.2 An Architecture for Data Flow Analyzers

Our architecture is based on the mathematical objects that constitute a data flow framework. The interface to each object is defined by an architectural template. Ab-
stractions that satisfy these interfaces are referred to as components of a data flow analyzer. Formally, a data flow framework is \((L, G, F, M)\):

\[
L = (V, \sqcap, \sqcup, \not\sqsubseteq, \sqsubseteq, \perp)
\]

\[
G = (E, \text{Start}, \text{Pred} | E \to \mathcal{P}(E), \text{Succ} | E \to \mathcal{P}(E))
\]

\[
F = \{ f | V \to V \}
\]

\[
M | E \to F
\]

\(L\) is a meet-semilattice with a partially ordered set of values, \(V\); these values encode information about program behavior that we are interested in collecting. A meet operator, \(\sqcap\), join operator, \(\sqcup\) and not above operator, \(\not\sqsubseteq\), are included. These operators compute for two lattice values the greatest lower bound, least upper bound and whether the values are ordered in the poset. We don’t need to explicitly define \(\sqsubseteq\) because it is can be defined in terms of \(\sqcap\) and \(\sqcup\). \(G\) is a flow graph, with a set of entities, \(E\), a set of designated start entities, \(\text{Start}\), and the predecessor, \(\text{Pred}\), and successor, \(\text{Succ}\), functions. Entities can be defined as subsets of flow graph nodes or edges; thereby allowing a wide variety of graphs to be viewed as a flow graph. \(F\) is a function space consisting of a set of transfer functions defined over the lattice values, \(V\); these functions encode the effects of the program on the lattice values. \(M\) is a function map that binds flow graph entities, \(E\), to transfer functions. The lattice definition given above can express both semi-lattices, required by traditional sequential data flow frameworks, and complete-lattices, required by the frameworks described in Chapter 3.

The architecture consists of templates that specify the interface to each part of a data flow framework and to the solution algorithm. In practice, function maps

---

1Although we require \(\sqsubseteq\), this restriction could be lifted as long as initial values are available for the problem.

2We include the \(\sqsubseteq\) explicitly since it can be used to optimize the performance of analyzers.
are defined in terms of attributes of flow graph entities; consequently, our interface merges the specification of function space and function map. We refer to a mutually consistent set of lattice, function space, and flow graph components as a data flow problem. A solver is instantiated with a data flow problem to produce an analyzer, as illustrated in Figure 6.1.

In the remainder of this section we provide a description of the interfaces specified by the architectural templates.

Lattice

The lattice values constitute the data that are propagated throughout the flow graph. These values are transformed by transfer functions and combined at merge points in the flow graph. The interface to the lattice is:

```plaintext
type LatticeValue;
function Create return LatticeValue;
procedure Destroy(v : in out LatticeValue);
function Equal(x, y : in LatticeValue) return Boolean;
procedure Assign(l : out LatticeValue; r : in LatticeValue);
function NotAbove(x, y : in LatticeValue) return Boolean;
```
function Meet(x, y : in LatticeValue) return LatticeValue;
function Join(x, y : in LatticeValue) return LatticeValue;

Top : constant LatticeValue;
Bottom : constant LatticeValue;

In this interface, LatticeValue is $V$, Meet is $\sqcap$, Join is $\sqcup$, NotAbove is $\not\sqsubseteq$, Top is $\top$, and Bottom is $\bot$. We include constructor, destructor, assignment and equality operators to allow manipulation of lattice values in intermediate computations.

Flow Graph

The flow graph consists of a collection of entities and predecessor and successor functions that describe the ordering of entities. The interface to a flow graph is:

type FlowGraph;
type Entity;
function MaxEntity(g : in FlowGraph) return Natural;
function GetIndex(e : in Entity) return Natural;
function Starts(g : in FlowGraph) return SetOfEntity;
function Predecessors(e : in Entity) return SetOfEntity;
function Successors(e : in Entity) return SetOfEntity;

In this interface, FlowGraph is $G$, Entity is $E$, Starts is $Start$, Predecessors is $Pred$ and Successors is $Succ$. Entities can be defined as a collection of nodes or edges; this interface allows us to view a wide variety of program representations as flow graphs. We require two additional operators, GetIndex and MaxEntity, that map entities to unique indexes and provide the maximum index value in a graph, respectively. These operators enable construction of analyzers that are more time and space efficient; in our experience these requirements are easily satisfied and have significant payoff.
The direction of the data flow analysis is defined by the predecessor and successor operators. For example, backward flow analyses can be defined by exchanging the Predecessors and Successors operators and by defining Starts to be the exit entities of the flow graph.

Function Space

The function space consists of a set of transfer functions that propagate and potentially transform lattice values at each flow graph entity. While the operators of the function space are defined over lattice values they are not part of the lattice; there may be many function spaces defined over a given lattice. The interface to a function space is:

```plaintext
function Init return LatticeValue;
function Start return LatticeValue;
function FunctionMap(e : in Entity; v : in LatticeValue)
    return LatticeValue;
```

The FunctionMap operator can be thought of as selecting a function from $F$ when given an entity and applying that function to the given lattice value. We include Init and Start to specify initialization values for non-start flow graph entities and start entities respectively; for many problems these are defined using $\top$ or $\bot$. The function space described by this template is equivalent to a set of flow equations of the form:

\[
In(e) = \text{Confluence}_{p \in \text{Pred}(e)}(Out(p))
\]

\[
Out(e) = \text{FunctionMap}(e, In(e))
\]

where the Confluence operator is typically the lattice Meet or Join operator. In this notation the operators have been extended to sets of lattice values, and $In$ and $Out$
are the values flowing into and out of the entity. Note that initially for start entities $In = Start$, and for other entities $Out = Init$.

**Solver**

A data flow problem is formulated as a mutually consistent lattice, function space and flow graph. To produce a data flow analyzer for a problem we need to specify a solution algorithm, which we refer to as the solver component. In this architecture all solvers have a common interface that consists of a component for each part of a data flow problem. These components are given as input to the solver component and the following are produced:

```plaintext
type Results;

function GetInValue(e : in Entity; r : in Results)
    return LatticeValue;

function GetOutValue(e : in Entity; r : in Results)
    return LatticeValue;

function Solve(g : in FlowGraph) return Results;
```

The `Solve` operator computes the solution to the given data flow problem for the input graph and returns the final values for flow graph entities as a `Results` value. `GetInValue` and `GetOutValue` are used to retrieve the lattice value for individual entities. These operations constitute the interface to the data flow analyzer.

**6.3 The Library of Components**

The architectural templates define interfaces that individual components must satisfy in order to be combined to produce a data flow analyzer. Conceptually, other than the template specifications, there are no restrictions on the components that may be used to fill the roles of each template. Practical data flow analyzers usually consist
of monotone function spaces, finite lattices, and flow graphs that are linear in the size of the program. In this section, we describe a library of components designed to support the production of practical analyzers. Figure 6.2 depicts the library organized around each architectural template. Boxes with G denote generators for components that satisfy the associated template. Some components are specifically designed to work together, such as the boolean variable lattice and function space.

To illustrate the construction of a data flow analyzer with the architecture and library of components we present a running example. The problem is the computation of local dominators for communication interval refinement described in Section 4.6.1. This example is implemented using a collection of Ada generic packages that constitute the library of components and generators.

### 6.3.1 Lattice Components

We provide three components that satisfy the lattice interface: bit-vectors, sets and boolean variables. Bit-vectors are a common representation in data flow problems for which the values of interest can be easily embedded in a powerset. Although they can be consumptive of space, they provide efficient Meet, Join and NotAbove operators. In addition, the transfer functions defined in many function spaces can be implemented efficiently as bit-vector manipulations. When the domain of values of interest is too large or the mapping from values to indexes, required for bit-vectors,
Lattice

Figure 6.3 Boolean Variable Lattice

is too expensive, one can use a more general set component. The boolean variable lattice component can be represented as a special case of the lattice of singletons [80]. This lattice, illustrated in Figure 6.3, is designed to work with the boolean variable function space to track the values of boolean variables. We are building components for other common types of state variables, such as bounded counters, and for the lattice of singletons, lattice of intervals, and lattice of arithmetic congruences [30].

The communication dominator problem uses a bit-vector encoding of the set of TFG nodes that represent communication nodes, where the Meet operator is bit-vector intersection and Join is union.

### 6.3.2 Function Space Components

Data flow frameworks for many classic compiler optimization problems, such as available expressions and reaching definitions, have a regular form. Individual transfer functions are constructed from a description of the values generated and killed at each flow graph entity. We provide a generator for this common class of gen-kill function spaces; it takes as input a lattice component, an indication of whether the problem is all paths or any path, and the following two functions:

```c
function Gen(e : in Entity) return LatticeValue;
function Kill(e : in Entity) return LatticeValue;
```

Based on the values of these inputs, the generator produces the Confluence operator, which is Meet for an all paths problem and Join for an any path problem. It also produces the function map operator to define the function space.
function Confluence(x, y : in LatticeValue) return LatticeValue;

function FunctionMap(e : in Entity; v : in LatticeValue)
    return LatticeValue;

This provides a function space that is equivalent to flow equations of the form:

\[
\begin{align*}
    \text{In}(e) &= \text{Confluence}_{p \in \text{Preds}(e)}(\text{Out}(p)) \\
    \text{Out}(e) &= \text{Join}(\text{Gen}(e), (\text{In}(e) - \text{Kill}(e)))
\end{align*}
\]

We also provide a monotone function space defined over the boolean variable lattice. The transfer functions are illustrated in Figure 6.3. The mapping of transfer functions to flow graph entities is made by querying attributes of entities. If the entity assigns constant \textit{true} to the variable or the entity is the \textit{true} branch of a conditional that tests the variable, then \textit{Ftrue} is used. The conditions for \textit{Ffalse} are analogous. If the entity assigns an unknown value to the variable then \textit{Funknown} is used. If the entity assigns the negation of the variable to itself then \textit{Fnot} is used. All other entities are bound to \textit{Fident}, the identity transfer function. We intend to support tracking values of other common types of \textit{state} variables, such as bounded counters.

For the communication dominator problem we define \textit{Gen} to set only the bit corresponding to the node’s index if the node is a communication node. We define \textit{Kill} to return a bit-vector of zeros. The function space is defined as an \textit{all-paths} problem so the \textit{Confluence} operator will be bit-vector intersection, the \textit{Meet} operator for our lattice. Figure 6.4 gives the function space definitions for this example.

6.3.3 Flow Graph Components

A wide variety of control-flow-graph-like representations can be used directly as components in this architecture; others may require a shallow wrapper to make their interfaces conform. Graph types for concurrent programs such as state reachability
graphs, Petri nets, and a number of control-flow-graph-like representations [8, 24, 32, 53] can also be adapted to meet the interface.

The cost of solving a data flow problem is strongly dependent on flow graph size. A number of representations have been developed that effectively reduce flow graph size for some data flow analyses. Choi et. al [10] describe a general algorithm for constructing sparse data flow evaluation graphs (SDFEG) for monotone data flow frameworks. Use of this representation eliminates propagation of data through flow graph regions that add no information to the results. Constructing an SDFEG requires finding paths through the flow graph that correspond to chains of identity, or constant, transfer functions. This dependence on the function space, however, limits reuse of an SDFEG in different data flow problems.

In contrast to SDFEGs, we provide a generator that produces a sparse flow graph component that is independent of a particular data flow problem. We use a relevance predicate to determine whether the entity is relevant and should be included in the representation, or whether it is irrelevant and should be excluded. One can think of the SDFEG construction algorithm as using a restricted relevance predicate, i.e., the existence of a non-identity and non-constant transfer function mapped to a given entity means the entity is relevant. Abstracting away from the function space allows sparse representations to be defined and reused in different data flow problems for which the relevance predicate is appropriate. The generator is given a flow graph component, which defines graph and entity types, and start, predecessor and successor functions, and a relevance predicate:

\[
\text{function IsRelevant}(e : \text{in Entity}) \text{return Boolean;}
\]

It produces a sparse representation component and a constructor that builds a sparse representation from a given flow graph. The sparse representation conforms to the interface for flow graphs defined by the architecture, so that sparse representations can be used wherever a flow graph is required.
In the communication dominator problem, we could formulate the analysis over the set of all TFG nodes and use control flow predecessor and successor operators to produce a data flow analyzer. Alternatively, we could improve analyzer performance by using a sparse representation that elides all TFG nodes that are not communication nodes\(^3\). Figures 6.5 and 6.6 give the definition of the relevancy predicate, IsCOMNode, sparse representation that includes only communication nodes, and the iterative solver. Note that in this implementation, iterators are used for start, predecessor and successor functions.

### 6.3.4 Solver Components

The interface specified by the solver architectural template is very general. It makes few requirements on the structure of the lattice and no requirement on the structure of the function space and flow graph. It is well known that for certain classes of data flow problems, very efficient algorithms exist. Our intent is that those algorithms are to be implemented once, installed in the library of solver components, and incorporated into data flow analyzers as needed. To date, we have provided an iterative worklist solution algorithm for frameworks with monotone function spaces \([34]\).

For the communication dominator problem, the data flow analyzer is constructed using the bit-vector lattice, sparse flow graph, gen-kill function space and iterative solver. Figure 6.5 gives the details of this definition. The solver component accepts additional parameters that are used to optionally produce detailed tracing of analyzer computation. We setup and run the analyzer by calling COMSparseTFG.Create with a TFG.Graph to create a sparse representation of the flow graph, then we pass the sparse representation as input to COMDominators.Solve.

\(^3\)Additional so-called \(\phi\)-nodes may be included in the sparse representation to reduce the number of Confluence operator applications.
6.4 Extending the Architecture for Combined Analyses

The precision of data flow analysis suffers from the fact that all paths through the flow graph are considered executable. Encoding information about path executability can improve the precision of analysis results, but usually increases the size of the flow graph considerably. An alternate approach is to include information in the data flow problem that is used to restrict consideration of certain program paths. This has been done for individual data flow analyses, e.g., [80]. A more general method, and one that we employ, is to use qualified data flow analysis [38].

We refer to the data flow problem of interest as the primary problem. We formulate necessary conditions for path executability and encode those conditions as constraint data flow problems. A qualified problem is a combination of a primary and a set of constraint data flow problems. Conceptually, the qualified problem restricts the propagation of any value that violates one of the necessary conditions encoded in the constraint problems. Care must be taken at flow graph merge points so that information that could be used to restrict value propagation is not lost.

