Analysis of Synchronization in a Parallel Programming Environment

Jaspal S. Subhlok

CRPC-TR90068
August, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Analysis of Synchronization in a Parallel Programming Environment

Jaspal S. Subhlok

Abstract

Parallel programming is an intellectually demanding task. One of the most difficult challenges in the development of parallel programs for asynchronous shared memory systems is avoiding errors caused by inadvertent data sharing, often referred to as *data races*. Static prediction of data races requires data dependence analysis, as well as analysis of parallelism and synchronization. This thesis addresses *synchronization analysis* of parallel programs. Synchronization analysis enables accurate prediction of data races in a parallel program. The results of synchronization analysis can also be used in a variety of other ways to enhance the power and flexibility of a parallel programming environment.

We introduce the notion of *schedule-correctness* of a parallel program and relate it to data dependences and execution orders in the program. We develop a framework for reasoning about execution orders and prove that static determination of execution orders in parallel programs with synchronization is NP-hard, even for a simple language.

We present two different algorithms for synchronization analysis that determine whether the cumulative effect of synchronization is sufficient to ensure the execution ordering required by data dependence. The first algorithm iteratively computes and propagates ordering information between neighbors in the program flow graph, analogous to data flow analysis algorithms. The second algorithm computes the necessary path information in the program graph and uses it to transform the problem into an integer programming problem, which also is NP-hard. We present a heuristic approach to solving the integer programming problem obtained and argue that it is efficient for the problem cases that we expect to encounter in our analysis. We discuss the merits, shortcomings and suitability of the two algorithms presented.

RICE UNIVERSITY

Analysis of Synchronization in a Parallel Programming Environment

by

Jaspal S. Subhlok

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Kenneth Kennedy

Kenneth Kennedy, Chairman
Noah Harding Professor of Computer Science

Willy Zwaenepoel

Willy Zwaenepoel
Associate Professor of Computer Science

E. Andrew Boyd

Andrew Boyd
Assistant Professor of Mathematical Sciences

Houston, Texas

August, 1990

We have developed a prototype implementation of synchronization analysis. We discuss the implementation and practical issues relating to the effectiveness and usefulness of our analysis.

Acknowledgments

I first wish to thank my advisor Ken Kennedy for being the architect of an excellent environment for this research, and for being a source of advice and encouragement. This dissertation originated from discussions with David Callahan, who directed most of the initial research. Other members of my thesis committee, Willy Zwaenepoel and Andy Boyd, provided plenty of help and support. I had many technical discussions with Chau-Wen Tseng and John Mellor-Crummey which had a significant impact on my dissertation. Don Baker and Kathryn McKinley helped with proofreading the thesis draft.

Finally I wish to thank all those people who contributed in many different ways, but are too numerous to thank by name. Much inspiration for this research came from discussions with many members of the PFC and ParaScope research groups. My basketball and other sports teammates provided the perfect counter to academic research. Thanks to everybody who helped create a pleasant insanity around my existence in graduate school.

Contents

Chapter 1

Introduction and Overview

1.1 Introduction

In the last decade automatic vectorizing and parallelizing technology has become fairly sophisticated enabling transformations that lead to fast execution of originally sequential programs on high speed multiprocessors. Although there have been many successes, a fully automated approach to utilization of multiprocessors has inherent limitations. It is increasingly apparent that a significant amount of parallelism cannot be extracted from sequential programs by automatic methods because automatic techniques cannot grasp certain program intricacies that are critical for exploiting parallelism. Moreover, automatically generated parallel programs often lead to a high synchronization overhead during execution. Often the most suitable way to increase parallelism in an application is to redesign the algorithms being used or directly encode the parallelism.

Thus, it seems that to make the best possible use of multiprocessors, the user needs to be more involved in the development of parallel programs. Unfortunately parallel programs are harder to write and debug than sequential programs. The speedups achieved by parallel programming may simply not be enough to justify the additional effort that a programmer has to invest to achieve the desired results. To make parallel programming profitable, it is extremely important that the development time for a parallel program be comparable to that of an an equivalent sequential program. We believe it is possible to make parallel programming more profitable and less of a burden in a programming environment with powerful tools that aid in the development and debugging of parallel programs.

To help a programmer with parallel programming, we must understand how parallel programming is different from sequential programming. A programmer has to specify controlled parallelism when writing a parallel program. Parallel programming languages provide constructs for specifying parallel execution of code segments. The language or a runtime library provides mechanisms for *synchronization*, that can be

used to enforce execution ordering within the code segments that are specified to execute in parallel. Language constructs are used to specify parallelism, and synchronization mechanisms are used to constrain parallelism to ensure correctness. The intricate relationship between parallelism, synchronization and correctness is an important factor that makes parallel programming difficult. An incorrectly specified parallel program can lead to various anomalies that include deadlocks, unexpected non-determinism, or just poor performance.

Although, only the programmer alone can know and specify exactly what a program should do, it is possible for a programming environment to automatically identify a large class of possible programmer errors by static analysis of the program. Moreover, a static analysis tool can suggest transformations that will uncover additional parallelism. Information collected at runtime can sharpen the results obtained by static analysis. Static and dynamic analysis can be used together for debugging and removing performance bottlenecks.

Considerable amount of research has been dedicated to the development of tools and techniques to enhance parallelism and to aid programmers in parallel program development. The goal of this research is to develop an understanding of synchronization, so that the tools and techniques for program development can be extended to programs with explicit synchronization. The specific problem addressed in this thesis is to automatically detect if synchronization in a parallel program is sufficient to ensure that there is no anomalous behavior, and to point out potential anomalous behavior if it exists.

1.2 Research Background - PTOOL

PTOOL [ABKP86, BBC$^+$88] is a prototype tool developed at Rice University to aid parallel program development. The tool consists of two components, PSERVE and PQUERY. PSERVE is a static program analyzer that identifies potential regions for parallelization and collects data dependence information for the program. PQUERY is an interactive browser that provides parallelization related information to the user to aid parallelization. PTOOL has proven effective at assisting users in developing parallel programs [Hen87].

When we first implemented PTOOL, we assumed that programmers would use it on sequential Fortran programs that were to be converted to a parallel form suitable for execution on a shared memory multiprocessor. Hence, it only handled sequential

programs. However in practice, its users preferred to write a program in parallel form, invoking PTOOL only when a data race was discovered during testing. Users of PTOOL felt that a tool of this nature should accept "parallel Fortran" as input and understand its features, including synchronization statements. Also, in many instances PTOOL would report a data race that the user with superior knowledge of his program knew did not exist. This made it difficult for users to locate the actual sources of problems in their programs. An effort was made to overcome these limitations in a redesign of PTOOL, and in the design of ParaScope, a parallel programming environment being developed at Rice [CCH+87, BKK+89].

There are two main reasons why PTOOL would sometimes report a false data race. Since data dependence analysis assumes a dependence between two references to the same variable whenever the absence of a dependence cannot be proved, many dependences assumed to cause a data race did not actually exist. An effort was made to make the computation of parallelization-inhibiting dependences as accurate as possible. In particular, since dependences on variables that are private to a loop body do not inhibit parallelization, accurate detection of private variables significantly reduces the number of false data races that are reported. The other reason for false reports of data races was PTOOL's inability to understand synchronization. Since user inserted synchronization can be used to eliminate potential non-determinism caused by dependent statements specified to execute in parallel, it is important to analyze synchronization in a program and infer its effect on program behavior. In this thesis we discuss the handling of event variable style synchronization. Extensions to other synchronization mechanisms are also briefly discussed.

Through the proper use of event synchronization, the user can insure that a particular dependence is always satisfied by forcing the code at the sink of the dependence to wait until the code at the source has been executed. This thesis develops a method for determining which dependences in a program cannot possibly result in a data race because of schedule restrictions enforced by event synchronization.

1.3 Related Work

We discussed in the last section that experience with PTOOL was the main motivation for this thesis. In this section we discuss other parallel programming environments and research in anomaly detection. While we are not aware of any commercial ambitious

parallel programming environments, there have been several research efforts in that direction.

Taylor [Tay83a] has established that most interesting questions about the synchronization structure of ADA programs (for example, can there be a deadlock?) are NP-hard. He suggests an algorithm for static analysis of programs based on establishing concurrency states [Tay83b]. However the algorithm is roughly exponential in the number of concurrent tasks in the program.

The Anomaly Reporting Tool(ART) [AM85] is a system for developing parallel Fortran programs on shared memory multiprocessors. The static analysis phase of this tool examines the parallelization and synchronization constructs in a parallel program and constructs a *concurrency history graph* that represents all the *states* that the program can possibly enter. Program analysis and concurrency history graph can reveal various anomalies in the program, such as deadlocks and race conditions. The main drawback of this approach seems to be that the size of the concurrency history graph is potentially exponential in the number of possibly concurrent tasks. They claim that the the actual size is manageable for many real programs.

Another approach to developing parallel programs is used in the BLAZE programming environment [MR85, KMR87]. The language BLAZE is designed in a way that parallelism is easily exposed and can be exploited by a parallelizing compiler. Thus explicit parallelization is done solely by the compiler.

FAUST [GGGJ88] is a programming environment being developed at the University of Illinois. Its goal is to present a consistent user interface, which can be used for all aspects of program development. Interactive optimizing transformations are provided which can be tuned to various target architectures. The literature on this tool does not mention whether explicit synchronization mechanisms are supported.

Emrath and Padua [EP88] suggest some ideas on detection and classification of nondeterminism by analyzing synchronization on a shared memory multiprocessor.

Dinning and Schonberg [Sch89, DS90] investigate methods for "on the fly" detection of access anomalies in parallel programs. Padua and others [EP88, EGP89] discuss "post-mortem" debugging of parallel programs aimed at detecting parallel access anomalies. Miller and Choi [MC88], and LeBlanc and Mellor-Crummey [LMC87] present efficient methods to reconstruct a program's execution schedule from program traces for debugging and performance analysis.

ParaScope [BKK$^+$89] is a programming environment designed for interactive development and transformation of parallel programs. The implementation discussed in this thesis was developed inside ParaScope.

1.4 Overview of Research

An outline of the research presented in this thesis is enumerated below:

1. A model to reason about parallel programs with event style synchronization that is used for all our analysis procedures. Development of the notion of "schedule-correct" execution for parallel programs based on data dependences and execution orders.

2. A data flow analysis based algorithm to find execution orders in parallel programs to determine "schedule-correctness" and to find potential data races.

3. An algorithm to convert the problem of identifying data races to finding solutions of a system of linear equations in non-negative integers. The algorithm is based on finding regular expressions representing all paths between statements which are candidates for potential data races.

4. An algorithm to find if a system of linear equations has a solution in non-negative integers. This method is suitable for the problem instances encountered in synchronization and data dependence analysis.

5. Discussion of potential and limitations of extending these techniques to other synchronization mechanisms.

6. A prototype implementation to detect data races statically using some of the algorithms in this thesis.

Chapter 2

Analyzing Parallel Programs

The compiler analysis of parallel programs is fundamentally different from corresponding analysis for sequential programs. While the execution of a sequential program for a given data set corresponds to a unique ordering of statement instances, for a parallel program the set of statement instances that will execute may not be defined, and only a partial order is defined between the statement instances that do execute. In this chapter we shall describe a simple parallel language which captures the paradigm of programming on a shared memory multiprocessor, and discuss the fundamentals of analyzing execution orders in parallel programs. In subsequent chapters we will develop methods for synchronization analysis of parallel programs.

2.1 A Parallel Language

Our main interest is in parallel programs written for shared memory multiprocessors. The programming language assumed is Fortran 77, with parallel constructs that are supported on most parallel Fortran implementations. We now state the extensions to Fortran used for expressing parallelism and synchronization.

Parallel DO and Parallel Case

Parallelism can be expressed via two language constructs: parallel DO and parallel case. These are reasonably standard [FJS85, Par88] and we make no unusual assumptions. The parallel DO is syntactically and semantically very similar to the Fortran 77 DO loop. Here we use `DOALL` to indicate a parallel DO (illustrated in Figure 2.1(a)). When control reaches a parallel DO, all instances (iterations) of the loop body are started and proceed concurrently, but asynchronously. A parallel DO completes and control continues to the next program statement only when all instances of the loop body have completed.