To simplify the discussion, we describe qualified analysis for a single constraint, where both the primary and constraint problems operate over the same flow graph. In this case, a qualified lattice value is a set, whose members are pairs\(^4\) of primary and constraint lattice values, called \(PC\)tuples. We construct the qualified \textit{Confluence} operator to preserve information that may be used to restrict value flow at some point during analysis. To enable this, the developer specifies a function, \textit{MayDiffer}, that defines equivalence classes of constraint lattice values such that, at any point in the flow graph, either all or no members of a class cause flow to be restricted. The \textit{Confluence} operator merges \(PC\)tuples with equivalent constraint values, as determined by \textit{MayDiffer}. The qualified \textit{FunctionMap} is constructed by applying the primary(constraint) \textit{FunctionMap} to the primary(constraint) component of each \(PC\)-

---

\(^4\)For \(k\) constraints we would have \(k\)-tuples rather than pairs.
tuple in the given qualified lattice value. The developer specifies a function, \texttt{Restrict}, that is used to restrict the set of values processed at an entity to only the input values that satisfy the constraint. The induced set of flow equations is as follows:

\[
\begin{align*}
In(e) &= Confluence_{p \in \texttt{Pred}(e)}(\texttt{Out}(p)) \\
\text{Restricted}(e) &= \{PCTuple | PCTuple \in In(e) \land \text{Restrict}(e, PCTuple . \text{constraint})\} \\
\text{Out}(e) &= \texttt{FunctionMap}(e, \text{Restricted}(e))
\end{align*}
\]

We could move the restriction operation inside the \texttt{FunctionMap} and use any solver component; instead, for performance reasons, we provide a specialized solver component for qualified data flow problems. To generate a qualified data flow problem, developers provide a flow graph, primary and constraint lattice and function space components and the following predicates:

\[
\begin{align*}
\text{function} \ \text{MayDiffer}(x, y : \text{in ConstraintValue}) & \ \text{return} \ \text{Boolean}; \\
\text{function} \ \text{Restrict}(e : \text{Entity}; v : \text{in ConstraintValue}) & \ \text{return} \ \text{Boolean};
\end{align*}
\]

The interface described above has been simplified for this presentation. A more general interface produces types and operators so that a qualified problems can itself fill the role of the primary problem in specifying a new qualified analysis. This allows construction of analyzers for qualified data flow problems by incrementally composing a primary problem with a series of constraints.

Analyzers for qualified data flow problems offer increased accuracy, over the primary problem, at the expense of analysis time. In general, the cost of qualified analysis is exponential in the number of constraints, so care must be taken in defining qualified problems. Qualified analyses are similar to state propagation with a CPA. The CPA incorporates constraint automata that are used to constrain the solution to the primary state propagation problem. Our preliminary experience, described in Chapter 5, suggests that for some problems, considerable increases in the precision of
analysis results can be obtained for relatively small increases in analysis time using
the kind of constraints that qualified analyses can provide.

6.5 Experience with the Architecture

In addition to the example presented throughout the previous sections, we have
developed a number of other data flow analyzers using this architecture. In this
section we describe six of those analyzers. Some of these analyzers solve problems
that are recognizable as extensions of familiar data flow analysis problems; others are
less familiar. Our goal here is not to motivate or explain the information produced by
the analyzers, instead we hope to illustrate the way these analyzers were constructed
from the library of existing components and generators. We note that all of the
analyzers use the iterative worklist solver component.

The local dominator analyzer, presented as a running example earlier in this chap-
ter, is a traditional bit-vector problem. We were able to reuse the function space,
lattice and sparse representation to generate an analyzer for the associated post-
dominator problem by switching predecessor and successor operators and defining
the flow graph exit nodes as the start nodes for the data flow problem. Using the ar-
chitecture, this analyzer was easy to construct and proved to have good performance
for our application.

After preliminary experimentation with the communication dominator analyzers
we wanted to generalize the class of communication intervals that could be detected
in a TFG. In doing so we were able to reuse much of the existing COMDom inator
analyzer with minor modifications. Instead of communication nodes, we required
dominators and post-dominators for all send and receive nodes in each task control
flow graph of the TFG. We redefined the Gen function to set appropriate bits only for
send(receive) rather than communication nodes. For the new problems we consider
only predecessors and successors within a given task. Task-specific predecessor and
successor iterators are defined by the TFG data type. The interface to these iterators, however, does not match the flow graph interface defined by the architecture. We were able to incorporate these task-specific iterators in our analyzer by defining shallow wrapper operators that have interfaces that satisfy the architectural definitions. The sparse representation relevance predicate was modified to check that the node kind was send(receive) rather than communication. Finally, the generation of the gen-kill function space, sparse representation, and the instantiation of the iterative solver component for these problems was the same as for the communication dominator problems. It took less than an hour to design and carry out this modification and the resultant analyzers work as expected.

We implemented a preliminary version of the state propagation algorithm from Chapter 4 using the architecture. We have also designed a data flow analyzer for a qualified analysis where the primary problem is state propagation and the constraint problem models a boolean variable. This is useful in the analysis of distributed systems, since it is common to have a local state variable control the pattern of inter-task communication, for example, enforcing exclusive write access. For such systems, it is often possible to improve the precision of state propagation analysis by modeling the control variable's values and restricting propagation of finite automaton states to TFG paths that are consistent with a given value of the controlling variable. For this qualified data flow problem, which uses the constraint problem lattice and function space illustrated in Figure 6.3, \texttt{Restrict}(n, v) returns false if the node, n, is a true branch(false branch) and the constraint value, v, is false(true), otherwise it returns true. Intuitively, \texttt{Restrict} returns true if a value is inconsistent with a particular program execution state as defined by a TFG node. \texttt{MayDiffer}(x, y) returns false if $x = y$ or if one value is \texttt{both} and the other is \texttt{unknown}, otherwise it returns true. The values \texttt{both} and \texttt{unknown} are the $\bot$ and $\top$ of the boolean variable lattice presented in Figure 6.3. This qualified analysis performs an equivalent analysis as
state propagation on a CPA that incorporates a boolean variable automaton. An interesting empirical question is "how does the cost of qualified analysis versus CPA-based analysis compare?".

We have prototyped a data flow analyzer that is capable of finding dead transitions in a TIG-based Petri Net (TPN) [25]. The problem is formulated over the set of TPN transitions. The lattice is a bit-vector encoding of the set of transitions. The function space consists of a distinct function for each TPN transition; each function tests if any predecessor has its bit set in the input value, and if so, adds the bit for the current transition. This is quite similar to the dominator function spaces, except that we test the input value here in a way that is not supported by the GenKill function space generator.

Recently colleagues have begun to use the architecture to construct analyzers for checking state sequencing properties over paths in the reachability graph of a concurrent program and for computing variable liveness information to allow state space reduction of models of concurrent programs that include data state information.

### 6.6 Summary

Our experience has demonstrated that it is relatively easy to construct a new data flow analyzer using this architecture. The existing library of components reduces the software development cost involved in building analyzers. There are two cost reduction benefits; one does not have to write the code and one does not have to test and debug it. This may seem simplistic but in practice the payoff is high. When we needed to build new components we often found a number of opportunities for reuse. A good example of such reuse, discussed in section 6.5, is the use of the GenKill function space and the sparse representation in both communication dominator and post-dominator problems. We developed the state propagation analyzer in a matter of days. In contrast, it took approximately three weeks to develop a hand-crafted
analyzer for the state propagation algorithm. In fairness, this hand-crafted analyzer was built first and provided valuable insight that was used in defining the components of the analyzer generated with the architecture. Nevertheless, the reduction in programming and testing cost gained by using the architecture and components was large.

We believe the flexibility of the architecture allows relatively efficient analyzers to be constructed. While we do not expect to obtain the performance of hand-crafted analyzers, it is possible to improve the performance of selected analyzer components, as illustrated by the use of sparse representations.

Our experience to date suggests that architectural support for constructing data flow analyzers is beneficial, especially at the early stage of design when developers have yet to settle on the right combination of information to encode in the problem. We were able to rapidly prototype a variety of data flow analyzers; this has allowed us to evaluate a number of analysis design alternatives on realistic programs.

In summary, data flow analyses are a class of computations that have a well developed theoretical basis and a large base of interesting applications. As these applications are extended and new applications are considered, developers need support for evaluating the analysis design space. Our architecture, and associated library of components, allow low-cost construction of data flow analyzers to enable such evaluations.
function GenNode(n : in TFG.Node; s : in Natural)
    return BitVector.Value is
begin
    if TFG.GetKind(n) /= TFG.COM then
        return BitVector.Set(BitVector.Zero(s),
                              TFG.GetIndex(n));
    else
        return BitVector.Zero(s);
    end if;
end GenNode;

function KillNode(n : in TFG.Node; s : in Natural)
    return BitVector.Value is
begin
    return BitVector.Zero(s);
end KillNode;

package COMFunctionSpace is new GenKill(
.-- Bind lattice types and operators
    LatticeValue => BitVector.Value,
    Create => BitVector.Create,
    Print => BitVector.Print,
    Assign => BitVector.Assign,
    Meet => BitVector.Intersect,
    Join => BitVector.Union,
.-- Bind relevant flow graph operators
    Entity => TFG.Node,
    GetIndex => TFG.GetIndex,
.-- Indicator for all versus any path problem
    isAllPaths => TRUE,
.-- Bind Gen and Kill functions for node
    Gen => GenNode,
    Kill => KillNode);

Figure 6.4 Example Function Space
function IsCOMNode(n : in TFG.Node)
    return Boolean is
begin
    return TFG.GetKind(n) /= TFG.COM;
end IsCOMNode;

package COMSparseTFG is new SparseRepresentation(
    -- Bind graph and entity types
    FlowGraph => TFG.Graph,
    Entity => TFG.Node,
    MaxEntity => TFG.GetMaxNodes,
    GetIndex => TFG.GetIndex,

    -- Bind graph start/end and edge operators
    IterateStart => TFG.IterateStart,
    DoneStart => TFG.DoneStart,
    GetNextStart => TFG.GetStart,
    IterateEnd => TFG.IterateExit,
    DoneEnd => TFG.DoneExit,
    GetNextEnd => TFG.GetExit,
    IteratePreds => TFG.IteratePreds,
    DonePreds => TFG.DonePreds,
    GetNextPred => TFG.GetNextPred,
    IterateSuccs => TFG.IterateSuccs,
    DoneSuccs => TFG.DoneSuccs,
    GetNextSucc => TFG.GetNextSucc,

    -- Bind the relevancy predicate
    IsRelevant => IsCOMNode,

    -- Indicate whether to include Phi nodes
    includePhi => TRUE);

Figure 6.5 Example Sparse Representation
package COMDominators is new IterativeSolver(

    -- Bind lattice type and operators
    LatticeValue => BitVector.Value,
    Create => BitVector.Create,
    Destroy => BitVector.Destroy,
    Equal => BitVector.AreEqual,
    Assign => BitVector.Assign,
    NotAbove => BitVector.IsSubset,

    -- Bind flow graph operators
    FlowGraph => COMSparseTFG.Graph,
    Entity => COMSparseTFG.Node,
    MaxEntity => COMSparseTFG.GetMaxNodes,
    GetIndex => COMSparseTFG.GetIndex,
    IterateStart => COMSparseTFG.IterateStart,
    DoneStart => COMSparseTFG.DoneStart,
    GetNextStart => COMSparseTFG.GetStart,
    IteratePreds => COMSparseTFG.IteratePreds,
    DonePreds => COMSparseTFG.DonePreds,
    GetNextPred => COMSparseTFG.GetNextPred,
    IterateSuccs => COMSparseTFG.IterateSuccs,
    DoneSuccs => COMSparseTFG.DoneSuccs,
    GetNextSucc => COMSparseTFG.GetNextSucc,

    -- Bind function space operators
    Init => BitVector.One,
    Start => BitVector.Zero,
    Confluence => COMFunctionSpace.Confluence,
    Fmap => COMFunctionSpace.FunctionMap,

    -- Bind routines used in tracing analyzer activity
    Print => BitVector.Print,
    problemId => "TFG COM Dominators"));

Figure 6.6 Example Analyzer Generation
CHAPTER 7
FUTURE WORK AND CONCLUSION

In this chapter, we discuss a number of directions for future research that build off of the work presented in this dissertation. We conclude with a summary of the contributions of this dissertation.

7.1 Future Directions

In this section, we discuss five directions for future work: population of the data flow analysis architecture with additional components, adapting classic compiler optimizations to concurrent programs, developing techniques for further increasing the precision of FLAVERS analyses, developing techniques to enable efficient analysis of real programs, and an empirical evaluation of those techniques.

Developing Additional Analysis Components

In Chapter 6 we describe an architecture for data flow analyzers and an initial implementation of that architecture. Currently, there are four classes of components: lattices, flow graphs, function spaces, and solvers. While this architecture has proven useful, there are a number of extensions that can be made to increase its flexibility and utility.

For monotone data flow analysis frameworks it is possible to construct a parallel iterative solver. The technique, called asynchronous iteration [81], has been described for general round-robin iterative solvers. We can adapt it to iterative worklist solvers. The idea is to spawn a number of worker processes that take nodes off of the worklist
and process them as in the sequential case. Typically, such an approach will suffer because of the synchronization required on access to the worklist. The observation with this technique is that the monotonicity of the data flow analysis problem preserves the conservativeness of the solution even if we take the same node from the worklist multiple times. Thus, the solution procedure can become highly parallel and fundamentally asynchronous in nature. We are not aware of any empirical data on the effectiveness of this approach, but given the size of the problems that are encountered in FLAVERS the approach may reduce analysis cost.

We have described how qualified analyses can be accommodated with a specialized solver component. We will look to generalize the construction of analyzers for combined data flow problems. To do this we must allow for more complex types of data to flow between sub-problems and to allow problems greater freedom in communicating data to other sub-problems. This is in contrast to the distinguished violation value used in qualified analyses and the uni-directional flow of data from constraint sub-problems to the primary analysis problem.

We can further generalize the architecture by separating the information used in analysis from the flow graph itself. Thus, a new class of components, called annotations, could be defined. For example, state propagation annotations are symbols from $\Sigma$ and annotations for many of the classic compiler analysis problems are variable definition and use information.

**Classic Problems as Complete-Lattice Frameworks**

In Chapter 3 we formulated a classic sequential data flow analysis problem as a complete-lattice data flow analysis framework. This allows for more precise solutions to that problem when analyzing concurrent programs. There are a number of other classic data flow analysis problems that could be formulated as complete-lattice frameworks including:
- Reaching Definitions
- Available Expressions
- Constant Propagation
- Live Variables
- Very Busy Expressions

We can add a complete-lattice iterative solver component to the data flow architecture described in Chapter 6. A version of the TFG can be added to the architecture, as well, to provide a statement-level concurrent flow graph. These two additions will support the rapid-prototyping of implementations for these classic analysis problems over concurrent flow graphs for Ada programs. This will provide a vehicle for empirical evaluation of the improvements in precision that are promised by complete-lattice data flow frameworks.

**Increasing Precision in FLAVERS**

Our basic TFG and state propagation algorithm correspond to a conservative test for whether a pattern of program events expressed in a \( \forall(\exists) \) QRE is exhibited by all(no) program executions. As mentioned in Chapter 4, we improve precision by encoding and enforcing necessary conditions for executability in the TFG and state propagation algorithm. The TFG, for instance, can be transformed to eliminate portions of the TFG that are guaranteed to violate some necessary condition for program executability. The state propagation algorithm, for instance, can incorporate information that restricts the flow of information to portions of the TFG that do not violate some necessary condition.

In order to develop additional techniques for increasing precision we need to understand the kinds of information about program executions that can be encoded as
necessary conditions and enforced cost-effectively during analysis. An execution of a program is modeled as a sequence of program events; such a sequence is defined by the set of events in the trace, the number of times each event occurs, and the ordering of each instance of an event. We refer to information about events, event count and event order as sources of program information. In the following, we list a number of ways that we can get at program information that is related to a given source.

**Events** The events that occur during a program execution.

- set of events in an execution
- set of dominators/post-dominators
- set of precedence/post-precedence
- events that must happen in same execution
- events that are impossible in same execution

**Event Count** The number of times an event occurs during a program execution.

- absolute number of event instances
- number of event instances relative to another event
- bounded number of event instances

**Event Ordering** The order of instances of events during a program execution.

- with respect to all other events
- with respect to a specific event
- precedes any instance
follows any instance
precedes all instances
follows all instances

We have designed refinements and feasibility constraints for some of these, but, have not yet implemented them.

A rich, but fundamentally untrustworthy, source of information is the user of the analysis system. Users often have detailed information about the specification, design and implementation of the program under analysis. We may allow users to describe certain patterns of behavior as unexecutable. If the description is made as a regular expression, then it could be encoded as a feasibility constraint. If the description is, for example, a collection of pairs of nodes that cannot execute concurrently, then it could be used to refine the TFG. As long as the user is made aware of the assumptions upon which a given analysis result is based, then this could provide significant leverage in the analysis of large complex applications.

Efficient Analysis of Real Programs

Ultimately, our goal is to provide automated analysis techniques that produce precise analysis results for a wide range of production software in time that is consistent with typical software development processes. We are a long way from even knowing if that is possible, much less, producing such a technique. The road to that understanding is empirical evaluation of analysis techniques.

Empirical evaluation has two major benefits. Trying an analysis on a realistic problem gives feedback on the parts of the approach that work and those that don't. Learning about the structure of realistic problems can provide fuel for the development of improved analyses; for example, early empirical evaluations with FLAVERS led to the development of communication interval refinement.
To enable application of FLAVERS to real applications we intend to work on:

- enhancing and tuning the FLAVERS/Ada toolset, extending FLAVERS/Ada to perform inter-procedural analyses, assessing the feasibility of constructing a version of FLAVERS for C++ Client/Server applications, providing support to users for diagnosing the source of inconclusive results, and making it easy for users to specify properties as QREs.

Tuning the toolkit will involve work at a number of levels. The evaluation described in Chapter 5 highlights a number of components as bottlenecks to low cost analysis; the question is whether these components are necessarily high-cost or whether they are amenable to algorithmic or code-level improvement. We intend to assess these questions with a performance evaluation of each of the components of FLAVERS/Ada. We know that constructing large CPAs and using them in state propagation analysis is expensive. An alternative is to formulate FCs as constraint problems of a qualified analysis as described in Chapter 6. The cost of constructing such a qualified analysis will be much less than the cost of CPA construction, since the qualified analysis does not explicitly build the product problem. The cost of performing such a qualified analysis, however, could be significant. Only empirical evaluation will tell us which approach is more cost-effective for real analysis problems. Incremental data flow analysis can be applied for re-analysis, after either the program or QRE changes; we are aware of a number of special cases for which incremental analysis is able to reuse most of the analysis results from the previous run, for example, extending a QRE specification with a suffix language. A number of other interesting potential performance improvements are possible.

Inter-procedural FLAVERS analysis is tricky. The TFG is polynomial in the size of the program because it explicitly represents only pair-wise event interleavings. If we were to annotate TFG nodes with regular languages or even strings that summarize the behavior of sub-programs the cost of processing a node would increase dramati-
cally. Basically, at a node we would have to compute the possible interleavings of the string at that node with all the nodes strings for all other nodes that are connected to it by a MIP edge. We would then need to apply the PA transition function to each of those interleavings to get the value at the node. The cost of processing a single node would grow to become exponential in the number of tasks in the program, since, in the worst-case, a node can have MIP edges connecting it to nodes in all tasks in the program. Clearly, we cannot use summaries in this way. Inlining procedures, however, is an attractive alternative. The drawback to inlining is an explosion in the size of the flow graph itself. In a compiler, we need to keep all of the information about a sub-program so that we can generate correct code; FLAVERS does not have the same requirement. We need only inline the information that is relevant for the checking of a property. Thus, we could perform an alphabet specific inlining operation that only inlines the projection of a sub-program over a given alphabet. It is quite likely that a majority of sub-programs will have an empty projection for a given alphabet. While the resulting TFG would be a conservative representation of a programs complete inter-procedural behavior, the cost of performing the inlining for every alphabet may be too high. Just as alphabet refinement can be performed in stages with successively smaller alphabet, so too could alphabet inlining. It's a classic space/time tradeoff. We can pre-compute a number of different versions of inlined TFGs from which we can choose the "smallest" one that is conservative for the property we wish to check.

FLAVERS analyses produce very detailed information about prefixes (or suffixes for backwards analysis) of program executions and the extent to which they satisfy or fail to satisfy a property under analysis. This information can be harnessed, and presented to users in the form of visualizations of TFG fragments, to aid in the location of sources of inconclusive results. The idea is that a user would indicate which PA states she was interested in monitoring, then during state propagation analysis, whenever that state is entered at a node the analysis system could display
the set of TFG paths that could lead to that node. PA state information could be used
to constrain the set of reported paths to eliminate spurious paths. This technique
could be more generally useful in other data flow analyses. We are planning to develop
this capability, in conjunction with flow architecture from Chapter 6, as a teaching
aid.

Our experience to date with FLAVERS makes it clear that users will require
assistance in specifying complex properties. It is true that regular expressions and
finite state automata are familiar to a large number of software developers; it is also
true that the size and complexity of QREs quickly grows to the point where subtle
errors are easily introduced.

The problem of annotating a program with an alphabet of events to be reasoned
about is at present labor intensive drudgery. A mixture of three different approaches
to specifying program events would greatly improve the situation. We can auto-
mate the definition of classes of program events; this is already done for inter-task
communication events. These classes could include variables, procedures, packages
specifications, for which a default set of events is constructed and associated with
appropriate flow graph nodes. A more flexible approach is to use an event definition
language. Cecil [60] uses a simple language to create annotations. We envision a
slightly more powerful language where the user can describe complex program pat-
terns to be annotated by a single event symbol. Finally, the user should be allowed
to annotate the program explicitly. Unlike the current FLAVERS/Ada tools, which
support annotations as source comments, a simple GUI which allows users to "drag"
event symbols from a key and "drop" them on parts of the source code would provide
a friendly interface, the same functionality, and would disassociate the annotations
from the source code.

We have encountered a number of specifications that contain repeated patterns.
Incorporation of a simple macro language into the QRE specification language could
go a long way to making QREs more general and reusable. It would also reduce the size of specifications, thereby reducing the opportunity for introducing errors.

Although not a specification issue, we intend to investigate different techniques for modeling blocking in the TFG. If we can model the potential for a communication statement to block, then we have the opportunity to address questions of program deadlock.

**Empirical Evaluation**

The cost-effectiveness of techniques for improving precision and reducing the cost of FLAVERS analyses on real programs, in general, cannot be judged analytically. We will implement and experiment with the approaches described above in the context of the existing FLAVERS/Ada toolset. We intend to evaluate these techniques over a range of programs and properties. Initially, we can look at the growing stable of examples in the concurrency analysis literature. As FLAVERS/Ada matures we will be able to analyze real applications. There is a wealth of large complex sequential Ada programs, within the Arcadia consortium alone, to experiment with. Large complex concurrent Ada programs are less common, but some are becoming available.

We judge the effectiveness of the techniques with respect to their ability to provide conclusive analysis results. While it may be possible to provide analysis results with a small enough number of spurious results so as to be useful to software developers, we do not attempt to quantify precision in that way. We will perform a more thorough empirical study using a similar set of cost measures as the evaluation in Chapter 5. We will use this data to evaluate the:

- Cost-effectiveness of refinements
  - in isolation
  - in combination with other refinements
- in combination with FCs

- Cost-effectiveness of feasibility constraints
  - in isolation
  - in combination with other refinements
  - in combination with FCs

In addition, we intend to provide sufficient data to support comparison with other analysis techniques.

**Related Future Directions**

The concurrency analysis community has been moving towards more in-depth empirical evaluation of analysis techniques. Conducting comparative empirical evaluations of different analysis techniques is a significant challenge, but it can provide information about the relative cost-effectiveness of analysis techniques.

As a community our goal should be to provide cost-effective analysis techniques that are of use to practitioners. It is unlikely that a single technique will be found that meets all of the needs of practitioners. Thus, it will become increasingly important to understand for what kinds of programs and properties each analysis technique is most effective. Supported by broad empirical evaluation there is some hope that this understanding will be generalizable to new applications. We need to develop heuristics that guide the selection and tuning of analyses for a given program and property. With more information about the program or property under analysis we may be able to develop more effective these heuristics. For example, if we focus on an application domain, such as client-server systems, the architecture of programs in that domain may lend itself to a particular kind of analysis or we may be able to apply special kinds of heuristics that use domain information. Therefore, a ripe area
for research is in the empirical evaluation of large complex applications that are all
drawn from a single software domain.

7.2 Conclusions

We have extended the theoretical foundations of data flow analysis frameworks
to accommodate concurrent program analysis. By exploiting knowledge of data flow
analysis and the construction of data flow analyzers we were able to construct a
domain-specific architecture to support the rapid construction of practical data flow
analyzers. Leveraging off of these advances we have developed FLAVERS, a family
of polynomial-time conservative data flow analysis algorithms that support the anal-
ysis of explicitly stated correctness properties of concurrent programs. FLAVERS
differs significantly from other approaches to static analysis of concurrent programs
by allowing flexibility and variety in the amount of program information that is en-
coded into the analysis. This provides users with the ability to trade-off analysis
cost and the precision of analysis results. We have implemented FLAVERS/Ada,
a version of FLAVERS for Ada tasking programs. Our experience with applying
FLAVERS/Ada to a collection of programs and properties suggests that, for a vari-
ety of interesting non-trivial properties of concurrent programs, FLAVERS/Ada can
provide cost-effective and precise analysis.
In this appendix, we describe the design of a toolset that supports FLAVERS analysis on programs written in Ada. The toolset is built out of a collection of general and special purpose reusable components; we describe these components and their interfaces. The analysis artifacts that are manipulated by the toolset are maintained as persistent data; we describe our use of Pleiades [73] to accomplish this.

A.1 A Technology Base

The development of FLAVERS/Ada was done in the context of the Arcadia [76] project. A significant contribution of this project has been the development and distribution of a variety of reusable data types, object management and language processing capabilities.

Reusable Components

A large collection of general reusable abstract data types (ADT)s have been constructed to support Arcadia development. These include generic implementations of, for example, lists, queues, stacks, hash tables. Addition components have been developed to alleviate some limitations of the Ada language, e.g., an implementation of dynamically sized strings, and to provide abstractions to interface with operating system services, e.g., POSIX and command line interfaces.

The FLAVERS/Ada toolset makes widespread use of these components; their availability greatly reduced the time required to develop the toolset.
Object Management

As is clear from the description of FLAVERS in Chapter 4 there are a variety of artifacts created and manipulated during analysis. These include TFGs, PAs, VAs, TAs, and CPAs. To reduce overhead costs a FLAVERS analysis can reuse artifacts that have been created during previous analyses.

The Pleiades object management system provides a rich set of capabilities for defining, storing, and retrieving persistent representations of complex data types. Pleiades extends the set of "built-in" types available to Ada programmers to include: graph types, collections and relations.

We use Pleiades in the construction of a TFG ADT, a deterministic finite state automaton (DFSA) ADT, an alphabet ADT, a regular expression ADT, and a QRE ADT. An application-level ADT is constructed by defining a low-level ADT through Pleiades. For example, we define the nodes and edges of a TFG including all of the attributes mentioned in Chapter 4 in Pleiades' declarative input language; Pleiades generates an implementation for the TFG that provides constructors, destructors, attribute access functions, and persistence operations. Using this low-level ADT essentially as language built-in functionality, we implement the semantics of the application-level TFG; the result is an application-level ADT with persistence. Persistence in Pleiades is by instance; we can select the TFGs we want to keep and allow the others to be deleted. Thus, we can gain the advantages of artifact reuse without incurring unnecessary an space penalty.

While Pleiades generated low-level implementations provide reasonable performance, they are not tuned to the needs of particular applications. To address this, Pleiades provides the ability for users to define attributes of persistent objects that are not persistent. We take advantage of this feature to construct very efficient in-memory representations for a number of ADTs. Persistence is accomplished through a process of faulting data from(to) long-term storage to(from) memory; the design of Pleiades
low-level representations isolates this faulting to two operations \texttt{GetNPR} and \texttt{GetPID}. We define our own versions of these operations by wrapping the Pleiades generated operations, constructing the appropriate in-memory representation, and storing that representation as a transient attribute. The result is a very high-performance persistent implementation of an ADT; overhead costs are minimized because the transformation at faulting time happen once per tool invocation.

Persistent objects are stored in a \textit{repository}. Applications can manipulate objects stored in a repository in the context of a \textit{session}, a transaction-like construct. Once inside a session operations on persistent and non-persistent appear the same at the application-level; this seamless treatment of persistent and non-persistent data is a significant feature of Pleiades.

\section*{Language Processing}

The Arcadia project has developed a collection of tools for performing syntactic, semantic and control flow analysis of programs written in Ada. Unlike existing Ada compilation systems, the Arcadia language processing capabilities provide well-engineered ADTs for the data produced by those capabilities. ADTs have been defined for IRIS graphs\cite{27}, an abstract syntax tree, and for CFGs, statement-level control flow graphs. These are all persistent first-class types that come with a collection of tools for constructing and visualizing instances of the types.