The parallel case describes a fixed set of tasks to be executed concurrently. An example is illustrated in Figure 2.1(b), where each `Si` represents a list of statements.

```
DOALL 10 I = 1,10        PARALLEL BEGIN        POST(ev)
    ⋮                        S1                 ⋮
END DOALL                PARALLEL              WAIT(ev)
                             ⋮
                         PARALLEL
                             Sn
                         PARALLEL END

   (a)                      (b)                  (c)
```

Figure 2.1 Asynchronous parallel constructs and Synchronization statements

When control flow reaches a parallel case, each `Si` begins execution concurrently, but asynchronously, with the others. A parallel case completes and control continues to the next program statement only when all of the `Si` have completed.

Parallel constructs may be nested inside parallel constructs. Branches are not allowed from within a parallel construct to outside the parallel construct or vice versa.

We use the term **task** to refer generically to an instance of the body of a parallel DO or one of the statement lists of a parallel case. We will use the term **thread** to refer to a task while it is executing. When a parallel construct is executed, one thread is created for every task in the construct.

2.2 Synchronization

Many different mechanisms are used for synchronization in programs written for shared memory multiprocessors. These include locks, semaphores, event variables, critical sections and a few others. The analysis procedures developed in this thesis are based on inferring execution orders between statements. For most of the discussion in this thesis, we shall assume that synchronization is provided by *event* variables, which is one way of specifying execution orders between statements. In the rest of this section we briefly discuss some of the synchronization methods that are commonly used in shared memory multiprocessors. A more detailed discussion of how other synchronization methods fit into our analysis is postponed to Chapter 6.

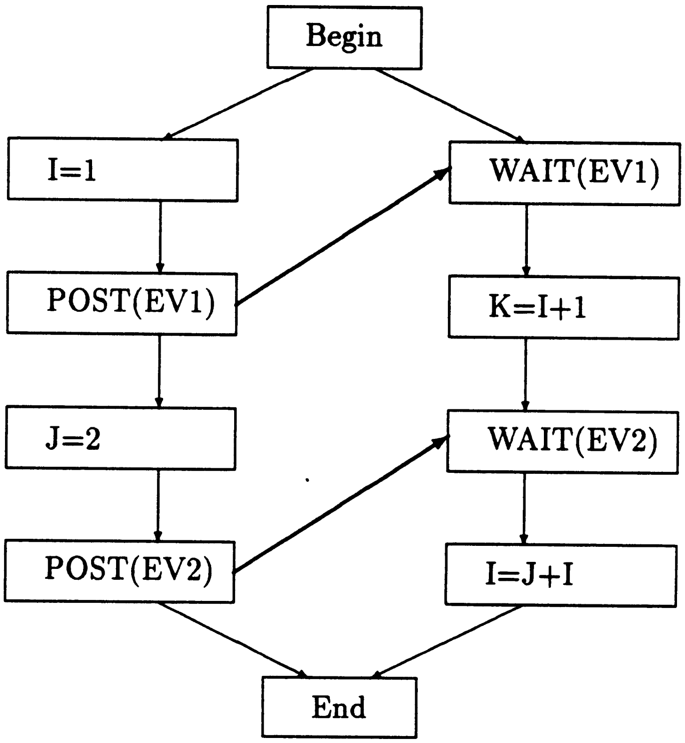
Event Variables

An event variable is always in one of two states: *clear* or *posted.* The initial value of an event variable when a program starts execution is always *clear.* The value of an event variable can be set to *posted* with the POST statement shown in Figure 2.1(c). The value of an event variable can be "tested" with a WAIT statement, also shown in Figure 2.1(c). A WAIT statement suspends execution of the thread which executes it, until the specified event variable's value is set to *posted.* A CLEAR statement resets the value of an event variable to *clear.*

As an example of the use of asynchronous parallel constructs with event variable synchronization, consider the program fragment shown in Figure 2.2(a). The parallel case defines two tasks: the first produces values of I and J, and the second uses these

```
PARALLEL BEGIN
   I = 1
   POST(EV1)
   J = 2
   POST(EV2)
PARALLEL
   WAIT(EV1)
   K = I + 1
   WAIT(EV2)
   I = I + J
PARALLEL END
```

(a)



(b)

Figure 2.2 Example of explicit synchronization

values. Figure 2.2(b) is a graphical representation of this program fragment. The thicker lines crossing between tasks represent synchronization between the tasks.

In the analysis procedures developed in the next two chapters, we assume the absence of CLEAR statements. We discuss the implications of CLEAR statements in programs in Chapter 6.

ADA Rendezvous, Ordered Critical Sections and Semaphores

These synchronization mechanisms specify certain order of execution between code sections. Thus it is possible to statically infer facts about execution orders from synchronization statements to enhance program analysis. In Chapter 6 we discuss how our analysis techniques can be extended to these kinds of synchronization mechanisms.

Locks and Critical Sections

Locks and critical sections specify code sections which must be executed as mutually exclusive blocks. By themselves, they do not provide any information about statement execution orders. Along with other control flow and synchronization constraints, however, locks and critical sections can provide useful static information.

2.3 Schedule-Correct Parallel Execution

We introduce the notion of "schedule-correctness" for parallel execution to characterize the kinds of program errors that are caused by incorrect parallelism in a program.

Sequential Execution

A canonical sequential execution schedule of a parallel program can be defined by describing a canonical sequential execution schedule for each of the parallel constructs. The sequential execution of a parallel DO is simply the obvious sequential iteration of the instances of the body, as if it were a sequential Fortran DO loop. The sequential execution of a parallel case is the sequential execution of statement lists in the textual order in which they occur in the program.

If during sequential execution of a parallel program, a WAIT statement is executed and the corresponding event variable is not already posted, then we say that the sequential execution of that program is **undefined**. If a program's sequential execution is defined for all input data sets, we classify that program as **serializable**. We

assume serializability throughout this thesis, but the techniques developed can also be used for non-serializable programs. Our primary interest is in debugging scientific programs (where parallelism is primarily a performance issue) using event variable synchronization for software pipelining, so the restriction to serializable programs is reasonable.

Correctness and Dependence Analysis

A **data dependence** [AK87, Kuc78] exists between two statements if they both access some memory location M and at least one of them modifies that location:

Definition 2.1 A data dependence $s_1 \Delta s_2$ (read "s_2 depends on s_1") exists if there is a memory location M such that both s_1 and s_2 access M, at least one of them stores into M, and, in the sequential execution of the program, s_1 is executed before s_2. A data dependence is *carried* by a loop L if s_1 and s_2 are statement instances in different iterations of L.

Much research has focused on identifying data dependence relationships and using them for loop restructuring transformations [KKP+81]. Data dependences capture the constraints on a program's data flow that are necessary to ensure correct results if a program is executed in parallel. If there are no data dependences between a pair of statements, the results of executing them are independent of the order in which they are executed.

An incorrect parallel program is defined to be a program whose output may differ from the sequential execution output on some input data:

Definition 2.2 A parallel program is **schedule-correct** relative to sequential execution if all parallel executions compute the same results as the sequential execution.

Since the execution state of a program consists of the program counter for each thread and the value for each memory cell, a difference between a parallel execution and the sequential execution must be recorded in memory at some point. In particular, for some memory location an incorrect program must contain a read/write, write/read, or write/write/read sequence that holds in the sequential execution but does not hold in some parallel execution. This notion is captured in the next definition:

Definition 2.3 A data dependence $s_1 \Delta s_2$ is **preserved** if, for all parallel executions, s_2 begins execution after s_1 has completed execution, whenever both are executed.

In a language with asynchronous parallel constructs, the semantics of a program are well-defined (schedule-independent) if there are no dependences between concurrent tasks. Thus, a parallel program is schedule-correct if all dependences are preserved on all parallel executions.

When a parallel program is not provably correct, any dependence which is not preserved is a potential error in the parallel program. Our objective is to be able to isolate such dependences and present them to the user as parallel execution hazards.

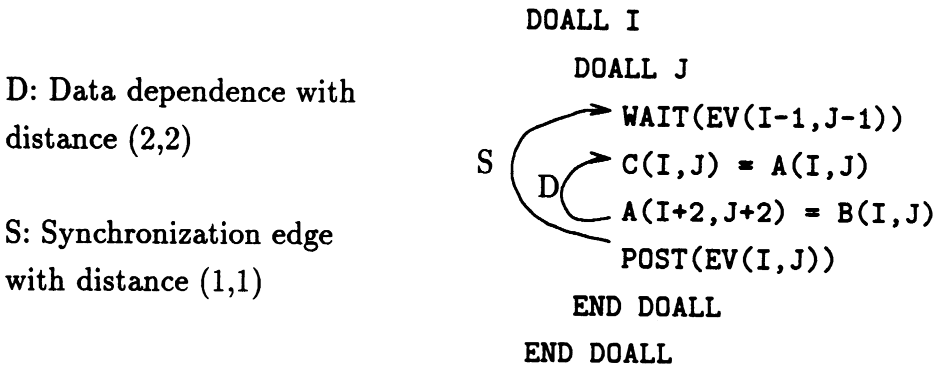
2.4 Distance Vectors in Programs with Loops

For reasoning about parallel programs, we have to identify and represent individual statement executions. This is not a problem for programs without loops, where mapping from static to dynamic statement instances is straightforward. However, for a program with loops, each static statement may correspond to multiple execution statement instances.

We associate an **iteration number** with every execution instance of a loop body, which is the order of execution of that instance of the loop body in the program's canonical sequential execution. Thus every execution of a statement is associated with a unique **iteration vector** whose components are the iteration numbers of all the enclosing loop bodies.

Data dependence analysis associates distance vectors with data dependences, denoting the number of iterations across which a dependence is carried. Here we state the definition of dependence distance vectors in terms of iteration vectors. Let there be a data dependence D from statement s_1 to s_2 (s_2 depends on s_1) due to an access to a variable in shared memory. If s_1 executing with iteration vector $\vec{i_1}$ always accesses the same memory location as s_2 executing with iteration vector $\vec{i_2}$, then $\vec{\delta_D} = \vec{i_2}$ - $\vec{i_1}$ is the **dependence distance vector** of data dependence D.

In our discussion we assume that all data dependence vectors are integer vectors that can be computed by data dependence analysis techniques. While this is by far the most common case, many exceptions are found in scientific programs. Our analysis procedures can use partial dependence distance information if constant integer vectors


D: Data dependence with distance (2,2)
S: Synchronization edge with distance (1,1)
S
D

```
DOALL I
  DOALL J
    WAIT(EV(I-1,J-1))
    C(I,J) = A(I,J)
    A(I+2,J+2) = B(I,J)
    POST(EV(I,J))
  END DOALL
END DOALL
```

Figure 2.3 Dependence and Synchronization distance vectors

do not exist or are not computable. In particular, if only direction vectors are available (it is only known whether the distance terms are positive, negative or zero) they can be used for partial analysis.

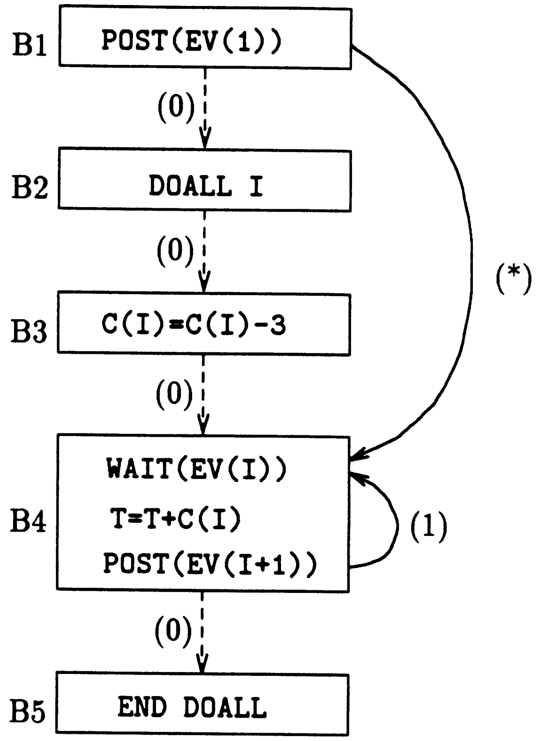
A distance vector term is meaningful for a POST and WAIT statement pair acting on the same event variable. Let s_1 be a POST statement and s_2 be a WAIT statement accessing a common set of memory locations (they act on the same event variable, which may be an array). If s_1 executing with iteration vector $\vec{i_1}$ always posts the same memory location that s_2 executing with iteration vector $\vec{i_2}$ waits on, then $\vec{\delta_S} = \vec{i_2} - \vec{i_1}$ is the **synchronization distance vector** for the *synchronization edge* from s_1 to s_2. The context of synchronization edges is a program graph discussed in the Section 2.5. Figure 2.3 shows an example of a dependence and a synchronization distance vector.