\section*{A.2 FLAVERS/Ada Components}

One of the design goals of FLAVERS/Ada was to create an implementation that could easily be extended. Our approach to doing this was to define a collection of ADTs that provide general capabilities and build increasingly higher-level application specific capabilities on top of them. Figure A.1 depicts this hierarchy of types. Arrows depict direct dependences between types, for example, TFGs having CFG attribute
values or TFG nodes having alphabet symbols as attribute values. At the base of
the hierarchy are the Pleiades generated low-level types. All of the FLAVERS/Ada
components make use of Pleiades, as are many of the reusable Arcadia components.
The graph types IRIS, CFGs, and TFGs are built on top of one-another; TFGs do
not directly refer to IRIS graphs, rather, TFGs refer to CFGs which in turn refer to
IRIS graphs. A somewhat more complicated hierarchy arises from the relationship
between QREs, DFSAs, regular expressions and the alphabet ADT, where alphabets
are shared between each of the higher-level types. All of these are nameable, first-
class, persistent data types.

An alphabet represents $\Sigma$ for a program. It is a collection of symbols that can be
manipulated either in terms of an ASCII string representation or a identifying index
that is unique across an alphabet.

A regular expression is defined over an alphabet. It represents patterns of alphabet
symbols and can be manipulated as either an ASCII string or as a parse-tree.

A DFSA is a graph type where edges are attributed with symbols from an al-
phabet. The DFA ADT provides constructors, output routines, and access to states
(nodes) and transitions (edges) and their attributes. In addition, higher-level operations are provided for minimizing and forming the product of two DFSA. This type is used to represent a variety of FLAVERS artifacts including PA, VA, TA and CPA.

A QRE consists of a regular expression, the DFSA representing the PA, and a quantifier, all or none.

The TFG is perhaps the most complicated of the data types. It provides a variety of interfaces to allow analyses to view the TFG in the most appropriate form. We can think of the TFG as a collection of CFGs, one for each program task, with additional nodes that represent inter-task communication and synchronization. Some analyses, for example the domination computations in CI refinements, view only a single task CFG at a time. This is supported by a flexible iterator mechanism that allows a user at iterator creation time to constrain the kinds of nodes that are produced during iteration; for example, nodes not in a given task. Similarly, analyses such as statement precedence, B4 or After are not dependent on the MIP in the TFG; those analyses can use iterators that ignore MIP edges. The state propagation algorithm must consider the entire TFG.

Construction of the TFG can be performed in stages; the construction of MIP edges can be delayed until after all TFG refinements have been applied. This is a significant advantage when analyzing programs with large unrefined TFGs and properties with small alphabets. Ultimately, the TFGs submitted to state propagation will be significantly smaller than the unrefined TFG, and will have many fewer MIP edges.

The alphabet, DFSA and TFG ADTs all share a common capability: indexing. Instances of these types are created and manipulated as for any other type. At some point during program execution we may decide that no further changes to an instance are required and from that point on the instance can be treated as a constant. In these cases, we can significantly improve the performance of algorithms
that manipulate such constant data. We do this by assigning to each sub-component of the instance an identifying natural number that is unique within the instance. For example, alphabet symbols, TFG nodes and edges, and DFSA states and transitions each have unique identity values. We can use these identity values to produce very efficient representations of those types; in particular, these identity values enable the use of bit-vector formulations of data flow analysis problems. Indexing is performed at most once per instance and can never be undone. Copying an indexed instance, however, will produce an equivalent unindexed instance.

A.3 FLAVERS/Ada Tools

Analysis of an Ada program with respect to a property specified as a QRE using FLAVERS/Ada involves running a sequence of tools. Figure A.2 depicts the meaningful execution orderings of tools\(^1\). A directed arrow means that execution of the tool at the source of the arrow precedes execution of the tool at its destination; for example, \texttt{iris to cfg} precedes \texttt{cfgs to etg}. The simplest ordering involves no refinements or feasibility constraints; this corresponds to the basic analysis described in Chapter 5.

In the remainder of this section we describe each of the FLAVERS/Ada tools. The presentation is organized by ADT. Most tools accept \texttt{verbose}, \texttt{debug}, and \texttt{repository} options that are used to enable levels of diagnostic output and for indicating the repository in which analysis artifacts reside or should be created.

TFG Tools

\texttt{cfgs to etg} constructs a TFG from a collection of control flow graphs. This tool implements phases 1 and 2 of Algorithm 3. In the current implementation TFG

\(^1\text{We have written this tool description using the actual names of the executables. For historical reasons TFGs were called ETGs. For the purpose of this section we can interpret all instances of ETG as TFG.}\)
Figure A.2 FLAVERS/Ada Tool Orderings
node labels are constructed from comment information stored in the IRIS graph from which the CFGs are derived. Work is underway to disassociate labels, or annotations, from the graph; once completed this will remove the dependence on IRIS graphs. A limitation of the current implementation is that all tasks in the program must be contained in a single IRIS graph. This is a workaround for a limitation in the IRIS comments manager implementation.

Usage: cfgs_to_etg [-verbose] [-debug]

[-repository <name> (defaults to "default.repository")]
[-etg <name> (defaults to "default.etg")]
[-iris <name> (defaults to "default.body")]
<cfg-name> [<cfg-name> ...]

Add_mip_edges constructs the set of MIP edges for a TFG. The edges are constructed based on the definitions given in Chapter 4. This tool implements phase 3 of Algorithm 3. Information from CI refinement is applied to restrict construction of MIP edges which are infeasible. Note that the addition of MIP edges is done in-place.

Usage: add_mip_edges [-verbose] [-debug]

[-repository <name> (defaults to "default.repository")]
<etg-name>

Etg_directory lists the TFGs in the given repository.

Usage: etg_directory [-verbose] <repository-name>

Etg_to_text produces either an ASCII dump of a TFG or an VCG[68] input file. If the color option is set the vcg file will include color information for distinguishing node and edges by kind.

Usage: etg_to_text [-verbose] [-debug] [-vcg] [-color]

[-repository <name> (defaults to "default.repository")]

235
QRE Tools

Spec_to_qre parses a QRE specification and constructs the PA. The QRE that contains this information is stored under the given name.

Usage: spec_to_qre [-verbose] [-debug]  
[-repository <name> (defaults to "default.repository")]
[-qre <name>]
<spec-file-name>

Qre_directory lists the QREs in the given repository.

Usage: qre_directory [-verbose] <repository-name>

Qre_to_text produces an ASCII dump of a QRE including: the regular expression, the quantifier and the PA.

Usage: qre_to_text [-verbose] [-debug]  
[-repository <name> (defaults to "default.repository")]
[-outputfile <name> (defaults to standard output)]
<qre-name>

DFSA Tools

DFSAs are constructed by a variety of other tools. Since they are all represented with a common type we provide common utility tools.

Dfa_directory lists the DFSAs in the current repository.

Usage: dfa_directory [-verbose] <repository-name>
Dfa_to_text produces an ASCII dump of a DFSA or an VCG input file; as with the TFG tool color output can be produced with the color switch. If the DFA is a CPA then the cpa switch should be set.

   [-repository <name> (defaults to "default.repository")]
   [-outputfile <name> (defaults to standard output)]
   <automaton-name>

Refinement Tools

Alpha_refine performs alphabet refinement for a given TFG based on the alphabet of a given QRE. The refined TFG is stored under the specified name. The degree of refinement can be restricted. Using the preserve switch maintains sufficient structural information so that CI refinement can be performed subsequently. Using the keep switch keeps all COM nodes in the refined TFG; COM nodes whose labels are not in the QRE alphabet are re-labeled to $\tau$. In the current implementation the keep option is required for correct processing of accept statements with bodies.

   [-repository <name> (defaults to "default.repository")]
   [-name <name> (defaults to "<qre-name><etg-name>")]
   -qre <name>
   <etg-name>

Ci_refine preforms all the forms of communication interval refinement described in Chapter 4. Unlike, that description this implementation is not selective in choosing pairs of tasks to consider, rather it exhaustively considers the possibility of communication intervals between all pairs of program tasks. Thus, it is significantly more expensive than the more selective approach. The refined TFG is stored under the
specified name. Information about infeasible MIP edges is stored along with the TFG; this allows `add_mip_edges` to restrict the construction of infeasible MIP edges.

Usage: `ci_refine [-verbose] [-debug]

[-repository <name> (defaults to "default.repository")]
[-name <name> (defaults to "ci<etg-name>")]<etg-name>

Etg_cpa_refine unifies the CPA and TFG alphabets. This is necessary when the CPA incorporates a TA. As described in Section 4.6.2.1 the TA uses a node-alphabet. Prior to state propagation the alphabets must be made equivalent. This involves relabeling TFG nodes with node-alphabet symbols.

Usage: `etg_cpa_refine [-verbose] [-debug]

[-repository <name> (defaults to "default.repository")]
-cpa <cpa-name>
-etg <etg-name>

Feasibility Constraint Tools

Etg_to_ta constructs a TA from the CFG of the task whose name is provided with the `ta` option. The TA is stored under the same name. This tool implements Algorithm 6.

Usage: `etg_to_ta [-verbose] [-debug]

[-repository <name> (defaults to "default.repository")]
-ta <name>
<etg-name>

Build_va constructs a VA for the variable with the specified name. This tool implements Algorithm 7 and a similar bounded counter VA construction algorithm.
In this implementation VAs are constructed from a template and may, consequently, include more symbols the TFG. This overestimate has a negligible impact on performance.

Usage: `build_va [-verbose] [-debug]` 
[-repository <name> (defaults to "default.repository")]
[-counter <max-value>]
[-va <name> (defaults to "<variable-name>.ya")]
<variable-name>

`Build.cpa` uses the DFSA product operation to construct the product automaton of the PA, for the specified QRE, and the specified TAs and VAs. The resulting automaton is stored under the name provided with the `cpa` option. The CPA can is minimized by default; this behavior can be overridden by setting the `nominimize` switch. This tool runs Algorithm 8 prior to construction of the CPA in order to preserve the semantics of the PA in the presence of TA node-alphabet symbols.

Usage: `build_cpa [-verbose] [-debug] [-nominimize]` 
[-repository <name> (defaults to "default.repository")]
[-qre <name> (defaults to "default.qre")]
[-cpa <name> (defaults to "default.cpa")]
[-ta <name> (defaults to "default.ta")]
[-va <name> (defaults to "default.va")]

State Propagation Tools

State propagation is performed by the `props` tool.

Usage: `props [-verbose] [-debug]` 
[-repository <name> (defaults to "default.repository")]
[-outputfile <name> (defaults to standard output)]
A.4 Experience

In addition to being a vehicle for empirically evaluating the FLAVERS analysis approach, FLAVERS/Ada was a significant test of the Arcadia object management infrastructure. The availability of existing reusable components and capabilities promised to significantly reduce the effort required to develop FLAVERS/Ada. Prior to development, however, there were concerns about the capacity and efficiency of the infrastructure upon which the tools were to be built.

Our experience has been that the cost develop of FLAVERS/Ada was, in fact, significantly reduced by building on top of Arcadia infrastructure. Some of the advantages of the existing toolset, for example the reuse across tools of analysis artifacts, would not have been possible without it.

During the course of the empirical evaluations described in Chapter 5 we were able to demonstrate that the Arcadia infrastructure is capable of supporting the creation and manipulation of very large artifacts in a reasonable amount of time. Part of this was due to the algorithm specific in-memory data structures that were created, but artifacts with hundreds of thousands of components were stored and retrieved in order to construct those in-memory structures.

As FLAVERS/Ada matures and is applied to the analysis of larger and more complex programs, we expect it to further stress the infra-structure. By providing feedback on performance and capacity bottlenecks FLAVERS/Ada will motivate enhancement of this infra-structure to the benefit of all applications.
APPENDIX B
DATA FOR EMPIRICAL EVALUATIONS

This appendix provides the raw data from the scalable programs that were presented in Chapter 5. The categories of cost measures and their units are the same as in Chapter 5.

B.1 A Simple Protocol Problem

To give a sense of the growth of TFG edges with nodes, we plot the rate of growth of the number of TFG edges versus TFG nodes in Figure B.1. The data used in this plot is for alphabet refined TFGs. The number of edges appears to be growing nearly quadratically with the number of nodes. The raw number of edges, however, is considerably less than $N^2$.

Single Header Packet

The raw data for checking the header-packet property on 5 different sizes of the protocol program are given in Table B.1. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.2 and B.3. For state propagation analysis we include both run-time and the platform independent measures of work.

Multiple Header Packet

The raw data for checking a version of a header-packet property for each task in 5 different sizes of the protocol program are given in Table B.2. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.4
Figure B.1 Refined TFG Edges for Protocol

### Table B.1 Raw Data for header-packet

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>CI Alpha TFG Nodes</th>
<th>CI Alpha TFG Edges</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>43</td>
<td>26</td>
<td>101</td>
<td>3</td>
<td>8.5</td>
<td>0.6</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>71</td>
<td>44</td>
<td>291</td>
<td>3</td>
<td>11.4</td>
<td>0.9</td>
<td>2.5</td>
<td>3.1</td>
</tr>
<tr>
<td>10</td>
<td>127</td>
<td>80</td>
<td>959</td>
<td>3</td>
<td>14.5</td>
<td>1.2</td>
<td>3.4</td>
<td>5.8</td>
</tr>
<tr>
<td>18</td>
<td>239</td>
<td>152</td>
<td>3477</td>
<td>3</td>
<td>25.8</td>
<td>2.7</td>
<td>9.2</td>
<td>31.7</td>
</tr>
<tr>
<td>34</td>
<td>463</td>
<td>296</td>
<td>13031</td>
<td>3</td>
<td>49.2</td>
<td>8.2</td>
<td>100.8</td>
<td>365.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP PROPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
</tr>
<tr>
<td>1.1</td>
</tr>
<tr>
<td>2.3</td>
</tr>
<tr>
<td>6.5</td>
</tr>
<tr>
<td>32.5</td>
</tr>
<tr>
<td>247.1</td>
</tr>
</tbody>
</table>

242
Figure B.2 Individual Cost Components for Single header/packet

Figure B.3 State Propagation for Single header/packet
Table B.2 Raw Data for Multiple header-packet

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>CI Alpha Nodes</th>
<th>TFG Edges</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>43</td>
<td>26</td>
<td>101</td>
<td>3</td>
<td>8.5</td>
<td>1.2</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>71</td>
<td>44</td>
<td>291</td>
<td>3</td>
<td>11.4</td>
<td>3.6</td>
<td>2.5</td>
<td>3.1</td>
</tr>
<tr>
<td>10</td>
<td>127</td>
<td>80</td>
<td>959</td>
<td>3</td>
<td>14.5</td>
<td>9.6</td>
<td>3.4</td>
<td>5.8</td>
</tr>
<tr>
<td>18</td>
<td>239</td>
<td>152</td>
<td>3477</td>
<td>3</td>
<td>25.8</td>
<td>43.2</td>
<td>9.2</td>
<td>31.7</td>
</tr>
<tr>
<td>34</td>
<td>463</td>
<td>296</td>
<td>13031</td>
<td>3</td>
<td>49.2</td>
<td>262.4</td>
<td>100.8</td>
<td>365.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>1.1</td>
<td>3.0</td>
<td>136</td>
</tr>
<tr>
<td>2.3</td>
<td>10.4</td>
<td>480</td>
</tr>
<tr>
<td>6.5</td>
<td>47.2</td>
<td>1728</td>
</tr>
<tr>
<td>32.5</td>
<td>504.0</td>
<td>6528</td>
</tr>
<tr>
<td>247.1</td>
<td>12265.6</td>
<td>25344</td>
</tr>
</tbody>
</table>

and B.5. For state propagation analysis we include both run-time and the platform independent measures of work.