The synchronization distance vector defines the relative distance in terms of iteration numbers for which a pair of event variable directives are meaningful. Standard data dependence analysis techniques are normally able to compute the synchronization distance vectors, which are almost always integer vectors. We mark the synchronization distance components that are not simple integers by "*". They are not of use in our methods directly but can have special significance. One such special case, that of a synchronization edge coming in from outside of a loop, is discussed in Chapter 3.

2.5 The Synchronized Control Flow Graph

Analysis of synchronization is similar to the analysis of control flow in that no resolution is needed in straight-line code without synchronization statements. The primary

program representation used in this section is a modified form of the control flow graph, in which the nodes represent a form of basic blocks and edges represent both control flow and "synchronization flow".



```
POST(EV(1))
DOALL I = 1, 100
   C(I) = C(I) - 3
   WAIT(EV(I))
   T = T + C(I)
   POST(EV(I+1))
END DOALL
```

Figure 2.4 Synchronized Control Flow Graph for a Simple Loop

The **control flow subgraph** is a directed graph, $G_c = (N, E_c)$, where the nodes N are basic blocks [ASU86] built under the condition that a block in N must contain no control flow[1] and may contain at most one WAIT, which must be the first statement in the block, and at most one POST, which must be the last statement in the block. An edge represents a possible transfer of control from one block to another. Both parallel case and parallel DO are represented as a pair of nodes. One node represents the point where new threads are created (the fork node) and the other represents the point where threads are destroyed (the join node). Each task in a parallel construct is required to be a single-entry single-exit region. For each task inside the parallel

[1] This restriction is stronger than is necessary. Internal control flow can be allowed, for example, loops and complete if-then-else blocks without synchronization statements in their bodies can be a part of a basic block.

construct, there is an edge from the fork node to the entry of the task and an edge from the exit of the task to the exit node. There is no "back edge" from the bottom of a parallel DO to the top.

All edges in this graph are labeled with a distance vector which represents the iteration span over which the edge is meaningful. The control flow edges are trivially labeled with all zero distance vectors implying that the execution of the sink block with a particular iteration vector must follow the execution of the source block with the same iteration vector, if both are executed. However, in the case of the control flow edge from the last basic block in a sequential loop to the top of the loop, the last basic block in the previous iteration of the loop body must finish execution before the first basic block in the next iteration can begin execution. Correspondingly, the distance vector component corresponding to that loop is one, while all others are zero. A control flow edge from basic block b_i to b_j with distance vector $\vec{\delta}$ is denoted by $\langle b_i, b_j, \vec{\delta} \rangle$.

The **synchronization flow subgraph** is a directed graph, $G_s = (N, E_s)$, where the nodes N are the same as in the control flow subgraph, but the edges E_s link blocks which post events to blocks which wait on the same events. In particular, $\langle b_1, b_2, \vec{\delta} \rangle \in E_s$ if the last statement in block b_1 is POST(f(ev)) and the first statement in block b_2 is WAIT(g(ev)), where f(ev) and g(ev) are subscript expressions referencing the same event variable ev, and $\vec{\delta}$ is the synchronization distance vector, whose value is determined by the arguments to POST and WAIT statements as discussed earlier.

The **synchronized control flow graph**, SCG, is the combination of the control flow subgraph and the synchronization flow subgraph: $SCG = (N, E)$, where E is the direct union $E_c \cup E_s$.

Our objective is to analyze the synchronized control flow graph (referred to as SCG for the rest of the paper) and conclude which data dependences are preserved by the synchronization and control flow constraints, and hence cannot lead to data races. In the next two chapters we present two different methods to solve this problem, both based on the SCG.

Chapter 3

Synchronization Analysis as a Data Flow Problem

In the last chapter we showed that "schedule-correctness" can be established for parallel programs by proving that all the dependences in the program are preserved. The objective of this section is to develop a method of proving whether a dependence is preserved.

3.1 Execution Order

We shall develop some machinery to reason about execution orders in parallel programs. Let the loops in the program be numbered from 1 to n. Let `Executed` and `Completed` be arrays of k+*1* dimensions where k is the maximum nesting level of the program. Every location in these arrays potentially represents an execution of a program block. In the dimensions 1 through k, the array size is the highest possible iteration number at that level of nesting. The size of the array in dimension 0 is the number of basic blocks in the program, say m. Thus every possible execution of a basic block in one execution of the program can be mapped to at least one location in the `Executed` and `Completed` arrays in a straightforward way. All elements of these arrays are assumed initialized to `.FALSE.`.

We now describe a way of instrumenting the program by which the start and termination of execution of basic blocks is registered in the arrays `Executed` and `Completed`. These arrays will then represent the state of the program that we use to reason about execution orders.

For each block b_i in the program, perform the following:

Let k be the nesting level of b_i. Let l_1 to l_k be the iteration variables of loops enclosing b_i at nesting levels 1 to k respectively. We denote the k tuple $(l_1, l_2, ..., l_k)$ an iteration vector of k components as $\vec{l}$. At the beginning of block b_i, after the wait statement if there is one, insert the statement `Executed`$(i, \vec{l})$ `= .TRUE.`. At the end of each block b_i, before the post statement if there is one, insert the statement `Completed`$(i, \vec{l},)$ `= .TRUE.`. Given that $\vec{l}$ has k components, subscript set $(i, \vec{l})$ ad-

dresses *all* locations in the array whose first k subscripts match with $\vec{l}$ and the last subscript is i. Thus the statement `Executed(`$i, \vec{l}$`) = .TRUE.` has the effect of setting *all* locations in the array `Executed` whose first k subscripts match with $\vec{l}$, and the last subscript is i, to `.TRUE.`. The statement `Completed(`$i, \vec{l}$`) = .TRUE.` has a similar effect on the array `Completed`.

We define the following predicate:

$Executed^t_x(i, \vec{l}) \iff$ At time t during parallel execution x, all locations addressed by `Executed(`$i, \vec{l}$`)` have the value `.TRUE.`

and similarly for $Completed^t_x$. $Executed^t_x$ and $Completed^t_x$ are assumed to have the value `.FALSE.` if their argument corresponds to subscripts that are not in the range of corresponding arrays `Executed` and `Completed` respectively. We define the following predicates in terms of $Executed^t_x$ and $Completed^t_x$.

$$\begin{aligned} IsExecuted_x(i, \vec{l}) &\iff \exists t\colon Executed^t_x(i, \vec{l}) \\ IsCompleted_x(i, \vec{l}) &\iff \exists t\colon Completed^t_x(i, \vec{l}) \end{aligned}$$

$$ExecutedBefore_x((i, \vec{l}), (j, \vec{m})) \iff \forall t\colon (Executed^t_x(j, \vec{m}) \Rightarrow Completed^t_x(i, \vec{l}))$$

3.2 Preserved Sets

The preserved set of a basic block b_p is the set of basic blocks that must complete execution before b_p can start executing. Since program basic blocks can have multiple execution instances, the preserved sets can be defined for execution instances of basic blocks, and have execution instances of basic blocks as members of preserved sets.

Definition 3.1 The **Preserved** set for a pair $(i, \vec{l})$ denoting an execution instance of block b_i corresponding to iteration vector $\vec{l}$, is defined in terms of such pairs and possible parallel executions by:

$$Preserved(i, \vec{l}) = \{(j, \vec{m}) \mid \forall x\colon (IsExecuted_x(j, \vec{m}) \Rightarrow ExecutedBefore((j, \vec{m}), (i, \vec{l})))\}$$

Thus $(j, \vec{m}) \in Preserved(i, \vec{l})$ if and only if for all parallel executions x, $\vec{m}$th iteration of basic block b_j is completed before the $\vec{l}$th iteration of block b_i is begun, if both are executed.

The *Preserved* set for a specific iteration of a basic block inside a loop nest can potentially be of size $O(\vec{l}_{max} * n)$ where $\vec{l}_{max}$ is the product of the maximum of the iteration ranges of the loops enclosing a basic block in the program, and n is the number of basic blocks in the program. Moreover there are $O(\vec{l}_{max} * n)$ such sets for a program. Fortunately the volume of information can be significantly reduced under the assumption that the synchronization information available is uniformly applicable over all iterations of a loop. This is equivalent to assuming that all components of all synchronization distance vectors are fixed constants. In our analysis we shall use only the synchronization information to which the above assumption applies. Ignoring other synchronization information can make our analysis less precise but cannot make it incorrect. The direct implication of this assumption is that we need to store only one preserved set per basic block in the loop body. We introduce a relation to capture facts obtained from uniform synchronization information only.

Definition 3.2 The **Preserved**$^\delta$ sets are defined for each block b_i by:

$$Preserved^\delta(i) = \{(j,\vec{\delta}) | \forall \vec{k}: (\exists x: (IsExecuted_x(i,\vec{k})) \Rightarrow (j,\vec{k}-\vec{\delta}) \in Preserved(i,\vec{k}))\}$$

Lemma 3.1 If all synchronization distance vectors in a program have constant components then

$$(j,\vec{m}) \in Preserved(i,\vec{l}) \Rightarrow (j,(\vec{l}-\vec{m})) \in Preserved^\delta(i)$$

This lemma states that uniform synchronization information over loops implies that *Preserved* relationship is also uniform.

We restate that a distance vector component whose value is not known is labeled as '*'. The addition/subtraction of two vectors with different number of components yields '*'s for all the components that are absent in one of the input vectors. Also addition or subtraction with a '*' component would yield a '*'.

Theorem 3.2 Proving a parallel program "schedule-correct" using only information from the synchronized control flow graph is co-NP-hard.

Proof:

This theorem holds even for programs without loops and the proof is constructed for such programs. The phrase "using only information from the synchronized control flow graph" implies that all control flow paths are assumed possible. In this proof we construct a program, and from it, a synchronized control flow graph which encodes an instance of the 3 CNF Satisfiability (3SAT) problem [AHU74]. It will be shown

that a particular data dependence in this program may not be preserved if and only if the 3SAT problem is satisfiable.

An instance of 3SAT can be stated as follows: let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of boolean variables and $C = \{c_1, c_2, \ldots, c_m\}$ be a collection of clauses over V with exactly three literals each: $c_j = l_j^1 + l_j^2 + l_j^3$ where each l_j^i is a variable in V or the negation of a variable in V. The collection C is satisfiable if there is an assignment of truth values to variables in V such that all clauses of C are satisfied (at least one literal is assigned a value of "true").

From the above stated instance of 3SAT we will construct a parallel program P including statements S_1 and S_2 such that S_2 can precede S_1 in a parallel execution of P if and only if C is satisfiable. P has two event variables called **ev**$_i$ and **env**$_i$ for every variable v_i in V. P consists of a PARALLEL CASE with two branches B_w and B_p:

```
PARALLEL CASE
   code for $B_p$
PARALLEL
   code for $B_w$
END PARALLEL
```

B_p consists of a sequence of n code fragments of the form:

```
if v$_i$
   then POST(ev$_i$)
   else POST(env$_i$)
```

followed by the statement:

```
$S_1$: T = 1
```

followed by $2 \cdot n$ statements:

```
POST(ev$_1$)
POST(env$_1$)
  ⋮
POST(ev$_n$)
POST(env$_n$)
```

B_w is constructed from the clause set C. B_w begins with m code fragments:

```
if $\mathtt{nd}_j^1$
   then WAIT($\mathtt{el}_j^1$)
else if $\mathtt{nd}_i^2$
   then WAIT($\mathtt{el}_j^2$)
else WAIT($\mathtt{el}_j^3$)
```

here $\mathtt{nd}_j^i$ are unique variables and $\mathtt{el}_j^i$ is defined by:

$$\mathtt{el}_j^i = \begin{matrix} \mathtt{ev}_k & \text{if } l_j^i \text{ is unnegated variable } v_k \\ \mathtt{env}_k & \text{if } l_j^i \text{ is negated variable } v_k \end{matrix}$$

The above sequence is followed in B_w by the statement S_2:

S_2: S = T

Observe that P is serializable. Since B_p has no wait statements, it can execute to completion. After it has executed, all of the event variables referenced by B_w are "posted" and so B_w can execute to completion. Note that in the sequential execution, statement S_1 stores into location T and S_2 reads from location T and so there is a data dependence from S_1 to S_2. The program is correct only if this dependence is preserved. We next show that the dependence may not be preserved if and only if C is satisfiable.

First suppose C is satisfiable. Let $A: V \mapsto \{true, false\}$ be an assignment of truth values to the variables in V such that C is satisfied. Let the branch B_p in P be executed until just before statement S_1, taking a control flow path through the POST($\mathtt{ev}_i$) block if $A(v_i) = true$ (corresponding to program variable $\mathbf{v}_i$ having value *true*), and taking the control flow path through the POST($\mathtt{env}_i$) block if $A(v_i) = false$ (corresponding to program variable $\mathbf{v}_i$ having value *false*).

Now begin executing B_w. By selection of A, each clause of C has one true literal. For c_j, assume that l_j^i is true under A. If l_j^i is the unnegated variable v_k, $A(v_k) = true$ and so during the partial execution of B_p, the statement POST($\mathtt{ev}_k$) was executed. Therefore, for some values of $\mathtt{nd}_j^1$ and $\mathtt{nd}_j^2$, the path that waits on $\mathtt{el}_j^i$ is chosen. Since l_j^i is the unnegated variable v_k, $\mathtt{el}_j^i$ is $\mathtt{ev}_k$, which was posted during execution of B_p. The case in which l_j^i is the negation of variable v_k is completely analogous. Hence, at the time immediately before S_1 is executed, there is a path from the beginning B_w through S_2 along which all event variables that appear in WAIT statements have been posted. We conclude that the dependence from S_1 to S_2 is not preserved.

To show the converse, assume that the dependence from S_1 to S_2 is not preserved in some parallel execution. At the point that S_2 is executed, and before S_1 is executed,

at most one of each pair of event variables $\mathbf{ev}_k$ and $\mathbf{env}_k$ has been posted. This defines a truth assignment A:

$$\mathrm{A}(\mathrm{v}_k) = \begin{array}{ll} \mathit{true} & \text{if } ev_k \text{ is posted} \\ \mathit{false} & \text{otherwise} \end{array}$$

The path from the beginning of B_w to S_2 taken in this parallel execution selects one literal from each clause. Assume that for clause c_j, the branch corresponding to l^i_j was taken. If l^i_j is the unnegated variable v_k, then $\mathbf{el}^i_j$ is $\mathbf{ev}_k$ and, since control has passed through this statement, $\mathbf{ev}_k$ must be posted. Hence $A(v_k) = \mathit{true}$, establishing that l^i_j is *true* and that c_j is satisfied. The case in which l^i_j is the negation of variable v_k is completely analogous. This argument holds for each clause in C and hence C is satisfied.

We have assumed that all execution paths are possible, so the above argument holds for any program whose synchronized control flow graph is isomorphic to the constructed program. Hence the problem of showing programs correct based on information in the synchronized control flow graph is Co-NP–hard.
End of Proof.

3.3 A Data Flow Formulation

We have established that the computation of preserved sets is intractable in general. Our approach in this section to find the least fixed point of a set of dataflow equations which compute an approximation to the $\mathit{Preserved}^{\delta}$ sets.

We first state some facts that hold for parallel programs that terminate normally.

Lemma 3.3 For all pairs $(i, \vec{l})$ denoting the $\vec{l}$th iteration of block b_i and all parallel executions x:

$$\forall t\colon (\mathit{Completed}^t_x(i, \vec{l}) \Longrightarrow \mathit{Executed}^t_x(i, \vec{l}))$$

and

$$\mathit{IsExecuted}_x(i, \vec{l}) \Longrightarrow \mathit{IsCompleted}_x(k, \vec{l})$$

This lemma states that if a block has completed execution, it has begun execution and that if a block begins execution, it completes execution.

Lemma 3.4 If b_i is not the root of the control flow graph, then for all parallel executions x and all iteration vectors $\vec{l}$ with which b_i executes:

$$\forall t\colon (\mathit{Executed}^t_x(i, \vec{l}) \Longrightarrow \exists \left\langle b_j, b_i, \vec{\delta} \right\rangle \in E_c\colon \mathit{Completed}^t_x(j, \vec{l} - \vec{\delta}))$$

This lemma states that control reaches a block from one of its immediate predecessors in the control flow subgraph.

Lemma 3.5 If b_i is the "join" node associated with a parallel case, then for all parallel executions x:

$$\forall t\colon (\mathit{Executed}_x^t(i,\vec{l}) \Longrightarrow \forall \left\langle b_j, b_i, \vec{\delta}\right\rangle \in E_c\colon \mathit{Completed}_x^t(j,\vec{l}-\vec{\delta}))$$

This lemma states that all cases in a parallel case construct must complete before control continues in the parent task. $\vec{\delta}$ trivially consists of all zero components since control flow edges coming into the join of a parallel case do not carry any distance.

Lemma 3.6 If b_i is the first node of a sequential DO loop, then for all parallel executions x:

$$\forall t\colon (\mathit{Executed}_x^t(i,\vec{l}) \Longrightarrow \forall \left\langle b_j, b_i, \vec{\delta}\right\rangle \in E_c\colon \mathit{Completed}_x^t(j,\vec{l}-\vec{\delta}))$$

whenever $\texttt{Completed}(j,\vec{l}-\vec{\delta})$ is defined.

This lemma states that for all iterations of a sequential loop, other than the first one, the previous loop iteration must have finished execution.

Lemma 3.7 If b_i waits on event variable **ev**, then for all parallel executions x:

$$\forall t\colon (\mathit{Executed}_x^t(i,\vec{l}) \Longrightarrow \exists \left\langle b_j, b_i, \vec{\delta}\right\rangle \in E_s\colon \mathit{Completed}_x^t(j,\vec{l}-\vec{\delta}))$$

This lemma states that if a basic block waits on an event variable, that block does not begin execution until the event variable is posted. Only the predecessors of the block in the synchronization flow subgraph can post this event variable and so one of them must complete before the block can begin execution.

The above facts are enough for the following data flow equations which compute *SCPreserved* sets, an approximation to *Preserved*$^{\delta}$ sets.

$$\begin{aligned}
\mathit{SCPreserved}(i) &= \mathit{CPreserved}(i) \cup \mathit{SPreserved}(i)\\
\mathit{CPreserved}(i) &= \begin{cases} \bigcup_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(\mathit{AddEdge}(j,i,\vec{\delta})\right) & \text{if } b_i \text{ is a ``join'' node;}\\ \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(\mathit{AddEdge}(j,i,\vec{\delta})\right) & \text{otherwise ;} \end{cases}\\
\mathit{SPreserved}(i) &= \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_s} \left(\mathit{AddEdge}(j,i,\vec{\delta})\right)\\
\mathit{AddEdge}(j,i,\vec{\delta}) &= \left(\{(k,\vec{\delta}+\vec{\delta'}) \mid ((k,\vec{\delta'}) \in \mathit{SCPreserved}(j)) \wedge (\vec{\delta}+\vec{\delta'}) < \vec{m}\} \cup \{j,\vec{\delta}\}\right)
\end{aligned}$$

Here $\vec{m}$ is the range of iteration space of the loop nest enclosing basic block b_k. The term "$(\vec{a} < \vec{b})$" returns *true* if and only if all components of the term "$(\vec{b}$ - $\vec{a})$" are non-negative.

The $SCPreserved^+$ sets are defined to be the least fixed point of the above system of equations.

Theorem 3.8 The above system of dataflow equations will reach a fixed point.

Proof:

We first prove that elements can only be added in successive computations of the SCPreserved set of a particular node. Let $SCPreserved_k$ be the the sets obtained after successively computing $SCPreserved$ sets k times by this system of equations over all the basic blocks of the program. Consider the computation of $SCPreserved_{k+1}(i)$ for some basic block b_i. Suppose an element $(m, \vec{\delta}^+)$ belongs to $SCPreserved_k(i)$ but does not belong to $SCPreserved_{k+1}(i)$. Without loss of generality we assume that $m \in CPreserved_k(i)$, since it must belong to $CPreserved_k(i)$ or $SPreserved_k(i)$ or both. We further assume that b_i is not the header node of a loop. The case where b_i is a loop header can be handled analogously. Now $(m, \vec{\delta}^+) \in CPreserved_k(i) \Rightarrow \forall < j, i, \vec{\delta} > \in E_c: (\vec{\delta}^+ - \vec{\delta}, m) \in SCPreserved_{k-1}(j)$.

By hypothesis $SCPreserved_{k-1}(i)$ is a subset of $SCPreserved_k(i)$ and therefore $\forall < j, i, \vec{\delta} > \in E_c: (\vec{\delta}^+ - \vec{\delta}, m) \in SCPreserved_k(j)$. This is sufficient to ensure that $(\vec{\delta}^+, m) \in CPreserved_{k+1}(i)$ and hence that $(\vec{\delta}^+, m) \in SCPreserved_{k+1}(i)$. Thus we have a contradiction which proves that $SCPreserved_k(i) \subseteq SCPreserved_{k+1}(i)$ for any k and any basic block b_i. Also $SCPreserved_0(i)$ is empty, hence the proof by induction. We have shown that elements can only be added (and not removed) in successive computations of SCPreserved sets.

Also the total number of possible elements of a SCPreserved set is finite since the number of distinct distance vector entries for the same basic block is bounded by the iteration space, and the number of basic blocks is finite.

Thus we have proved that the process of computing SCPreserved sets must reach a fixed point.

End of Proof.

Theorem 3.9 If *SCPreserved*$(k) \subseteq$ *Preserved*$^{\delta}(k)$ for all blocks b_k, then if b_i is the "join" node for a parallel case:

$$\begin{aligned} CPreserved(i) &= \bigcup_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(Addedge(j,i,\vec{\delta})\right) \\ &\subseteq Preserved^{\delta}(i) \end{aligned}$$

and otherwise:

$$\begin{aligned} CPreserved(i) &= \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(Addedge(j,i,\vec{\delta})\right) \\ &\subseteq Preserved^{\delta}(i) \end{aligned}$$

Proof:

We prove for the case when b_i is not a "join" node. The other case is analogous. Let $(m,\vec{\delta})$ be an element of *CPreserved*(i). We will show that $(m,\vec{\delta}) \in Preserved^{\delta}(i)$. Let x be a parallel execution such that, during x, block b_m is executed. If no such x exists, then $(m,\vec{\delta}) \in Preserved^{\delta}(i)$ vacuously. Let $\vec{l}$ be any iteration vector with which b_m is executed (see figure 3.1). By Lemma 3.4, at all times t, $Executed_x^t(i,\vec{l}+\vec{\delta})$ implies

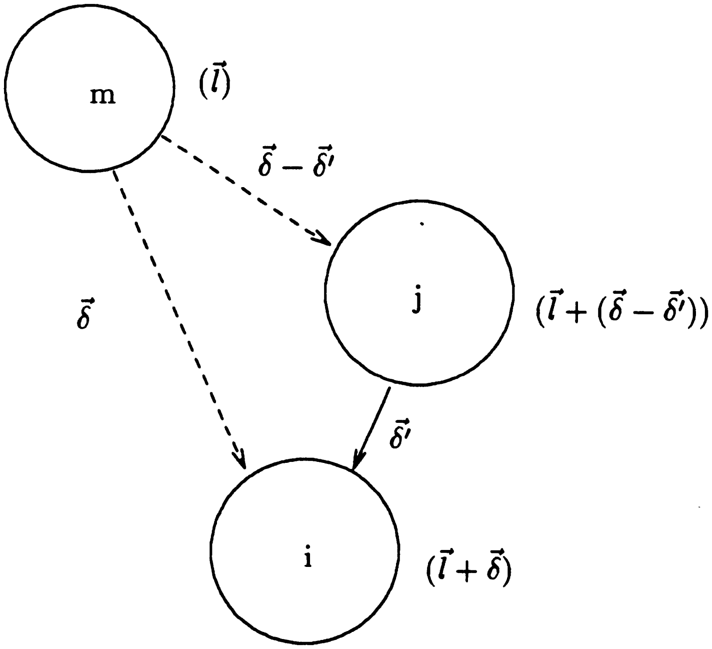

m
$(\vec{l})$
$\vec{\delta}-\vec{\delta}'$
$\vec{\delta}$
j
$(\vec{l}+(\vec{\delta}-\vec{\delta}'))$
$\vec{\delta}'$
i
$(\vec{l}+\vec{\delta})$