**Composite Header Packet**

The raw data for checking the composite header-packet property for 5 different sizes of the protocol program are given in Table B.3. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.6 and B.7. For state propagation analysis we include both run-time and the platform independent measures of work.

**No Orphans**

The raw data for checking the no orphans property for 5 different sizes of the protocol program are given in Table B.4. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.8 and B.9. For state propagation analysis we include both run-time and the platform independent measures of work.
Figure B.4 Individual Cost Components for Multiple header-packet

Figure B.5 State Propagation for Multiple header-packet
Table B.3 Raw Data for Composite header-packet

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>CI Alpha TFG Nodes</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>43</td>
<td>26</td>
<td>89</td>
<td>4</td>
<td>19.3</td>
<td>1.0</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>71</td>
<td>44</td>
<td>162</td>
<td>6</td>
<td>21.3</td>
<td>1.5</td>
<td>5.5</td>
</tr>
<tr>
<td>10</td>
<td>127</td>
<td>80</td>
<td>447</td>
<td>10</td>
<td>30.3</td>
<td>3.8</td>
<td>7.1</td>
</tr>
<tr>
<td>18</td>
<td>239</td>
<td>152</td>
<td>1399</td>
<td>18</td>
<td>82.1</td>
<td>32.2</td>
<td>40.7</td>
</tr>
<tr>
<td>34</td>
<td>463</td>
<td>296</td>
<td>13031</td>
<td>34</td>
<td>106.2</td>
<td>807.6</td>
<td>55.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>3.7</td>
<td>4.4</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>5.4</td>
<td>100</td>
</tr>
<tr>
<td>8.5</td>
<td>8.7</td>
<td>180</td>
</tr>
<tr>
<td>39.5</td>
<td>43.4</td>
<td>340</td>
</tr>
<tr>
<td>925.4</td>
<td>444.2</td>
<td>992</td>
</tr>
</tbody>
</table>

Figure B.6 Individual Cost Components for Composite header-packet
Figure B.7 State Propagation for Composite header-packet

Table B.4 Raw Data for no-orphans

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>CI Alpha TFG Nodes</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>43</td>
<td>26</td>
<td>69</td>
<td>4</td>
<td>17.8</td>
<td>0.7</td>
<td>5.1</td>
</tr>
<tr>
<td>6</td>
<td>71</td>
<td>44</td>
<td>123</td>
<td>4</td>
<td>21.0</td>
<td>0.7</td>
<td>5.3</td>
</tr>
<tr>
<td>10</td>
<td>127</td>
<td>80</td>
<td>231</td>
<td>4</td>
<td>25.8</td>
<td>0.7</td>
<td>7.4</td>
</tr>
<tr>
<td>18</td>
<td>239</td>
<td>152</td>
<td>447</td>
<td>4</td>
<td>40.0</td>
<td>0.7</td>
<td>14.2</td>
</tr>
<tr>
<td>34</td>
<td>463</td>
<td>296</td>
<td>879</td>
<td>4</td>
<td>85.3</td>
<td>0.7</td>
<td>53.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>4.2</td>
<td>4.8</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>5.2</td>
<td>146</td>
</tr>
<tr>
<td>7.9</td>
<td>8.1</td>
<td>270</td>
</tr>
<tr>
<td>23.1</td>
<td>18.1</td>
<td>518</td>
</tr>
<tr>
<td>91.9</td>
<td>53.6</td>
<td>1014</td>
</tr>
</tbody>
</table>
Figure B.8 Individual Cost Components for no-orphans

Figure B.9 State Propagation for Composite no-orphans
Figure B.10 Refined TFG Edges for Readers/Writers

B.2 The Readers/Writers Problem

To give a sense of the relationship between the number of TFG nodes and edges, we plot the rate of growth of the number of TFG edges versus TFG nodes in Figure B.10. The data for this plot comes is for alphabet refined TFGs. The number of edges appears to be growing nearly quadratically with the number of nodes. The raw number of edges, however, is considerably less than $N^2$.

Exclusive Read Write

The raw data for checking the exclusive-read-write property on 5 different sizes of the readers/writes program are given in Table B.5. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.11 and B.12. For state propagation analysis we include both run-time and the platform independent measures of work.
Table B.5 Raw Data for exclusive-read-write

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Edges</th>
<th>PA States</th>
<th>VA States</th>
<th>CPA States</th>
<th>LP</th>
<th>QRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>65</td>
<td>36</td>
<td>281</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>22.2</td>
<td>0.8</td>
</tr>
<tr>
<td>9</td>
<td>105</td>
<td>65</td>
<td>887</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>24.2</td>
<td>0.8</td>
</tr>
<tr>
<td>17</td>
<td>185</td>
<td>117</td>
<td>3107</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>43.3</td>
<td>0.8</td>
</tr>
<tr>
<td>33</td>
<td>345</td>
<td>221</td>
<td>11579</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>68.7</td>
<td>0.8</td>
</tr>
<tr>
<td>65</td>
<td>665</td>
<td>429</td>
<td>44651</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>152.7</td>
<td>0.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alpha</th>
<th>VA</th>
<th>CPA</th>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
<td>δ Evals</td>
<td>con</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>0.3</td>
<td>4.0</td>
<td>4.4</td>
<td>5.7</td>
<td>190</td>
</tr>
<tr>
<td>6.4</td>
<td>0.3</td>
<td>4.3</td>
<td>7.0</td>
<td>10.2</td>
<td>350</td>
</tr>
<tr>
<td>11.5</td>
<td>0.4</td>
<td>4.4</td>
<td>22.5</td>
<td>44.6</td>
<td>670</td>
</tr>
<tr>
<td>25.9</td>
<td>0.3</td>
<td>7.8</td>
<td>129.5</td>
<td>370.4</td>
<td>1310</td>
</tr>
<tr>
<td>91.7</td>
<td>0.4</td>
<td>13.6</td>
<td>1121.0</td>
<td>4391.0</td>
<td>2590</td>
</tr>
</tbody>
</table>

Figure B.11 Individual Cost Components for exclusive-read-write
Write First

The raw data for checking the write-first property for 5 different sizes of the readers/writers program are given in Table B.6. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.13 and B.14. For state propagation analysis we include both run-time and the platform independent measures of work.

No Read Upon Write

The raw data for checking the no-read-upon-write property for 5 different sizes of the readers/writers program are given in Table B.7. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.15 and B.16. For state propagation analysis we include both run-time and the platform independent measures of work.
Table B.6 Raw Data for write-first

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Alpha TFG Edges</th>
<th>PA States</th>
<th>VA States</th>
<th>CPA States</th>
<th>LP</th>
<th>QRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>65</td>
<td>36</td>
<td>194</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>21.9</td>
<td>0.6</td>
</tr>
<tr>
<td>9</td>
<td>105</td>
<td>62</td>
<td>424</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>25.2</td>
<td>0.5</td>
</tr>
<tr>
<td>17</td>
<td>185</td>
<td>114</td>
<td>1512</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>36.6</td>
<td>0.6</td>
</tr>
<tr>
<td>33</td>
<td>345</td>
<td>218</td>
<td>5704</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>63.9</td>
<td>0.7</td>
</tr>
<tr>
<td>65</td>
<td>665</td>
<td>426</td>
<td>22152</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>135.5</td>
<td>0.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alpha</th>
<th>VA</th>
<th>CPA</th>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
</tr>
<tr>
<td>5.4</td>
<td>0.4</td>
<td>4.2</td>
<td>4.3</td>
<td>5.2</td>
<td>80</td>
</tr>
<tr>
<td>6.7</td>
<td>0.4</td>
<td>4.6</td>
<td>6.3</td>
<td>7.2</td>
<td>150</td>
</tr>
<tr>
<td>10.9</td>
<td>0.3</td>
<td>5.6</td>
<td>14.8</td>
<td>16.4</td>
<td>290</td>
</tr>
<tr>
<td>26.4</td>
<td>0.4</td>
<td>7.6</td>
<td>62.1</td>
<td>107.4</td>
<td>570</td>
</tr>
<tr>
<td>110.2</td>
<td>0.3</td>
<td>12.7</td>
<td>521.7</td>
<td>1127.5</td>
<td>1130</td>
</tr>
</tbody>
</table>

**Figure B.13** Individual Cost Components for write-first
Figure B.14 State Propagation for write-first

Table B.7 Raw Data for no-read-upon-write

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Edges</th>
<th>PA States</th>
<th>VAs States</th>
<th>CPA States</th>
<th>LP</th>
<th>QRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>65</td>
<td>42</td>
<td>354</td>
<td>4</td>
<td>4,5</td>
<td>38</td>
<td>19.4</td>
<td>1.4</td>
</tr>
<tr>
<td>9</td>
<td>105</td>
<td>68</td>
<td>1030</td>
<td>4</td>
<td>4,5</td>
<td>38</td>
<td>21.5</td>
<td>1.5</td>
</tr>
<tr>
<td>17</td>
<td>185</td>
<td>120</td>
<td>3390</td>
<td>4</td>
<td>4,5</td>
<td>38</td>
<td>27.2</td>
<td>1.2</td>
</tr>
<tr>
<td>33</td>
<td>345</td>
<td>224</td>
<td>12124</td>
<td>4</td>
<td>4,5</td>
<td>38</td>
<td>45.0</td>
<td>1.3</td>
</tr>
<tr>
<td>65</td>
<td>665</td>
<td>432</td>
<td>31694</td>
<td>4</td>
<td>4,5</td>
<td>38</td>
<td>88.0</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alpha</th>
<th>VA</th>
<th>CPA</th>
<th>MIP</th>
<th>PROPS</th>
<th>Time</th>
<th>Nodes</th>
<th>Confluence</th>
<th>δ Evals</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>0.6, 0.4</td>
<td>7.0</td>
<td>2.7</td>
<td>6.3</td>
<td>376</td>
<td>4779</td>
<td>con</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>0.6, 0.3</td>
<td>6.9</td>
<td>5.2</td>
<td>12.4</td>
<td>692</td>
<td>14427</td>
<td>con</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>0.4, 0.4</td>
<td>6.9</td>
<td>14.8</td>
<td>44.4</td>
<td>1324</td>
<td>48555</td>
<td>745</td>
<td>con</td>
<td></td>
</tr>
<tr>
<td>11.6</td>
<td>0.4, 0.3</td>
<td>6.9</td>
<td>74.8</td>
<td>328.6</td>
<td>2588</td>
<td>176139</td>
<td>1425</td>
<td>con</td>
<td></td>
</tr>
<tr>
<td>34.4</td>
<td>0.3, 0.3</td>
<td>6.7</td>
<td>357.0</td>
<td>2564.8</td>
<td>4924</td>
<td>434859</td>
<td>2785</td>
<td>con</td>
<td></td>
</tr>
</tbody>
</table>
Figure B.15 Individual Cost Components for no-read-upon-write

Figure B.16 State Propagation for no-read-upon-write
Table B.8 Raw Data for Single neighbors-think

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Alpha TFG Edges</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>65</td>
<td>35</td>
<td>139</td>
<td>3</td>
<td>15.1</td>
<td>0.6</td>
<td>2.6</td>
</tr>
<tr>
<td>12</td>
<td>122</td>
<td>68</td>
<td>322</td>
<td>3</td>
<td>19.4</td>
<td>0.7</td>
<td>3.7</td>
</tr>
<tr>
<td>24</td>
<td>242</td>
<td>134</td>
<td>688</td>
<td>3</td>
<td>31.2</td>
<td>0.8</td>
<td>6.1</td>
</tr>
<tr>
<td>48</td>
<td>482</td>
<td>266</td>
<td>1420</td>
<td>3</td>
<td>61.1</td>
<td>0.8</td>
<td>19.7</td>
</tr>
<tr>
<td>96</td>
<td>962</td>
<td>530</td>
<td>2884</td>
<td>3</td>
<td>151.5</td>
<td>1.0</td>
<td>68.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>1.7</td>
<td>2.2</td>
<td>78</td>
</tr>
<tr>
<td>2.8</td>
<td>3.7</td>
<td>174</td>
</tr>
<tr>
<td>6.2</td>
<td>7.4</td>
<td>366</td>
</tr>
<tr>
<td>19.7</td>
<td>22.6</td>
<td>750</td>
</tr>
<tr>
<td>86.3</td>
<td>122.0</td>
<td>1518</td>
</tr>
</tbody>
</table>

B.3 The Dining Philosophers Problem

Single Neighbors Think

The raw data for checking the single neighbors-think property on 5 different sizes of the dining philosophers program are given in Table B.8. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.17 and B.18. For state propagation analysis we include both run-time and the platform independent measures of work.

Multiple Neighbors Think

The raw data for checking the multiple neighbors-think property on 5 different sizes of the dining philosophers program are given in Table B.9. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.19 and B.20. For state propagation analysis we include both run-time and the platform independent measures of work.
Figure B.17 Individual Cost Components for Single neighbors-think

Figure B.18 State Propagation for Single neighbors-think
Table B.9 Raw Data for Multiple neighbors-think

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>Alpha TFG Edges</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>65</td>
<td>35</td>
<td>139</td>
<td>3</td>
<td>15.1</td>
<td>0.6</td>
<td>7.8</td>
</tr>
<tr>
<td>12</td>
<td>122</td>
<td>68</td>
<td>322</td>
<td>3</td>
<td>19.4</td>
<td>0.7</td>
<td>22.2</td>
</tr>
<tr>
<td>24</td>
<td>242</td>
<td>134</td>
<td>688</td>
<td>3</td>
<td>31.2</td>
<td>0.8</td>
<td>73.2</td>
</tr>
<tr>
<td>48</td>
<td>482</td>
<td>266</td>
<td>1420</td>
<td>3</td>
<td>61.1</td>
<td>0.8</td>
<td>352.8</td>
</tr>
<tr>
<td>96</td>
<td>962</td>
<td>530</td>
<td>2884</td>
<td>3</td>
<td>151.5</td>
<td>1.0</td>
<td>3268.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>5.1</td>
<td>6.6</td>
<td>234</td>
</tr>
<tr>
<td>16.8</td>
<td>22.2</td>
<td>1044</td>
</tr>
<tr>
<td>74.4</td>
<td>73.2</td>
<td>4392</td>
</tr>
<tr>
<td>472.8</td>
<td>352.8</td>
<td>18000</td>
</tr>
<tr>
<td>4142.4</td>
<td>3268.8</td>
<td>72864</td>
</tr>
</tbody>
</table>

Figure B.19 Individual Cost Components for Multiple neighbors-think
Figure B.20 State Propagation for Multiple neighbors-think
B.4 The Gas Station Problem

We scaled up two different FLAVERS/Ada analyses for the one-per-pump property.

Alpha Refined

The raw data for checking the one-per-pump property using only alphabet refinement on 5 different sizes of the gas station program are given in Table B.10. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.21 and B.22. For state propagation analysis we include both run-time and the platform independent measures of work.

Alpha and CI Refined

The raw data for checking the one-per-pump property using both alphabet and CI refinement on 4 different sizes of the gas station program are given in Table B.11. We plot the rates of growth for the measures of cost of each of the phases of analysis in Figures B.23 and B.24. For state propagation analysis we include both run-time and the platform independent measures of work.
Table B.10 Raw Data for Alpha Refined one-per-pump

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha TFG Nodes</th>
<th>PA Edges</th>
<th>States</th>
<th>LP</th>
<th>QRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>101</td>
<td>48</td>
<td>182</td>
<td>3</td>
<td>24.2</td>
<td>0.7</td>
</tr>
<tr>
<td>8</td>
<td>173</td>
<td>78</td>
<td>521</td>
<td>3</td>
<td>27.8</td>
<td>0.7</td>
</tr>
<tr>
<td>14</td>
<td>317</td>
<td>138</td>
<td>1739</td>
<td>3</td>
<td>46.6</td>
<td>0.7</td>
</tr>
<tr>
<td>26</td>
<td>605</td>
<td>258</td>
<td>6334</td>
<td>3</td>
<td>96.4</td>
<td>0.9</td>
</tr>
<tr>
<td>50</td>
<td>1181</td>
<td>498</td>
<td>24166</td>
<td>3</td>
<td>230.7</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alpha</th>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
<td>δ Evals</td>
</tr>
<tr>
<td>3.8</td>
<td>2.4</td>
<td>3.1</td>
<td>101</td>
</tr>
<tr>
<td>6.6</td>
<td>5.2</td>
<td>5.7</td>
<td>167</td>
</tr>
<tr>
<td>18.6</td>
<td>17.7</td>
<td>21.0</td>
<td>299</td>
</tr>
<tr>
<td>252.6</td>
<td>103.4</td>
<td>234.1</td>
<td>687</td>
</tr>
<tr>
<td>288.1</td>
<td>298.3</td>
<td>1102.7</td>
<td>1335</td>
</tr>
</tbody>
</table>

Figure B.21 Individual Cost Components for Alpha Refined one-per-pump
**Figure B.22** State Propagation for Alpha Refined *one-per-pump*

**Table B.11** Raw Data for Alpha+CI Refined *one-per-pump*

<table>
<thead>
<tr>
<th>Tasks</th>
<th>TFG Nodes</th>
<th>Alpha+CI TFG Nodes</th>
<th>PA States</th>
<th>LP</th>
<th>QRE</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>101</td>
<td>69</td>
<td>251</td>
<td>3</td>
<td>24.4</td>
<td>0.7</td>
</tr>
<tr>
<td>8</td>
<td>173</td>
<td>117</td>
<td>764</td>
<td>3</td>
<td>27.8</td>
<td>0.7</td>
</tr>
<tr>
<td>14</td>
<td>317</td>
<td>213</td>
<td>2654</td>
<td>3</td>
<td>46.6</td>
<td>0.7</td>
</tr>
<tr>
<td>26</td>
<td>605</td>
<td>405</td>
<td>9889</td>
<td>3</td>
<td>96.4</td>
<td>0.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CI</th>
<th>MIP</th>
<th>PROPS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Nodes</td>
<td>Confluence</td>
</tr>
<tr>
<td>5.3</td>
<td>3.1</td>
<td>3.6</td>
<td>169</td>
</tr>
<tr>
<td>9.6</td>
<td>6.7</td>
<td>6.5</td>
<td>292</td>
</tr>
<tr>
<td>31.0</td>
<td>23.1</td>
<td>19.4</td>
<td>538</td>
</tr>
<tr>
<td>169.1</td>
<td>118.5</td>
<td>235.5</td>
<td>1319</td>
</tr>
</tbody>
</table>
**Figure B.23** Individual Cost Components for Alpha+CI Refined one-per-pump

**Figure B.24** State Propagation for Alpha+CI Refined one-per-pump
APPENDIX C

SOURCE CODE FOR EMPIRICAL EVALUATIONS

This appendix provides the Ada source code for the programs that whose analyses are presented in Chapter 5.
C.1 Simple Protocol Program

procedure protocol is

done : BOOLEAN;

task LockManager is
  entry Acquire;
  entry Release;
end LockManager;

task Channel is
  entry Header(h : in INTEGER);
  entry Packet(p : in INTEGER);
end Channel;

task Client1;
task Client2;

task body LockManager is
  begin
    loop
      exit when done;
      accept Acquire;
      accept Release;
    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;

      protocol.LockManager.Acquire;
      null; -- $QR Ea[ht]$ 
      protocol.Channel.Header(h);
      null; -- $QR Ea[pt]$ 
      protocol.Channel.Packet(p);
      protocol.LockManager.Release;
    end loop;
  end Client1;
task body Client2 is
  h, p : INTEGER := 0;
begin
  loop
    exit when done;
      protocol.LockManager.Acquire;
      null; -- 
      protocol.Channel.Header(h);
      null; -- 
      protocol.Channel.Packet(p);
   protocol.LockManager.Release;
  end loop;
end Client2;

begin
  done := FALSE;
  delay 17.0;
  done := TRUE;
end protocol;
C.2 Readers/Writers Problem

procedure rw is

  task Reader1;
  task Reader2;

  task Writer1;
  task Writer2;

  task Control is
    entry Start_Read;
    entry Stop_Read;
    entry Start_Write;
    entry Stop_Write;
  end Control;

  task body Reader1 is
    done : Boolean;
    begin
      loop
        rw.Control.Start_Read;
        null; -- QREa/read1
        rw.Control.Stop_Read;
        exit when done;
      end loop;
    end Reader1;

  task body Reader2 is
    done : Boolean;
    begin
      loop
        rw.Control.Start_Read;
        null; -- QREa/read2
        rw.Control.Stop_Read;
        exit when done;
      end loop;
    end Reader2;

  task body Writer1 is
    done : Boolean;
    begin
      loop
        rw.Control.Start_Write;
        null; -- QREa/write1
        rw.Control.Stop_Write;
        exit when done;
      end loop;
    end Writer1;

  task body Writer2 is
    done : Boolean;
    begin
      loop
task body Control is
  ActiveReaders : Natural; -- Allow simultaneous readers
  WriterPresent : BOOLEAN; -- Allow a single writer
  Done : Boolean;
begin
  ActiveReaders := 0; -- QRE[ar=0]
  WriterPresent := FALSE; -- QRE[wp=f]
accept Start_Write;
accept Stop_Write;
loop
  select when not WriterPresent =>
  accept Start_Read; -- QREb[wpis=f]
    ActiveReaders := ActiveReaders+1; -- QREa[arinc]
    null; -- Workaround for comment related bug
  or accept Stop_Read;
    ActiveReaders := ActiveReaders-1; -- QREa[ardec]
    null; -- Workaround for comment related bug
  or when ActiveReaders = 0 and not WriterPresent =>
  accept Start_Write; -- QREb[aris=0,wpis=f]
    WriterPresent := TRUE; -- QREa[wp=t]
    null; -- Workaround for comment related bug
  or accept Stop_Write;
    WriterPresent := FALSE; -- QREa[wp=f]
    null; -- Workaround for comment related bug
end select;

-- Only allow exit when there are no writers or readers active
if (not WriterPresent)
  and
  (ActiveReaders = 0)
  then
    null; -- QREb[wpis=f,aris=0]
  if Done then
    exit;
  end if;
end if;
end loop; -- until done

end Control;

begin
  null;
end rw;
C.3 Dining Philosophers

procedure phils is

Finished : Boolean := FALSE;

task F0 is
  entry U0;
  entry D0;
end F0;

task F1 is
  entry U1;
  entry D1;
end F1;

task F2 is
  entry U2;
  entry D2;
end F2;

task P0;
task P1;
task P2;

task body F0 is
begin
  loop
    accept U0;
    accept D0;
    exit when Finished;
  end loop;
end F0;

task body F1 is
begin
  loop
    accept U1;
    accept D1;
    exit when Finished;
  end loop;
end F1;

task body F2 is
begin
  loop
    accept U2;
    accept D2;
    exit when Finished;
  end loop;
end F2;
task body P0 is
begin
  loop
    null; -- QREa[think0]
    phils.F0.U0;
    phils.F1.U1;
    null; -- QREa[eat0,rest0]
    phils.F0.D0;
    phils.F1.D1;
    exit when Finished;
  end loop;
end P0;

task body P1 is
begin
  loop
    null; -- QREa[think1]
    phils.F2.U2;
    phils.F1.U1;
    null; -- QREa[eat1,rest1]
    phils.F2.D2;
    phils.F1.D1;
    exit when Finished;
  end loop;
end P1;

task body P2 is
begin
  loop
    null; -- QREa[think2]
    phils.F0.U0;
    phils.F2.U2;
    null; -- QREa[eat2,rest2]
    phils.F0.D0;
    phils.F2.D2;
    exit when Finished;
  end loop;
end P2;

begin
  null;
end phils;
C.4 Gas Station Problem

procedure gas is

    task Operator is
        entry PrePay_1;
        entry PrePay_2;
        entry PrePay_3;
        entry Charge;
    end Operator;

    task Pump is
        entry Activate;
        entry Start_Pumping;
        entry Stop_Pumping;
        entry Turn_Off;
    end Pump;

    task Customer_1 is
        entry Change;
    end Customer_1;

    task Customer_2 is
        entry Change;
    end Customer_2;

    task Customer_3 is
        entry Change;
    end Customer_3;

    task body Operator is
        ActiveCustomers : Natural := 1;
        CustomerQueue   : Array (1..3) of Natural;
        Done             : Boolean;
    begin
        null; -- QRE_a/ac=1

        loop
            select
                accept PrePay_1 do
                    ActiveCustomers := ActiveCustomers+1; -- QRE_a/acinc
                    CustomerQueue(ActiveCustomers) := 1;
                    if ActiveCustomers = 1 then
                        null; -- QRE_b/acis=1
                    end if;
                end PrePay_1;

                or accept PrePay_2 do
                    ActiveCustomers := ActiveCustomers+1; -- QRE_a/acinc
                    CustomerQueue(ActiveCustomers) := 2;
                    if ActiveCustomers = 1 then
                        null; -- QRE_b/acis=1
                end PrePay_2;

    end Operator;
gas.Pump.Activate;
end if;
end PrePay_2;

or
accept PrePay_3 do
ActiveCustomers := ActiveCustomers + 1; -- QREb/acinc
CustomerQueue(ActiveCustomers) := 3;
if ActiveCustomers = 1 then
null; -- QREb/acis=1
end if;
end PrePay_3;

or
accept Charge;

if ActiveCustomers > 1 then
null; -- QREb/acis>1
end if;

if CustomerQueue(1) = 1 then
gas.Customer_1.Change;
elsif CustomerQueue(1) = 2 then
gas.Customer_2.Change;
else
end if;

ActiveCustomers := ActiveCustomers - 1; -- QREa/acdec

if ActiveCustomers > 0 then
null; -- QREb/acis>0

for i in 1..ActiveCustomers loop
CustomerQueue(i) := CustomerQueue(i + 1);
end loop;

end select;

if Done and ActiveCustomers = 0 then
null; -- QREb/acis=0
end if;
end loop;
end Operator;

task body Pump is
begin
loop

272
accept Activate;
accept Start_Pumping;
accept Stop_Pumping;
gas.Operator.Charge;

select
  accept Turn_Off;
  exit;
else
  null;
end select;
end loop;
end Pump;

task body Customer_1 is
  Done : Boolean;
begin
  loop
    gas.Operator.Prepay_1;

    gas.Pump.Start_Pumping;  -- QREa[start1]
    gas.Pump.Stop_Pumping;  -- QREa[stop1]

    accept Change;

    if Done then
      exit;
    end if;
  end loop;
end Customer_1;

task body Customer_2 is
  Done : Boolean;
begin
  loop
    gas.Operator.Prepay_2;

    gas.Pump.Start_Pumping;  -- QREa[start2]
    gas.Pump.Stop_Pumping;  -- QREa[stop2]

    accept Change;

    if Done then
      exit;
    end if;
  end loop;
end Customer_2;

task body Customer_3 is
  Done : Boolean;
begin
  loop

gas.Operator.PrePay_3;

gas.Pump.Start_Pumping; \text{ -- } QR_{Ea[start3]}
gas.Pump.Stop_Pumping; \text{ -- } QR_{Ea[stop3]}

\textbf{accept} Change;