Figure 3.1

that there exist b_j and $\vec{\delta'}$ such that $\left\langle j, i, \vec{\delta'} \right\rangle \in E_c$ and $Completed_x^t(\vec{l} + (j, \vec{\delta} - \vec{\delta'}))$. By Lemma 3.3, we have $Executed_x^t(j, \vec{l}+\vec{\delta}-\vec{\delta'})$. Since $(m, \vec{\delta}) \in CPreserved(i)$ and we have a distance $\vec{\delta'}$ control flow edge from b_j to b_i, the equation to compute CPreserved set requires that $(m, \vec{\delta} - \vec{\delta'}) \in SCPreserved(j)$ and therefore, by hypothesis, $(m, \vec{\delta} - \vec{\delta'}) \in Preserved^{\delta}(j)$. From this and $Executed_x^t(j, \vec{l} + \vec{\delta} - \vec{\delta'})$ we conclude $Completed_x^t(m, \vec{l})$. Thus we have $Executed_x^t(i, \vec{l} + \vec{\delta})$ implies $Completed_x^t(m, \vec{l})$. This holds for all times t, all parallel executions x and all iteration vectors $\vec{l}$ for which $(m, \vec{l})$ is executed. Thus we conclude $(m, \vec{\delta}) \in Preserved^{\delta}(i)$ and the theorem follows. The proof for when b_i is a "join" node is identical except that Lemma 3.5 is used rather than Lemma 3.4 and "there exists j" is replaced with "for all j".
End of Proof.

Theorem 3.10 If $SCPreserved(k) \subseteq Preserved^{\delta}(k)$ for all blocks b_k then for every block b_i:

$$\begin{aligned} SPreserved(i) &= \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_s} (Addedge(j, i, \vec{\delta})) \\ &\subseteq Preserved^{\delta}(i) \end{aligned}$$

The proof of this theorem is identical to the proof of the previous theorem except that Lemma 3.7 is used rather than Lemma 3.4.

Theorem 3.11 For all programs P:

$$SCPreserved^{+}(i) \subseteq Preserved^{\delta}(i)$$

Proof:

The least fixed point of the above equations, $SCPreserved^{+}$, is computed by initializing the set $SCPreserved(i)$ for each block b_i to the empty set and then repeatedly computing $CPreserved(i)$ and $SPreserved(i)$ as illustrated earlier, and then replacing $SCPreserved(i)$ with $CPreserved(i) \cup SPreserved(i)$. By the two previous theorems, this basic step preserves the invariant that $SCPreserved(i) \subseteq Preserved^{\delta}(i)$ (which holds trivially initially) and so we conclude the invariant holds when the least fixed point is obtained. This theorem follows.
End of Proof.

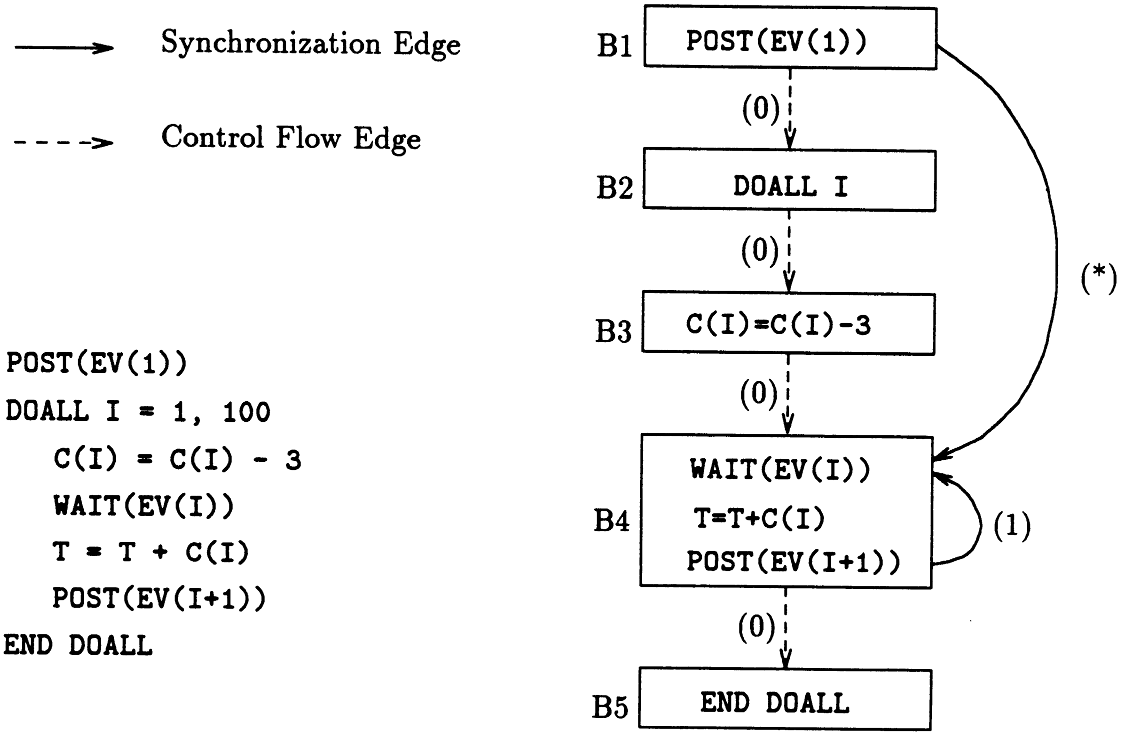
3.4 Adaptation for a Practical Implementation

In this section we shall further develop the techniques of the last section so that relevant information can be collected in reasonable time for practical programs. We shall point out some specific situations in which the framework developed does not do a reasonable job and discuss ways to overcome these shortcomings.

We use the example program in Figure 3.2 for illustration, which is the same program that we used in the last chapter to show a sample synchronized control flow graph. In this section we use the term *Preserved* to mean *Preserved*^{6} sets, since the difference between the two is not relevant to this discussion.

3.4.1 Synchronization Cycles

The number of distinct synchronization distance vectors which can be associated with one basic block, in the preserved set of another basic block can be $O(|m|)$ where $|m|$ is the maximum of the product of the iteration ranges of the loops enclosing a basic



```
POST(EV(1))
DOALL I = 1, 100
   C(I) = C(I) - 3
   WAIT(EV(I))
   T = T + C(I)
   POST(EV(I+1))
END DOALL
```

Figure 3.2 Initialization Edges and Synchronization Cycles

block. We can detect and represent large classes of synchronization distances by analyzing for *synchronization cycles* which would make the preserved sets compact and their computation more efficient. We shall illustrate this informally and then give the related formal definitions.

Consider the program in Figure 3.2. The backward synchronization edge from the basic block B4 to itself ensures that all iterations of B4 with iteration number less than k must complete execution before the kth iteration can start executing. This implies that $(B4, j)$ for all j less than the iteration range of the loop (100 here) belong to the preserved set of $B4$. Here $B4$ constitutes a synchronization cycle and the set of entries mentioned above can be written as $(B4, 1^*)$ implying that all $(B4, j)$ where j is a multiple of 1 (and in the iteration range) are in the preserved set of $B4$.

Definition 3.3 A sequence of blocks in the *SCG* $\{b_{j_0}, b_{j_1}, \ldots b_{j_{n-1}}\}$ constitute a **synchronization cycle** if there exist $\{\vec{\delta_{j_0}}, \vec{\delta_{j_1}}, \ldots \vec{\delta_{j_{n-1}}}\}$ such that $(j_i, \vec{\delta_{j_i}}) \in Preserved(j_{(i+1)\, modulo\, n})$ and there is an edge in the *SCG* from b_{j_i} to $b_{j_{(i+1)\, modulo\, n}}$.

A cycle in the *SCG* is a synchronization cycle, if for every edge in the cycle, the source basic block is in the preserved set of the sink basic block, with some distance vector. If b_j is a member of a synchronization cycle, then $(b_j, \vec{\delta})$ belongs to the preserved set of b_j, where $\vec{\delta}$ is the sum of distance vector terms around the cycle. Converse of this statement can be argued in an analogous manner. This yields the following theorem:

Theorem 3.12 A block b_m in the *SCG* is a member of some synchronization cycle if and only if there exists a $\vec{\delta}$ such that $(m, \vec{\delta}) \in Preserved(m)$.

Proof:

$(m, \vec{\delta}) \in Preserved(m)$ implies that b_m forms a synchronization cycle by itself by definition 3.3. Let b_m belong to a synchronization cycle. By the definition of Preserved sets, $(i, \vec{\delta_1},) \in Preserved(j)$ and $(j, \vec{\delta_2}) \in Preserved(k)$ implies $(i, \vec{\delta_1}+\vec{\delta_2}) \in Preserved(k)$. Applying this argument over the sequence forming the synchronization cycle yields that there exists a $\vec{\delta}$ such that $(m, \vec{\delta}) \in Preserved(m)$ which completes the proof.

End of Proof.

The above theorem provides a method of identifying synchronization cycles during the computation of preserved sets. We introduce a representation for a class of distance vectors to have the ability to capture and represent the effect of such synchronization cycles.

$$\vec{\delta^*} = \{n\vec{\delta}: n \text{ is a positive integer}\}$$
$$(i, \vec{\delta^*}) = \{(i, \vec{\delta}): \vec{\delta} \in \vec{\delta^*}\}$$

Theorem 3.13 $(i, \vec{\delta^*}) \subseteq \mathit{Preserved}(i)$ if $(i, \vec{\delta}) \in \mathit{Preserved}(i)$ for any basic block b_i.

Proof:

$(i, \vec{\delta}) \in \mathit{Preserved}(i)$ implies that if basic block b_i with iteration vector $\vec{I}$ is executed then b_i with iteration vector $\vec{I} - \vec{\delta}$ must be executed before it. Repeated application of this argument proves the result.

End of Proof.

The data flow framework described earlier in this section can be modified to detect synchronization cycles in a straightforward manner using Theorem 3.12. New members SCPreserved sets that are a consequence of synchronization cycles can be added compactly using Theorem 3.13. The *union* and *Addedge* functions have to be modified so that they understand the meaning of entries corresponding to synchronization cycles and know how to combine them.

3.4.2 Initialization Edges

When event variable arrays are used to synchronize iterations of a parallel loop, some locations of the arrays need to be *posted* to get the loop started up. In our *SCG* these lead to synchronization edges from outside to inside the loop. Without semantic analysis, the specific locations that are *posted* cannot be determined, and it is not possible to infer that these synchronization edges are meaningful only for starting up the loop, and not for synchronization between loop iterations. We need to treat these *initialization edges* as special cases so that we do not lose precision in our analysis. Following example illustrates this issue.

Let us examine how an iterative algorithm based on the data flow equations developed in the last section would work on the program in Figure 3.2. One iteration over the program nodes computes the fixed point shown in Figure 3.3. Note that the two synchronization edges coming into basic block $B4$ have no effect on the SCPreserved sets since the intersection of the elements due to these edges is null. Clearly it is desirable to know that the kth iteration of basic block $B4$ must execute before the $(k+1)$th iteration when both are defined, but this fact is not depicted in the computed SCPreserved sets. This is because the synchronization edge from $B1$ to $B4$, which is meant for starting up the first iteration, causes the effect of the synchronization edge

Block	CPreserved	SPreserved	SCPreserved
B1	ϕ	ϕ	ϕ
B2	$(<*>,1)$	ϕ	$(<*>,1)$
B3	$(<*>,1),(<0>,2)$	ϕ	$(<*>,1),(<0>,2)$
B4	$(<*>,1),(<0>,2),$ $(<0>,3)$	ϕ	$(<*>,1),(<0>,2),$ $(<0>,3)$
B5	$(<*>,1),(<0>,2),$ $(<0>,3),(<0>,4)$	ϕ	$(<*>,1),(<0>,2,),$ $(<0>,3),(<0>,4)$

Figure 3.3 Computation of SCPreserved Sets

from $B4$ to $B4$ to be ignored during analysis. We need to give special consideration to the initialization edge from $B1$ to $B4$ so that it does not hide the effect of other synchronization.

In general, if a dependence D between two statements with k component iteration vectors can be proved to be preserved by ignoring all synchronization edges other than those between nodes with with k component iteration vectors, D is indeed preserved. The only time wrong information can be obtained by ignoring a synchronization edge is when there are multiple synchronization edges into one node and only a subset of them are ignored. In this case such information is not relevant for loop carried dependences, although they do effect the way the loop starts up execution.

We use this idea to selectively ignore the effect of the initialization edges. In particular, if we are interested in dependences at level k and we have multiple synchronization edges coming into a basic block at level k, we ignore the effect of all the synchronization edges whose source basic block is *not* at k nesting level.

3.4.3 Forcing a Polynomial Solution

Although we consider the special handling of synchronization cycles discussed in the last section an important optimization, it does not ensure a polynomial time solution for the data flow equations. However, in practice we expect an iterative solution

to converge rapidly in most cases. Even if the computation of preserved sets is not completed, the relevant members of the preserved sets, which correspond to the data dependences in the program, may already have been computed. A polynomial solution can be forced by terminating the computation of preserved sets of a particular basic block, or of all basic blocks, at some stage in computation determined by a heuristic criterion. We shall briefly discuss two such criteria.

1. We can choose to stop further addition of elements to the preserved set of a specific basic block once the number of elements in its preserved set exceeds a predetermined maximum number. Let c be the maximum preserved set size that is chosen. Let n and e be the number of nodes and edges respectively in the synchronized control flow graph. Each iteration of the computation of the SCPreserved sets takes $O(ce)$ time since the members of the SCPreserved sets of the predecessors of each node are examined. Each iteration must add at least one element to the SCPreserved set of some basic block until a fixed point is reached. Since the total number of elements in all preserved sets of all basic blocks is at most nc, the computation must reach a fixed point in $O(c^2en)$ time.
2. Alternately we can limit the maximum value of the magnitude of any component of a distance vector during computation. This solution seems reasonable in the wake of our experience that dependence distance vector components tend to be small in real programs, and are usually determined during dependence analysis. This limits the maximum number of elements in the preserved set of any basic block to $O(k^dn)$ where k is the maximum value of any distance vector component and d is the maximum nesting depth of the program. Thus from the last case, the worst case complexity of computing the SCPreserved sets would be $O(k^den^3)$.

The upper bounds obtained on computation complexity discussed above may be overly conservative and do not necessarily reflect the actual time taken for real programs. The reason for this is that our methods are general enough to do a good job on programs with complex synchronization structure, but real programs tend to have a relatively simple synchronization structure.

Chapter 4

Algebraic Formulation of Synchronization Analysis

In a large class of programs, the problem of verifying whether a dependence is preserved can be transformed to the problem of determining whether a non-negative integer solution exists to a system of linear equations. The transformation procedure takes nearly linear time for the programs that can be directly analyzed in this manner, and the method can be used to get precise (up to symbolic execution) results. Solving a system of equations in non-negative integers is a form of the integer programming problem and is known to be NP-hard. However, due to the nature of the problem, the actual systems generated are expected to be small and easily solvable by heuristic procedures. In this chapter we present a transformation procedure to systems of linear equations. In the next chapter we shall discuss practical approaches to solving the equations generated.

4.1 Formulating Systems of Equations

In this section we describe how the synchronization analysis problem for programs satisfying certain restrictions can be transformed to the problem of solving systems of equations in non-negative integers. In the next section we discuss the generalization of this solution method.

4.1.1 Program Restrictions

The method we describe is directly applicable to all programs whose synchronized control flow graph(SCG) has at most one synchronization edge and at most one control flow edge into each block, with the exception that multiple control flow edges may be incident on the *join* node of a parallel case statement. The program blocks, as discussed in chapter 2, are a variation of basic blocks used in compiler analysis. In the next section we discuss a modified definition of program blocks that makes it possible to analyze a larger set of programs

4.1.2 Transformation Procedure

The solution procedure that we shall present is based on the following facts:

> **Lemma 4.1** Let there be a control flow edge from block b_p to block b_q with distance $\vec{c}$. Given that block b_q with iteration vector $\vec{l}$ starts executing in a program run, and $\vec{c} = \vec{m} - \vec{l}$, block b_p with iteration vector $\vec{m}$ has completed execution.

This lemma states that if there is a control flow edge coming into a node, then a certain iteration of the predecessor node must complete execution before the node can start executing. The actual iteration is determined by the distance vector of the edge connecting the two nodes.

The following lemma makes an analogous assertion for blocks connected by synchronization edges:

> **Lemma 4.2** Let there be a synchronization edge from block b_p to block b_q with distance $\vec{s}$. Given that block b_q with iteration vector $\vec{l}$ starts executing in a program run, and $\vec{s} = \vec{m} - \vec{l}$, block b_p with iteration vector $\vec{m}$ has completed execution.

The above two Lemmas are sufficient to prove the following theorem:

> **Theorem 4.3** A dependence from a block b_{src} to block b_{snk} with distance $\vec{\delta}$ can be proved to be preserved (using only the information in the SCG) if and only if there exists a path from b_{src} to b_{sink} of distance $\vec{\delta}$ in the synchronized control flow graph of the program.

Proof:

Suppose there does exist a distance $\vec{\delta}$ path from b_{src} to b_{snk}. Let b_p be a node on the path and b_q be its predecessor. Let the edge from b_p to b_q be a synchronization edge of distance $\vec{d}$. By Lemma 4.2, for any $\vec{i}$ if $(q, \vec{i})$ begins to execute, $(p, \vec{i} - \vec{d})$ must have completed execution. If there is a control flow edge from b_p to b_q, we obtain the same result using Lemma 4.1. Since this result holds for all the nodes in the path from b_{src} to b_{sink}, we infer that if $(sink, \vec{i})$ begins to execute, $(src, \vec{i} - \vec{\delta})$ must have completed execution, where $\vec{\delta}$ is the distance along the path from b_{src} to b_{snk}. This is equivalent to saying that a distance $\vec{\delta}$ dependence from b_p to b_q is preserved.

If there is no distance $\vec{\delta}$ path from b_{src} to b_{snk}, there is no constraint on the order of execution of iterations of src and snk separated by a distance $\vec{\delta}$. Thus, with the information in the SCG, we cannot prove that the dependence is preserved.

End of Proof.

An algorithm developed by Tarjan [Tar81] can be used to find a canonical regular expression representing all paths between any two nodes in a graph efficiently[1]. We now show how the regular expression representing all paths between the source and the sink of a dependence can be used to determine if the dependence is preserved.

Theorem 4.4 Let R be a regular expression over an alphabet consisting of all the edges in a program's SCG say $e_1, e_2, ...e_n$. Let $val(e_i)$ represent an integer vector which is the distance vector attached to e_i. There exists a set of systems of linear equations which has a solution in non-negative integers if and only if there exists a string in R, the value of whose symbols adds up to a fixed integer vector $\vec{\delta}$.

Proof:

We show how a set of systems of equations can be constructed, which is equivalent to the given regular expression in the above sense. We construct a set of linear expressions which represent all possible "values" which the regular expression can take. Equating these to $\vec{\delta}$ would give the systems of equations desired.

We build the value expressions by a bottom up traversal of the parse tree representing the regular expression. Figure 4.1 shows the possible productions used in parsing and the action taken. The value expression of a node in the parse tree is constructed from its children in accordance with these actions.

The set of expressions computed for a node in the parse tree is referred to as *expset* which is in the following form:

$$\begin{aligned} expset(R_1) &= exp(R_1^1) \cup exp(R_1^2) \cup ...exp(R_1^n) \\ exp(R_1^i) &= c_0^i + c_1^i x_1^i + c_2^i x_2^i + \cdots c_{m_i}^i x_{m_i}^i \end{aligned}$$

Here symbols of form c_j^i represent fixed integer constants and symbols of form x_j^i represent variables that can take non-negative integer values. In Figure 4.1, we assume that the expressions are in this form and are denoted by the same symbols.

We will explain and justify the actions for productions shown in table Figure 4.1. The production $R \longrightarrow e_i$ assigns the value of nonterminal e_i as the only expression in *expset* of R. The production $R \longrightarrow R_1^*$ assigns all the expressions in the $expset(R_1)$ to $expset(R)$ with the following modification: If there is a constant term, say c, in an expression, it is replaced by the term xc where x is a new variable which can take any

[1] The algorithm takes $O(m\alpha(m,n))$ on a reducible flow graph with n vertices and m edges where α is the inverse of Ackermann's function.

$R \longrightarrow e_i$	$expset(R) = \{val(e_i)\}$
$R \longrightarrow R_1^*$	$expset(R) = \bigcup_{i=1}^{n}\{(exp(R_1^i))^*\}$ *where* $(exp(R_1^i))^* = c_0^i x_0^i + \sum_{j=1}^{m_i} c_j^i x_j^i$ *given that* $exp(R_1^i) = c_0^i + \sum_{j=1}^{m_i} c_j^i x_j^i$
$R \longrightarrow R_1 + R_2$	$expset(R) = expset(R_1) \cup expset(R_2)$
$R \longrightarrow R_1.R_2$	$expset(R) = \bigcup_{i=1,j=1}^{i=n_1,j=n_2}\{exp(R_1^i) + exp(R_2^j)\}$

Figure 4.1 Computing the set of equations from a regular expression parse tree

non-negative value. No change needs to be made for other terms since they have a variable term that can take any non-negative integer value, and the effect of a closure does not affect the numerical values. The production $R \rightarrow R_1 + R_2$ makes $expset(R)$ the union of $expset(R_1)$ and $expset(R_2)$. The production $R \rightarrow R_1.R_2$ takes the sum of all pairs of terms, selecting one from $expset(R_1)$ and the other from $expset(R_2)$, and makes the new terms members of $expset(R)$. Note that the general form of *expset* does not change as the new sets of expressions are constructed during parsing.

We restate that the purpose of the transformation is to check if the sum of values of any string in the regular expression can evaluate to a given fixed constant vector $\vec{\delta}$. Equating each expression obtained in the final *expset* of the regular expression to $\vec{\delta}$ yields a system of linear equations in variables which are restricted to non-negative integer values. If any of these systems of linear equations has a solution in non-negative integers, the original regular expression can evaluate to $\vec{\delta}$.

End of Proof.