\textbf{if} Done \textbf{then}
\qquad \textbf{exit};
\qquad \textbf{end if};
\textbf{end loop};
\textbf{end Customer_3};

begin
\qquad \text{null};
\text{end gas;}
C.5 DARTES

procedure DARTES is

COMMAND : Boolean;

task MAIN;

task SERVER x NAV is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER x NAV;

task DISPLAY x GRAPHIC is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end DISPLAY x GRAPHIC;

task SERVER x RWR is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER x RWR;

task TRACKING x TARGET_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end TRACKING x TARGET_UPDATE;

task SERVER x RADAR is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER x RADAR;

task NAV x UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end NAV_x.UPDATE;

task NAV_x.STEERING_CMDS is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end NAV_x.STEERING_CMDS;

task SERVER_x.TRACKING is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER_x.TRACKING;

task NAV_x.STATUS is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end NAV_x.STATUS;

task SERVER_x.WEAPON is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER_x.WEAPON;

task RADAR_x.TARGET_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end RADAR_x.TARGET_UPDATE;

task RADAR_x.TRACKING_FILTER is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end RADAR_x.TRACKING_FILTER;

task RWR_x.CONTACT_MGMT is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end RWR_x.CONTACT_MGMT;
task SERVER\_x\_FLIR is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE\_DATA;
  entry RETRIEVE\_DATA;
  entry END\_OF\_SEQUENCE;
end SERVER\_x\_FLIR;

task FLIR\_x\_POINTING\_DATA is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END\_OF\_SEQUENCE;
end FLIR\_x\_POINTING\_DATA;

task DISPLAY\_x\_STORES\_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END\_OF\_SEQUENCE;
end DISPLAY\_x\_STORES\_UPDATE;

task SERVER\_x\_DISPLAY is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE\_DATA;
  entry RETRIEVE\_DATA;
  entry END\_OF\_SEQUENCE;
end SERVER\_x\_DISPLAY;

task BIT\_x\_STATUS\_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END\_OF\_SEQUENCE;
end BIT\_x\_STATUS\_UPDATE;

task SERVER\_x\_BIT is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE\_DATA;
  entry RETRIEVE\_DATA;
  entry END\_OF\_SEQUENCE;
end SERVER\_x\_BIT;

task DISPLAY\_x\_STATUS\_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END\_OF\_SEQUENCE;
end DISPLAY_x.STATUS_UPDATE;

task DISPLAY_x.KEYSET is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end DISPLAY_x.KEYSET;

task SERVER_x.TIME is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry STORE_DATA;
  entry RETRIEVE_DATA;
  entry END_OF_SEQUENCE;
end SERVER_x.TIME;

task TIME_x.UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end TIME_x.UPDATE;

task DISPLAY_x.HOOK_UPDATE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end DISPLAY_x.HOOK_UPDATE;

task DATA_BUS_x.POLLDEVICE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
end DATA_BUS_x.POLLDEVICE;

task SERVER_x.DATABUS is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry SEND_DATA;
  entry GET_DATA;
  entry RETRIEVE_DATA;
  entry STORE_DATA;
  entry END_OF_SEQUENCE;
end SERVER_x.DATABUS;

task WEAPON_x.BOMB_PROTOCOL is
  entry INITIALIZE;
  entry STOP;
  entry GO;
entry END_OF_SEQUENCE;
entry COMMAND;
end WEAPON.x_BOMB_PROTOCOL;

task WEAPON.x_BOMB_AIMING is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
  entry INITIATE_AIM;
end WEAPON.x_BOMB_AIMING;

task WEAPON.x_BOMB_RELEASE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
  entry END_OF_SEQUENCE;
  entry DROP;
end WEAPON.x_BOMB_RELEASE;

task BACKGROUND.x_IDLE is
  entry INITIALIZE;
  entry STOP;
  entry GO;
end BACKGROUND.x_IDLE;

task MODEL_PLATFORM.x_MODEL_TASK_SET is
  entry QUIT;
  entry START;
  entry STOP;
end MODEL_PLATFORM.x_MODEL_TASK_SET;

begin
  BACKGROUND.x_IDLE is
  begin
    loop
      select
        accept INITIALIZE do
          null;
        end INITIALIZE;
      or
        accept STOP do
          exit;
        end select;

        accept GO do
          DARTES.MODEL_PLATFORM.x_MODEL_TASK_SET.STOP;
        end GO;

      end loop;
end BACKGROUND.x_IDLE;

task body BIT.x_E_STATUS_UPDATE is
  working : BOOLEAN := TRUE;
begin
loop
  select
    accept INITIALIZE do
      null;
    end INITIALIZE;
  or
    accept STOP;
    exit;
  end select;

accept GO;

while working loop
  DARTES.SERVER.BIT.RETRIEVE_DATA;
  DARTES.SERVER.BIT.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end BIT.E.STATUS_UPDATE;

---

task body DATA_BUS.POLL_DEVICE is
  working : BOOLEAN := TRUE;
begin
loop
  select
    accept INITIALIZE do
      null;
    end INITIALIZE;
  or
    accept STOP;
    exit;
  end select;

accept GO;

while working loop
  DARTES.SERVER.DATA_BUS.RETRIEVE_DATA;
  DARTES.SERVER.DATA_BUS.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end DATA_BUS.POLL_DEVICE;

---

task body DISPLAY.STATUS_UPDATE is
  working : BOOLEAN := TRUE;
begin
loop
  select
    accept INITIALIZE do
null;
end INITIALIZE;
or
accept STOP;
exi;
end select;

accept GO;

while working loop
    DARTES.SERVER."DISPLAY.RETRIEVE_DATA;
    DARTES.SERVER."NAV.RETRIEVE_DATA;
    DARTES.SERVER."RWR.RETRIEVE_DATA;
    DARTES.SERVER."RADAR.RETRIEVE_DATA;
    DARTES.SERVER."TRACKING.RETRIEVE_DATA;
    DARTES.SERVER."BIT.RETRIEVE_DATA;
    DARTES.SERVER."WEAPON.RETRIEVE_DATA;
    DARTES.SERVER."DATA_BUS.GET_DATA;
    DARTES.SERVER."DATA_BUS.SEND_DATA;
    DARTES.SERVER."DISPLAY.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end DISPLAY."STATUS_UPDATE;

task body DISPLAY."KEYSET is
    working : BOOLEAN := TRUE;
    KEY : Natural range 0..7 := 0;
begin
    loop
        select
            accept INITIALIZE do
                null;
                end INITIALIZE;
            or
                accept STOP;
                exit;
                end select;
            accept GO;
        while working loop
            DARTES.SERVER."DISPLAY.RETRIEVE_DATA;
            DARTES.SERVER."DATA_BUS.GET_DATA;

            KEY := 0;
            if KEY = 0 then
                DARTES.SERVER."NAV.STORE_DATA;
            elsif KEY = 1 then
                DARTES.SERVER."DISPLAY.STORE_DATA;
            elsif KEY = 2 then
                DARTES.SERVER."TIME.STORE_DATA;
            end if;
        end loop;
    end loop;
end DISPLAY."KEYSET;
elsif KEY = 3 then
    DARTES.SERVER_x_RWR.STORE_DATA;
elsif KEY = 4 then
    DARTES.SERVER_x_BIT.STORE_DATA;
elsif KEY = 5 then
    DARTES.SERVER_x_TRACKING.STORE_DATA;
elsif KEY = 6 then
    DARTES.WEAPON_x_BOMB_PROTOCOL.COMMAND;
else
    null;
end if;
end loop;

accept END_OF_SEQUENCE;
end loop;
end DISPLAY_x_KEYSET;

task body DISPLAY_x_HOOK_UPDATE is
    working : BOOLEAN := TRUE;
begin
    loop
        select
            accept INITIALIZE do
                null;
            end INITIALIZE;
        or
            accept STOP;
            exit;
        end select;

        accept GO;

        while working loop
            DARTES.SERVER_x_DISPLAY.RETRIEVE_DATA;
            DARTES.SERVER_x_DATA_BUS.GET_DATA;
            DARTES.SERVER_x_DATA_BUS.SEND_DATA;
            DARTES.SERVER_x_DISPLAY.STORE_DATA;
        end loop;

        accept END_OF_SEQUENCE;
    end loop;
end DISPLAY_x_HOOK_UPDATE;

task body DISPLAY_x_GRAPHIC is
    working : BOOLEAN := TRUE;
begin
    loop
        select
            accept INITIALIZE do
                null;
            end INITIALIZE;
        or
accept STOP;
exit;
end select;

accept GO;

while working loop
  DARTES.SERVER.x.DISPLAY.RETRIEVE_DATA;
  DARTES.SERVER.x.NAV.RETRIEVE_DATA;
  DARTES.SERVER.x.RWR.RETRIEVE_DATA;
  DARTES.SERVER.x.RADAR.RETRIEVE_DATA;
  DARTES.SERVER.x.TRACKING.RETRIEVE_DATA;
  DARTES.SERVER.x.BIT.RETRIEVE_DATA;
  DARTES.SERVER.x.WEAPON.RETRIEVE_DATA;
  DARTES.SERVER.x.DATA_BUS.GET_DATA;
  DARTES.SERVER.x.DATA_BUS.SEND_DATA;
  DARTES.SERVER.x.DISPLAY.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end DISPLAY.x.GRAPHIC;

task body DISPLAY.x.STORES.UPDATE is
  working : BOOLEAN := TRUE;
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

    accept GO;

    while working loop
      DARTES.SERVER.x.DISPLAY.RETRIEVE_DATA;
      DARTES.SERVER.x.WEAPON.RETRIEVE_DATA;
      DARTES.SERVER.x.DATA_BUS.SEND_DATA;
      DARTES.SERVER.x.DISPLAY.STORE_DATA;
    end loop;

    accept END_OFSEQUENCE;
    end loop;
  end DISPLAY.x.STORES.UPDATE;

  task body FLIR.x.POINTING_DATA is
    working : BOOLEAN := TRUE;
begin
loop
select
  accept INITIALIZE do
  null;
  end INITIALIZE;
or
  accept STOP;
  exit;
  end select;
accept GO;

while working loop
  DARTES.SERVER_x_FLIR.RETRIEVE_DATA;
  DARTES.SERVER_x_NAV.RETRIEVE_DATA;
  DARTES.SERVER_x_DATA_BUS.GET_DATA;
  DARTES.SERVER_x_DATA_BUS.SEND_DATA;
  DARTES.SERVER_x_FLIR.STORE_DATA;
end loop;
accept END_OF_SEQUENCE;
end loop;
end FLIR_x_POINTING_DATA;

task body MODEL_PLATFORM_xMODEL_TASK_SET is
  begin
    loop
      select
        accept QUIT;
        exit;
or
        accept START do
          DARTES.NAV_x_UPDATE.INITIALIZE;
          DARTES.NAV_x_STEERING_CMDS.INITIALIZE;
          DARTES.NAV_x_STATUS.INITIALIZE;
          DARTES.DISPLAY_x_STATUS_UPDATE.INITIALIZE;
          DARTES.DISPLAY_x_KEYSET.INITIALIZE;
          DARTES.DISPLAY_x_HOOK_UPDATE.INITIALIZE;
          DARTES.DISPLAY_x_GRAPHIC.INITIALIZE;
          DARTES.DISPLAY_x_STORES_UPDATE.INITIALIZE;
          DARTES.RWR_x_CONTACT_MGMT.INITIALIZE;
          DARTES.RADAR_x_TARGET_UPDATE.INITIALIZE;
          DARTES.RADAR_x_TRACKING_FILTER.INITIALIZE;
          DARTES.TRACKING_x_TARGET_UPDATE.INITIALIZE;
          DARTES.BIT_x_E_STATUS_UPDATE.INITIALIZE;
          DARTES.DATA_BUS_x_POLL_DEVICE.INITIALIZE;
          DARTES.WEAPON_x_BOMB_PROTOCOL.INITIALIZE;
          DARTES.WEAPON_x_BOMB_AIMING.INITIALIZE;
          DARTES.WEAPON_x_BOMB_RELEASE.INITIALIZE;
          DARTES.BACKGROUND_x_IDLE.INITIALIZE;
          DARTES.SERVER_x_NAV.INITIALIZE;
          DARTES.SERVER_x_DISPLAY.INITIALIZE;
          DARTES.SERVER_x_RWR.INITIALIZE;
          DARTES.SERVER_x_RADAR.INITIALIZE;

  284
DARTELES.SERVER.xTRACKING.INITIALIZE;
DARTELES.SERVER.xBIT.INITIALIZE;
DARTELES.SERVER.xDATA_BUS.INITIALIZE;
DARTELES.SERVER.xWEAPON.INITIALIZE;
DARTELES.TIME.xUPDATE.INITIALIZE;
DARTELES.SERVER.xFLIR.INITIALIZE;
DARTELES.SERVER.xTIME.INITIALIZE;
DARTELES.FLIR.xPOINTING_DATA.INITIALIZE;
DARTELES.SERVER.xNAV.GO;
DARTELES.SERVER.xDISPLAY.GO;
DARTELES.SERVER.xRWR.GO;
DARTELES.SERVER.xRADAR.GO;
DARTELES.SERVER.xTRACKING.GO;
DARTELES.SERVER.xBIT.GO;
DARTELES.SERVER.xDATA_BUS.GO;
DARTELES.SERVER.xWEAPON.GO;
DARTELES.BACKGROUND.xIDLE.GO;
DARTELES.NAV.xSTATUS.GO;
DARTELES.BIT.xE_STATUS_UPDATE.GO;
DARTELES.DISPLAY.xSTATUS_UPDATE.GO;
DARTELES.DISPLAY.xSTORES_UPDATE.GO;
DARTELES.RADAR.xTARGET_UPDATE.GO;
DARTELES.NAV.xSTEERING_CMDS.GO;
DARTELES.DISPLAY.xKEYSET.GO;
DARTELES.DISPLAY.xGRAPHIC.GO;
DARTELES.TRACKING.xTARGET_UPDATE.GO;
DARTELES.DISPLAY.xHOOK_UPDATE.GO;
DARTELES.NAV.xUPDATE.GO;
DARTELES.RWR.xCONTACT_MGMT.GO;
DARTELES.DATA_BUS.xPOLL_DEVICE.GO;
DARTELES.RADAR.xTRACKING_FILTER.GO;
DARTELES.WEAPON.xBOMB_PROTOCOL.GO;
DARTELES.WEAPON.xBOMB_AIMING.GO;
DARTELES.WEAPON.xBOMB_RELEASE.GO;
DARTELES.TIME.xUPDATE.GO;
DARTELES.SERVER.xFLIR.GO;
DARTELES.SERVER.xTIME.GO;
DARTELES.FLIR.xPOINTING_DATA.GO;

accept STOP;

DARTELES.WEAPON.xBOMB_RELEASE.END_OF_SEQUENCE;
DARTELES.RADAR.xTRACKING_FILTER.END_OF_SEQUENCE;
DARTELES.NAV.xUPDATE.END_OF_SEQUENCE;
DARTELES.NAV.xSTEERING_CMDS.END_OF_SEQUENCE;
DARTELES.NAV.xSTATUS.END_OF_SEQUENCE;
DARTELES.DISPLAY.xSTATUS_UPDATE.END_OF_SEQUENCE;
DARTELES.DISPLAY.xKEYSET.END_OF_SEQUENCE;
DARTELES.DISPLAY.xHOOK_UPDATE.END_OF_SEQUENCE;
DARTELES.DISPLAY.xGRAPHIC.END_OF_SEQUENCE;
DARTELES.DISPLAY.xSTORES_UPDATE.END_OF_SEQUENCE;
DARTELES.RWR.xCONTACT_MGMT.END_OF_SEQUENCE;
DARTELES.RADAR.xTARGET_UPDATE.END_OF_SEQUENCE;
DARTELES.TRACKING.xTARGET_UPDATE.END_OF_SEQUENCE;

285
DARTES.BITx_E.STATUS_UPDATE.END_OF_SEQUENCE;
DARTES.DATA_BUSx_POLL_DEVICE.END_OF_SEQUENCE;
DARTES.WEAPONx_BOMB_PROTOCOL.END_OF_SEQUENCE;
DARTES.WEAPONx_BOMB_AIMING.END_OF_SEQUENCE;
DARTES.SERVERx_NAV.END_OF_SEQUENCE;
DARTES.SERVERx_DISPLAY.END_OF_SEQUENCE;
DARTES.SERVERx_RWR.END_OF_SEQUENCE;
DARTES.SERVERx_RADAR.END_OF_SEQUENCE;
DARTES.SERVERx_TRACKING.END_OF_SEQUENCE;
DARTES.SERVERx_BIT.END_OF_SEQUENCE;
DARTES.SERVERx_DATA_BUS.END_OF_SEQUENCE;
DARTES.SERVERx_WEAPON.END_OF_SEQUENCE;
DARTES.TIMEx_UPDATE.END_OF_SEQUENCE;
DARTES.SERVERx_LIR.END_OF_SEQUENCE;
DARTES.SERVERx_TIME.END_OF_SEQUENCE;
DARTES.FLIRx_POINTING_DATA.END_OF_SEQUENCE;

COMMAND = TRUE; -- QREa[command=t]
end START;

end select;
end loop;
end MODEL_PLATFORMx_MODEL_TASK_SET;

task body NAVx_UPDATE is
working : BOOLEAN := TRUE;
begins
loop
select
accept INITIALIZE do
null;
end INITIALIZE;
or
accept STOP;
exit;
end select;

accept GO;

while working loop
DARTES.SERVERx_NAV.RETRIEVE_DATA;
DARTES.SERVERx_DATA_BUS.GET_DATA;
DARTES.SERVERx_RADAR.RETRIEVE_DATA;
DARTES.SERVERx_NAV.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end NAVx_UPDATE;

task body NAVx_STEERING_CMDS is
working : BOOLEAN := TRUE;
begins
loop
  select
    accept INITIALIZE do
    null;
  end INITIALIZE;
  or
    accept STOP;
    exit;
  end select;

accept GO;

while working loop
  DARTES.SERVER.x_NAV.RETRIEVE_DATA;
  DARTES.SERVER.x_DISPLAY.RETRIEVE_DATA;
  DARTES.SERVER.x_NAV.STORE_DATA;
end loop;

  accept END_OF_SEQUENCE;
end loop;
end NAV.x_STEERING_CMDS;

task body NAV.x_STATUS is
  working : BOOLEAN := TRUE;
begin
  loop
    select
      accept INITIALIZE do
      null;
    end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

    accept GO;

    while working loop
      DARTES.SERVER.x_NAV.RETRIEVE_DATA;
      DARTES.SERVER.x_DATA_BUS.GET_DATA;
      DARTES.SERVER.x_NAV.STORE_DATA;
    end loop;

    accept END_OF_SEQUENCE;
  end loop;
end NAV.x_STATUS;

task body RADAR.x_TARGET_UPDATE is
  working : BOOLEAN := TRUE;
begin
  loop
    select

accept INITIALIZE do
  null;
end INITIALIZE;
or
  accept STOP;
  exit;
end select;

accept GO;

while working loop
  DARTES.SERVER.x_RADAR.RETRIEVE_DATA;
  DARTES.SERVER.x_NAV.RETRIEVE_DATA;
  DARTES.SERVER.x_DATA_BUS.GET_DATA;
  DARTES.SERVER.x_DATA_BUS.SEND_DATA;
  DARTES.SERVER.x_RADAR.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end RADAR.x_TARGET_UPDATE;

task body RADAR.x_TRACKING_FILTER is
  working : BOOLEAN := TRUE;
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

    accept GO;

    while working loop
      DARTES.SERVER.x_RADAR.RETRIEVE_DATA;
      DARTES.SERVER.x_NAV.RETRIEVE_DATA;
      DARTES.SERVER.x_DATA_BUS.GET_DATA;
      DARTES.SERVER.x_DATA_BUS.SEND_DATA;
      DARTES.SERVER.x_RADAR.STORE_DATA;
    end loop;

    accept END_OF_SEQUENCE;
  end loop;
end RADAR.x_TRACKING_FILTER;

task body RWR.x_CONTACT_MGMT is
  working : BOOLEAN := TRUE;
begin
loop
select
  accept INITIALIZE do
    null;
  end INITIALIZE;
or
  accept STOP;
  exit;
end select;
accept GO;

while working loop
  DARTES.SERVER.x RWR.RETRIEVE_DATA;
  DARTES.SERVER.x DATA_BUS.GET_DATA;
  DARTES.SERVER.x NAV.RETRIEVE_DATA;
  DARTES.SERVER.x WEAPON.RETRIEVE_DATA;
  DARTES.SERVER.x DATA_BUS SEND_DATA;
  DARTES.SERVER.x RWR.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end RWR.x CONTACT_MGMT;

task body SERVER.x NAV is
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

    accept GO;

    loop
      select
        accept STORE_DATA;
      or
        accept RETRIEVE_DATA;
      or
        accept END_OF_SEQUENCE;
      exit;
    end select;
  end loop;
end SERVER.x NAV;
task body SERVER\_x\_DISPLAY is
begin
loop
select
  accept INITIALIZE do
    null;
end INITIALIZE;
or
  accept STOP;
  exit;
end select;
accept GO;
loop
select
  accept STORE\_DATA;
or
  accept RETRIEVE\_DATA;
or
  accept END\_OF\_SEQUENCE;
  exit;
end select;
end loop;
end SERVER\_x\_DISPLAY;

task body SERVER\_x\_RWR is
begin
loop
select
  accept INITIALIZE do
    null;
end INITIALIZE;
or
  accept STOP;
  exit;
end select;
accept GO;
loop
select
  accept STORE\_DATA;
or
  accept RETRIEVE\_DATA;
or
  accept END\_OF\_SEQUENCE;
  exit;
end select;
end loop;
end loop;
end SERVER\_x\_RWR;
task body SERVER,x,RADAR is
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;
  accept GO;
  end loop;
end SERVER,x,RADAR;

task body SERVER,x,TRACKING is
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;
  accept GO;
  end loop;
end SERVER,x,TRACKING;
end loop;
end SERVER\textsubscript{X}.TRACKING;

\textbf{task body} SERVER\textsubscript{X}.TIME is
begin
loop
select
\hspace{1em} accept INITIALIZE do
\hspace{1em} null;
\hspace{1em} end INITIALIZE;
or
\hspace{1em} accept STOP;
\hspace{1em} exit;
end select;

accept GO;

loop
select
\hspace{1em} accept STORE\_DATA;
or
\hspace{1em} accept RETRIEVE\_DATA;
or
\hspace{1em} accept END\_OF\_SEQUENCE;
exit;
end select;
end loop;
end SERVER\textsubscript{X}.TIME;

\textbf{task body} SERVER\textsubscript{X}.FLIR is
begin
loop
select
\hspace{1em} accept INITIALIZE do
\hspace{1em} null;
\hspace{1em} end INITIALIZE;
or
\hspace{1em} accept STOP;
\hspace{1em} exit;
end select;

accept GO;

loop
select
\hspace{1em} accept STORE\_DATA;
or
\hspace{1em} accept RETRIEVE\_DATA;
or
\hspace{1em} accept END\_OF\_SEQUENCE;
exit;
end select;
end loop;
end loop;
end SERVER_.x_FLIR;

task body SERVER_.x_BIT is begin
loop
select
  accept INITIALIZE do
    null;
  end INITIALIZE;
or
  accept STOP;
  exit;
end select;

accept GO;
loop
select
  accept STORE_DATA;
or
  accept RETRIEVE_DATA;
or
  accept END_OF_SEQUENCE;
  exit;
end select;
end loop;
end loop;
end SERVER_.x_BIT;

task body SERVER_.x_DATA_BUS is begin
loop
select
  accept INITIALIZE do
    null;
  end INITIALIZE;
or
  accept STOP;
  exit;
end select;

accept GO;
loop
select
  accept SEND_DATA;
or
  accept GET_DATA;
or
accept STORE_DATA;
or
accept RETRIEVE_DATA;
or
accept END_OF_SEQUENCE;
exit;
end select;
end loop;
end loop;
end SERVER_X_DATA_BUS;

task body SERVER_X_WEAPON is
begin
loop
select
accept INITIALIZE do
null;
end INITIALIZE;
or
accept STOP;
exit;
end select;
accept GO;
loop
select
accept STORE_DATA;
or
accept RETRIEVE_DATA;
or
accept END_OF_SEQUENCE;
exit;
end select;
end loop;
end loop;
end SERVER_X_WEAPON;

task body TIME_X_UPDATE is
    working : BOOLEAN := TRUE;
begin
    loop
        select
            accept INITIALIZE do
                null;
            end INITIALIZE;
or
            accept STOP;
            exit;
        end select;
        accept GO;

294
while working loop
    DARTES.SERVER.x.TIME.RETRIEVE_DATA;
    DARTES.SERVER.x.TIME.STORE_DATA;
end loop;

accept END_OF_SEQUENCE;
end loop;
end TIME.x.UPDATE;

task body TRACKING.x_TARGET_UPDATE is
    working : BOOLEAN := TRUE;
begin
    loop
        select
            accept INITIALIZE do
                null;
            end INITIALIZE;
            or
            accept STOP;
                exit;
            end select;

        accept GO;

        while working loop
            DARTES.SERVER.x.TRACKING.RETRIEVE_DATA;
            DARTES.SERVER.x.DATA_BUS.GET_DATA;
            DARTES.SERVER.x.NAV.RETRIEVE_DATA;
            DARTES.SERVER.x.RADAR.RETRIEVE_DATA;
            DARTES.SERVER.x.RWR.RETRIEVE_DATA;
            DARTES.SERVER.x.RADAR.STORE_DATA;
        end loop;

        accept END_OF_SEQUENCE;
    end loop;
end TRACKING.x_TARGET_UPDATE;

task body WEAPON.x_BOMB_PROTOCOL is
begin
    loop
        select
            accept INITIALIZE do
                null;
            end INITIALIZE;
            or
            accept STOP;
                exit;
            end select;

        accept GO;
end


```plaintext
loop
  select
    accept COMMAND;
    DARTES.WEAPON_x.BOMB_AIMING.INITIATE_AIM;
  or
    accept END_OF_SEQUENCE;
    exit;
  end select;
end loop;

end loop;
end WEAPON_x.BOMB_PROTOCOL;

task body WEAPON_x.BOMB_AIMING is
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

    accept GO;

    loop
      select
        accept INITIATE_AIM;
        DARTES.WEAPON_x.BOMB_RELEASE.DROP;
      or
        accept END_OF_SEQUENCE;
        exit;
      end select;
    end loop;

  end loop;
end WEAPON_x.BOMB_AIMING;

task body WEAPON_x.BOMB_RELEASE is
begin
  loop
    select
      accept INITIALIZE do
        null;
      end INITIALIZE;
    or
      accept STOP;
      exit;
    end select;

end loop;
end WEAPON_x.BOMB_RELEASE;
```
accept GO;

loop
    select
        accept DROP;
    or
        accept END_OF_SEQUENCE;
        exit;
    end select;
end loop;
end loop;
end WEAPON\_BOMB\_RELEASE;

task body MAIN is
begin
    COMMAND := FALSE; -- QREa[command=f]
    loop
        if COMMAND then
            DARTES.NAV\_x\_UPDATE\_STOP; -- QREb[commandis=f]
            DARTES.NAV\_x\_STEERING\_CMDS\_STOP;
            DARTES.NAV\_x\_STATUS\_STOP;
            DARTES.DISPLAY\_x\_STATUS\_UPDATE\_STOP;
            DARTES.DISPLAY\_x\_KEYSET\_STOP;
            DARTES.DISPLAY\_x\_HOOK\_UPDATE\_STOP;
            DARTES.DISPLAY\_x\_GRAPHIC\_STOP;
            DARTES.DISPLAY\_x\_STORES\_UPDATE\_STOP;
            DARTES.RWR\_x\_CONTACT\_MGMT\_STOP;
            DARTES.RADAR\_x\_TARGET\_UPDATE\_STOP;
            DARTES.RADAR\_x\_TRACKING\_FILTER\_STOP;
            DARTES.TRACKING\_x\_TARGET\_UPDATE\_STOP;
            DARTES.TIME\_x\_UPDATE\_STOP;
            DARTES.FLIR\_x\_POINTING\_DATA\_STOP;
            DARTES.BIT\_x\_E\_STATUS\_UPDATE\_STOP;
            DARTES.DATA\_BUS\_x\_POLL\_DEVICE\_STOP;
            DARTES.WEAPON\_x\_BOMB\_PROTOCOL\_STOP;
            DARTES.WEAPON\_x\_BOMB\_AIMING\_STOP;
            DARTES.WEAPON\_x\_BOMB\_RELEASE\_STOP;
            DARTES.BACKGROUND\_x\_DLE\_STOP;
            DARTES.SERVER\_x\_NAV\_STOP;
            DARTES.SERVER\_x\_DISPLAY\_STOP;
            DARTES.SERVER\_x\_RWR\_STOP;
            DARTES.SERVER\_x\_RADAR\_STOP;
            DARTES.SERVER\_x\_TRACKING\_STOP;
            DARTES.SERVER\_x\_TIME\_STOP;
            DARTES.SERVER\_x\_FLIR\_STOP;
            DARTES.SERVER\_x\_BIT\_STOP;
            DARTES.SERVER\_x\_DATA\_BUS\_STOP;
            DARTES.SERVER\_x\_WEAPON\_STOP;
            DARTES.MODEL\_PLATFORM\_x\_MODEL\_TASK\_SET\_QUIT;
        exit;
    else
        DARTES.MODEL\_PLATFORM\_x\_MODEL\_TASK\_SET\_START; -- QREb[commandis=t]
        null;
    end if;
end loop;
null;
end if;
end loop;
end MAIN;

begin
null;
end DARTES;
Bibliography


[85] Young, M. and Taylor, R.N. Rethinking the taxonomy of fault detection techniques. SERC TR-62-P, Department of Computer Science, Purdue University, September 1991.