4.1.3 An Example

We illustrate the ideas contained in the above discussion with the example in Figure 4.2

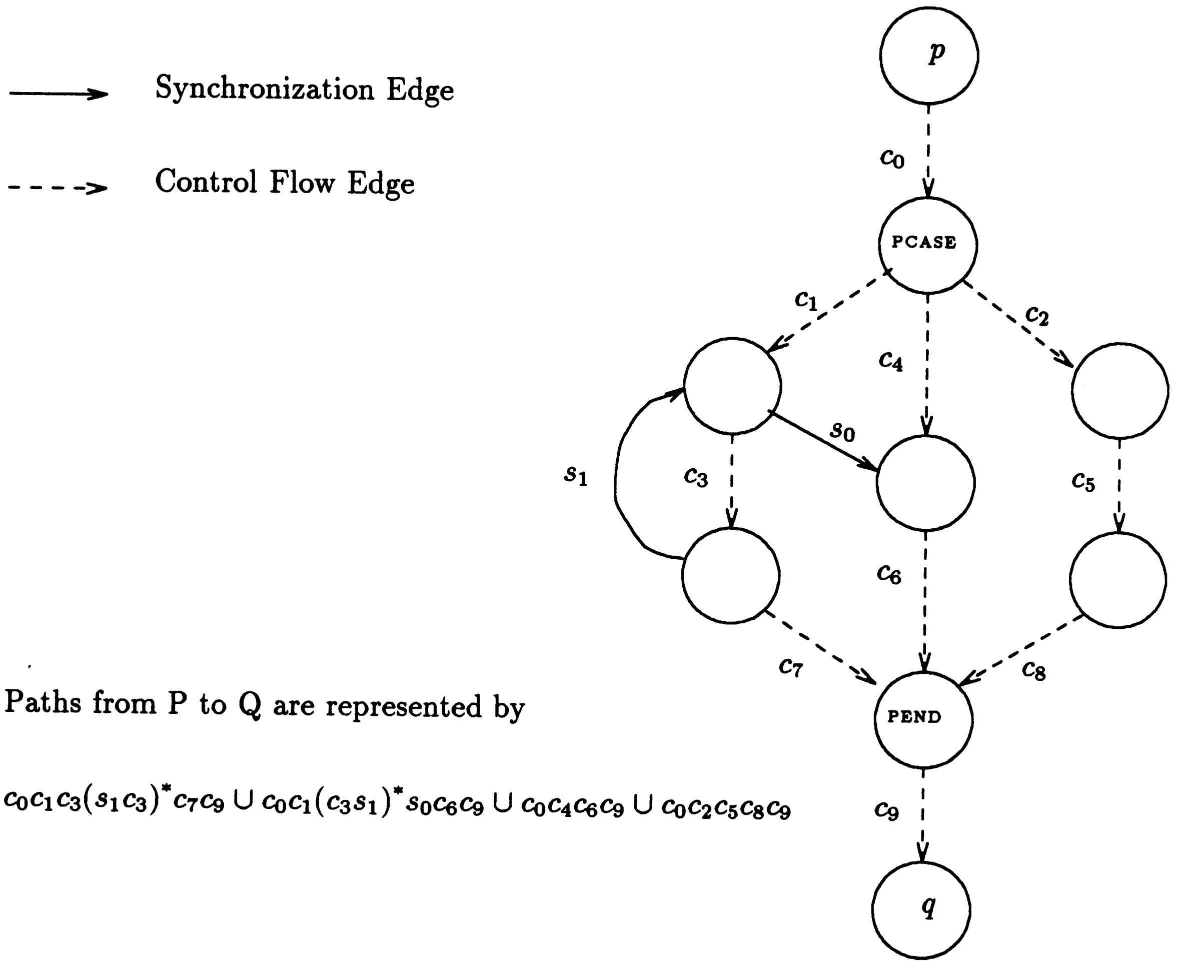

Synchronization Edge
Control Flow Edge
p
c_0
PCASE
c_1
c_2
c_4
s_0
s_1
c_3
c_5
c_6
c_7
c_8
PEND
c_9
q
Paths from P to Q are represented by
$c_0c_1c_3(s_1c_3)^*c_7c_9 \cup c_0c_1(c_3s_1)^*s_0c_6c_9 \cup c_0c_4c_6c_9 \cup c_0c_2c_5c_8c_9$

Figure 4.2 Set of paths between two nodes in a SCG graph

Figure 4.2 shows the synchronized control flow graph of a code section inside a loop nest. The set of paths from b_p to b_q is represented by the regular expressions:

$$p \to q = c_0c_1c_3(s_1c_3)^*c_7c_9 + c_0c_1(c_3s_1)^*s_0c_6c_9 + c_0c_4c_6c_9 + c_0c_2c_5c_8c_9$$

Since the control flow edges do not carry any distance, an equivalent regular expression for the purpose of analyzing distances is:

$$R_{p\to q} \doteq s_1^* + s_1^* s_0$$

Let the edges s_0 and s_1 have synchronization distance vectors $\vec{S_0}$ and $\vec{S_1}$ associated with them.

Let $\vec{\delta}$ be the distance of a dependence from b_p to b_q. By Theorem 4.3, this dependence is preserved if and only if there is distance $\vec{\delta}$ path from b_p to b_q.The regular expression stated above represents all the paths from b_p to b_q. By the construction procedure used in the proof of Theorem 4.4 the equivalent set of distance expressions for $R_{p\rightarrow q}$ is :

$$R_{p\rightarrow q} = \{(x_1\vec{S_1}), (x_2\vec{S_1} + \vec{S_0})\}$$

Here x_1 and x_2 are variables that can take any non-negative value.

By the statement of Theorem 4.4, a distance $\vec{\delta}$ path from b_p to b_q exists, and hence the above dependence is preserved if and only if there exist non-negative integral solutions to either of the following two systems of linear equations:

$$x_1\vec{S_1} = \vec{\delta}$$
$$x_2\vec{S_1} + \vec{S_0} = \vec{\delta}$$

Solving a system of equations in non-negative integers is known to be NP-hard. However, this formulation would normally lead to a small system of equations. Moreover, the coefficients of individual terms relate to components of distance vectors which are normally small integers. In the next chapter we will present practical approaches to solving such systems.

4.2 Analyzing more Complex Programs

The method of last section can be directly applied only to programs in which each block has at most one synchronization edge, and at most one control flow edge, incident on it. This suggests that only a small set of programs can be analyzed by these methods. In this section we present improvements to enable analysis of a wider class of programs and to make the analysis more efficient.

4.2.1 Extended Blocks

The blocks of *SCG* defined in Chapter 2 make a fine division of the program over which analysis procedures can be applied. We can also apply the procedure over larger blocks of the program, if the coarser dependence and synchronization information is sufficient for our purpose. Any lexically contiguous section of a program that satisfies the following criterion can be a single block:

1. Control flow can enter the block only through the first statement in the block, and can leave the block only from the last statement in the block.
2. Any synchronization edge entering the block must be incident on the first statement, and any synchronization edge leaving the block must leave from the last statement.
3. If there is a synchronization edge between two statements inside the block with a distance that is not zero for a certain enclosing loop, the whole loop must be part of the block.

Thus, internal control flow, including complete *IF* blocks and loops is permitted. Internal synchronization flow is also permitted, so long as it does not hide information relevant to the other blocks. The nesting level that we associate with a block is the nesting level of the first and the last statement in the block, which must be the same. We refer to the execution instances of these blocks as the execution instances of the bounding statements. Using larger program blocks gives us a coarser partition of the program. Since we can have statements with multiple control flow or synchronization edges incident on them internal to a block, a larger set of programs can be analyzed. Since the number of program nodes and edges is smaller, the analysis procedure would run faster. The obvious limitation of the approach is that it is possible to coalesce smaller blocks into larger blocks only if dependences between smaller blocks are not of concern.

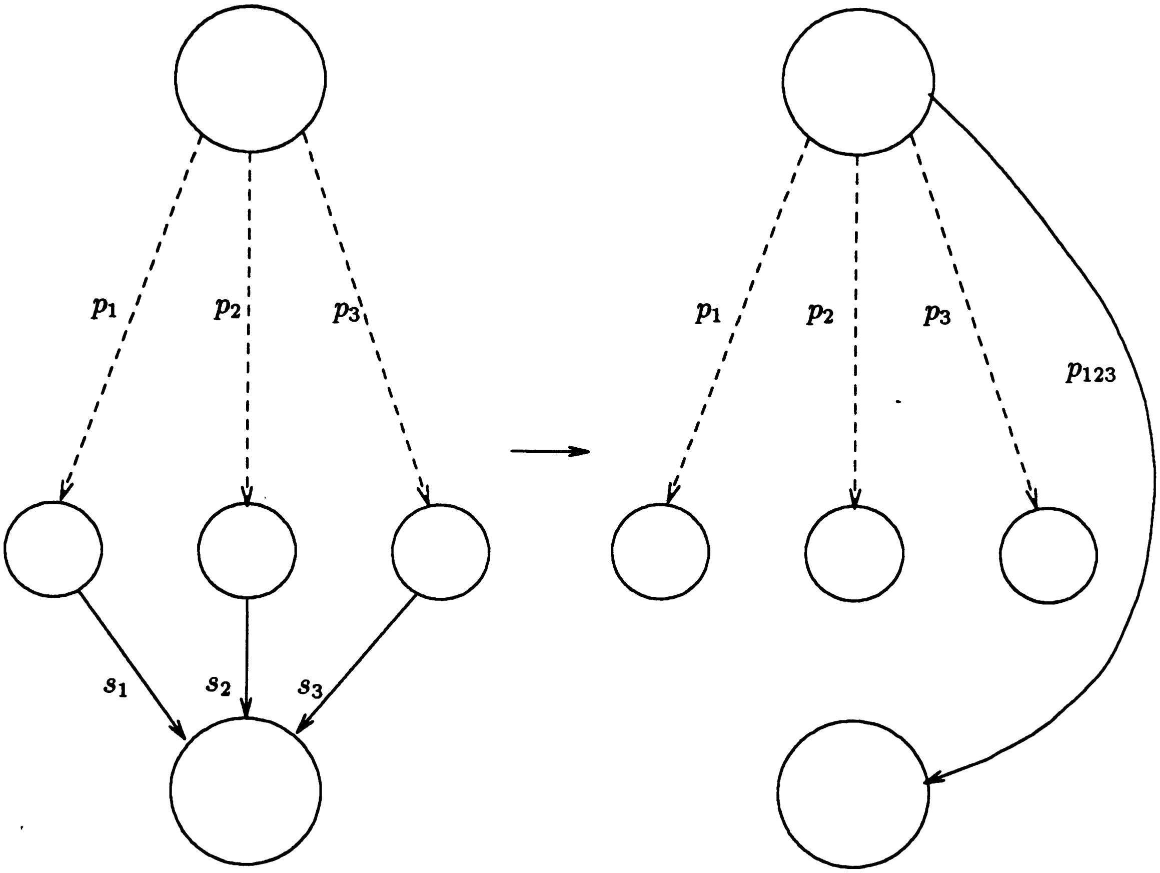
4.2.2 Eliminating Multiple Edges

We discuss a procedure by which a program having multiple edges of the same type into a node can be transformed to another program without such nodes.

Let b_p be a block in the SCG of a program. Let b_r be another block which has t edges of the same kind ($t > 1$) incident on it. Without loss of generality, we assume that these are synchronization edges and label them $s_1, s_2, ...s_t$. Further assume the following conditions:

1. For each synchronization edge s_i into b_r, there is a path from b_p to b_r including s_i.
2. There is no path from b_p to any predecessor of b_r which passes through any node which has multiple predecessors of the same kind.

If these conditions are satisfied, we perform the following transformation. For every synchronization edge s_i into b_r, let R_i be a regular expression representing all



$$R_{p_{123}} = R_{p_1.s_1} \cap R_{p_2.s_2} \cap R_{p_3.s_3}$$

Figure 4.3 Eliminating multiple in-edges in the *SCG*

paths from b_p to b_r which include synchronization edge s_i. Remove all t synchronization edges going into b_r. Add a single synchronization edge from b_p to b_r with a distance term R which is the intersection of regular expressions $R_1, R_2, ...R_t$. We have the following theorem:

Theorem 4.5 If the transformation procedure described above is applied when the conditions for it are satisfied, the resulting SCG is equivalent to the original SCG for the purpose of computing whether a dependence between b_p and any other program block is preserved.

Proof:

Suppose we are interested in finding if a dependence from b_p to b_q, d_{pq} with distance $\vec{\delta}$ is preserved. We shall show that the analysis of the transformed SCG will determine

that it is preserved if and only if it is possible to prove that it is preserved in the original SCG.

First we assume that the dependence d_{pq} is provably preserved in the modified SCG. Suppose it is critical for the decision procedure that there exist a distance $\vec{\delta'}$ path from b_p to b_r, including a synchronization edge (only s_R here). If this is not true, the analysis is unaffected by modifying the SCG. The path expression for s_R is the intersection of all paths from b_p to b_r which are incident on b_r via a synchronization edge in the original SCG. A distance $\vec{\delta'}$ path from b_p to b_r via s_R in the modified SCG implies that for each synchronization edge into b_r, there is a distance $\vec{\delta'}$ path from b_p to b_r including that edge, in the original SCG. This is sufficient information to infer that b_p, $\vec{\delta'}$ distance away must precede an execution of b_r. This is equivalent to having a single synchronization edge of distance $\vec{\delta'}$ from b_p to b_r. Thus we effectively have the same information in the original SCG and can prove that the dependence under consideration is preserved.

With analogous reasoning about equivalence of paths between the original and the modified SCG, we can show that if a dependence cannot be proved to be preserved in the modified SCG, it cannot be proved to be preserved in the original SCG.

End of Proof.

Intersection of two regular expressions of length n can potentially yield a regular expression of length $O(n^2)$. Thus, it is possible to handle a wider class of programs at an additional cost. However there are cases where the transformation procedure cannot be applied because of the conditions stated earlier.

4.3 Comparing the Two Methods

In the last chapter, we presented a method of synchronization analysis where synchronization information is collected for the program by an iterative algorithm acting on the program flow graph, analogous to data flow analysis, which we call the "dataflow" method. In this section we shall compare it with the "algebraic" method of this chapter.

The ordering information between statements is defined by the characteristics of the paths between them in the synchronized control flow graph. In the "dataflow" method we iterate along program paths propagating information along the program graph edges. At every step we do some vector arithmetic to compute the

new information obtained at the nodes. In many situations this leads to a simple algorithm which terminates in a single pass yielding the necessary information for all statement pairs.

However, loops in the synchronized control flow graph representation, (note that this is different from a program containing loops) may result in cyclic computation and a fixed point may not be reached in polynomial time. Although heuristic methods can be used to terminate the algorithm after all useful computation has been completed, a high computation cost may be involved in reaching this stage.

In the "algebraic" method, path information is computed symbolically before any arithmetic computation is performed. Facts like repeated computation over a cycle are captured before any actual computation is performed. This leads to the formation of a system of equations which need to be solved to obtain necessary information. Thus actual arithmetic is needed only for solving a standard mathematical problem and hence can be performed more efficiently.

There are two main drawbacks to this approach. Firstly it is a two step method and would perform slower on simple examples because of the extra overhead it causes. Secondly, this method is directly applicable only to a subclass of problems, although methods in this chapter can be used to alleviate the problem. However this method avoids the potential high cost of the "data-flow" method in many situations. Overall we consider this method to be more suitable for real applications, although our experience in this regard is limited.

Chapter 5

Solving Linear Equations in Non-negative Integers

We showed in Chapter 4 that the problem of determining whether the synchronization present in a program is sufficient to protect a dependence can be transformed to the problem of determining whether a solution in non-negative integers exists for a system of linear equations. This problem is known to be NP-hard. In this chapter we present a practical solution technique for solving the problem. This chapter is self contained and the methods developed can be used for solving such systems in other contexts as well. In particular, the problem of determining whether a data dependence exists between two array references can be transformed to this form [AK87, Wol82]. In our methods, we take advantage of the characteristics of the systems that we are likely to encounter. We state these as follows:

- The number of variables corresponds to the number of synchronization operations relevant to a specific data dependence and is expected to be small.
- The number of equations corresponds to the number of subscript positions in the array references causing a data dependence and is expected to be small.
- The coefficients of terms are synchronization distance vector components and are usually small integers, very often one or zero.

We first solve the system for all (not necessarily non-negative) integer solutions. This is often referred to as solving linear diophantine equations. If such a solution is unique or does not exist, we determine this fact and do not need to proceed any further. If there are multiple solutions, we obtain parametric equations that describe the solution space. Next we determine the non-negative solution space and check whether it is empty. If it is not empty, we use heuristic search procedures to determine if it contains an integer solution.

5.1 Solving Systems of Linear Diophantine Equations

The theory of solving linear diophantine equations has been discussed in many texts addressing number theory and integer programming [KHL77, GN72]. Our contribution is to present one solution method in a way that can be easily programmed on a computer and analyze the complexity. We also discuss why we chose the specific method that we present.

The problem can be stated as follows:

Find all integer solutions of

$$[A]_{m\times n}.[x]_{n\times 1} = [b]_{m\times 1}$$

We first present the necessary mathematical concepts, and then an algorithm to solve the problem.

5.1.1 Mathematical Concepts

Following are the *elementary* column operations on a matrix:

1. exchanging two columns
2. multiplying a column by -1
3. adding an integral multiple of one column to another column

Corresponding elementary row operations are similarly defined. We shall use elementary operations to transform a matrix to a form desirable for solving the equations. In the rest of this section, we introduce two kinds of matrices that describe specific elementary operation sequences that we shall use.

A *transposition* matrix $[P]_{n\times n}$ is a square matrix of 1s and 0s in which:

1. each row and each column has exactly one 1.
2. all the 1's except two are along the main diagonal.

The matrix shown below, P_{25} is a transposition matrix of order 5. The subscript 25 signifies that the 1s in rows(and columns) 2 and 5 are out of place relative to a unit matrix.

$$P_{25} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Theorem 5.1 Premultiplying a matrix A with a transposition matrix P_{ij} is equivalent to interchanging rows i and j of A, and postmultiplying A with P_{ij} is equivalent to interchanging columns i and j of A.

We show this with an example where premultiplying A with P_{25} yields A with rows 2 and 5 permuted.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{51} & a_{52} & a_{53} & a_{54} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix}$$

A *Subtraction* matrix $[S_{ij,\alpha}]_{nxn}$ is a square matrix in which:

1. all items along the main diagonal are 1.
2. $S[i,j] = -\alpha$.
3. all other elements are 0.

Following is an order 5 subtraction matrix $S_{13,\alpha}$

$$S_{13,\alpha} = \begin{bmatrix} 1 & 0 & -\alpha & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Theorem 5.2 Premultiplying a matrix A with a subtraction matrix $S_{ij,\alpha}$ has the effect of reducing the ith row of A by α times jth row of A. Postmultiplying a matrix A with a subtraction matrix $S_{ij,\alpha}$ has the effect of reducing the jth column of A by α times ith column of A.

We show this with an example where postmultiplying A with $S_{13,\alpha}$ yields A with every element of column 1 reduced by α times the corresponding row element in column 3.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{bmatrix} \begin{bmatrix} 1 & 0 & -\alpha & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} - \alpha a_{13} & a_{12} & a_{13} & a_{14} \\ a_{21} - \alpha a_{23} & a_{22} & a_{23} & a_{24} \\ a_{31} - \alpha a_{33} & a_{32} & a_{33} & a_{34} \\ a_{41} - \alpha a_{33} & a_{42} & a_{43} & a_{44} \\ a_{51} - \alpha a_{43} & a_{52} & a_{53} & a_{54} \end{bmatrix}$$

Multiplication of two subtraction matrices, say S_{ij,α_1} and S_{ik,α_2} yields a matrix with 1*s* along the diagonal, α_1 and α_2 at locations S_{ij} and S_{ik} respectively, and 0s elsewhere. The operation of subtracting different multiples of a column of a matrix from other columns, which can be accomplished by successive multiplication by different subtraction matrices, can be achieved by a multiplication with a *composite subtraction matrix* constructed as described above.

Transposition matrices and subtraction matrices are examples of a more general class of matrices called *regular unimodular matrices.* A regular unimodular matrix is a square matrix whose determinant has the value $(+1)$ or (-1). We state some of the properties of regular unimodular matrices that are relevant for our purposes:

- the product of two regular unimodular matrices is a regular unimodular matrix.
- the inverse of a regular unimodular matrix formed of all integers always exists and is a regular unimodular matrix.

5.1.2 Smith Normal Form

In this section we define a matrix form called the *Smith normal form* [Smi61] and how a matrix can be transformed to Smith normal form using operations defined by transposition and subtraction matrices. The theory of transforming matrices to Smith normal form and solving systems of equations in integers using it can be found in [KHL77, GN72]. Here we present an algorithmic description of the method and argue that it is a suitable first step for solving the systems of equations we expect to encounter.

The Smith normal form of a matrix $[A]_{m\times n}$ of rank r is a matrix D of the following form:

$$[D]_{m\times n} = \begin{bmatrix} d_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & d_r & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix}$$

where $d_k \neq 0,\ k = 1, 2, \cdots r$.

Theorem 5.3 For every matrix $[A]_{m\times n}$, there exist regular unimodular matrices $[U]_{m\times m}$ and $[V]_{n\times n}$ such that

$$[U]_{m\times m}.[A]_{m\times n}.[V]_{n\times n} = [D]_{m\times n}$$

where $[D]_{m\times n}$ is in Smith normal form

We present a procedure to compute the Smith normal form of a matrix and the corresponding unimodular multiplier matrices.

algorithm

Input: Matrix $[A]_{m\times n}$

Output: Matrices $[D]_{m\times n}$, $[U]_{m\times m}$ and $[V]_{n\times n}$ such that

$$[U]_{m\times m}.[A]_{m\times n}.[V]_{n\times n} = [D]_{m\times n}$$

where U and V are regular unimodular matrices and D is in Smith normal form.

Initialization: $D = A$, U and V are unit square matrices of dimensions m and n respectively.

Method: We use the procedure FindSmith stated in Figure 5.1, which converts a matrix of the form D_r to D_{r+1} shown below in each step.

$$D_r = \begin{bmatrix} \begin{matrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_r \end{matrix} & [0]_{r\times n-r} \\ [0]_{m-r\times r} & [D'_r]_{m-r\times n-r} \end{bmatrix}$$

$$D_{r+1} = \begin{bmatrix} \begin{matrix} d_1 & 0 & \cdots & 0 & 0 \\ 0 & d_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & d_r & 0 \\ 0 & 0 & \cdots & 0 & d_{r+1} \end{matrix} & [0]_{r+1\times n-r-1} \\ [0]_{m-r-1\times r+1} & [D'_{r+1}]_{m-r-1\times n-r-1} \end{bmatrix}$$

The procedure stops when D is transformed to a Smith normal form.

Complexity: We analyze the complexity of procedure FindSmith shown in Figure 5.1. Steps 1 and 2 search a submatrix of $[D]_{m\times n}$ in *O(mn)* time. Step 3 interchanges one pair of rows and one pair of columns and multiplies corresponding permutation matrices to unimodular matrices U and V, which is again equivalent to a row pair and

```
procedure FindSmith(D)

$r = 1$
$U$ and $V$ are unit matrices of rank $m$ and $n$ respectively.
$size(D)$ is the minimum of numbers of rows and number of columns in $D$.
We refer to the submatrix of $D$ formed of elements with row and column number
greater than $r$ as $D'_r$.
 1. If $r > size(D)$ or the matrix $D'_r$ has all 0s $STOP$.
    /* D is already in Smith normal form */
 2. Determine $minrow$, $mincol$ and $minval$ - the location and value of the smallest
    nonzero number in $D'_r$.
 3. Interchange row $r$ with $minrow$ and column $r$ with $mincol$.
    /* Now location $D_{rr}$ has value $minval$. */
    Premultiply $V$ and postmultiply $U$ with permutation matrices that reflect this
    transformation.
    $V = P_{minrow,r}.V$ and $U = U.P_{mincol,r}$
 4. Reduce row and column $r$ with appropriate subtraction matrices:
    For $i = r + 1, n$:
    $\alpha_i = D_{ri}\, \mathtt{div}\, minval$
    $D_{ri} = D_{ri}\, \mathtt{mod}\, minval$
    The relevant composite subtraction matrix $S$ is an $n \times n$ unit matrix except that
    $\alpha_{r+1}, \alpha_{r+2}, \cdots \alpha_n$ are the entries in row $r$, columns $r + 1$ through $m$, respectively.
    (see earlier discussion).
    $V = S.V$
    Repeat the subtraction procedure for column $r$ and postmultiply $U$ accordingly.
 5. If row and column $r$ of $D$ contain all 0s, $r = r + 1$ $GOTO$ step 2. Else $GOTO$
    step 1.
end of procedure
```

Figure 5.1 Conversion of a Matrix to Smith normal form

a column pair interchange. The time taken is $O(m + n)$. Step 4 performs addition of a multiple of a column to the remaining columns in a submatrix of D and a similar operation with rows of D. A similar operation is performed on unimodular matrices U and V, when they are multiplied by the subtraction matrix computed above. All of these operations take $O(mn)$ time. Thus the overall complexity of one execution of steps 1-4 is $O(mn)$.

In step 5, the control goes back to step 2 if row and column r do not have all 0s, except in location D_{rr}. If D_{rr} became 1, the above condition must be met in the next step. Also D_{rr} is reduced in every step, in a manner similar to computing *gcd* of a set of numbers and is expected to reach 1 in $log_2(value)$ steps, where *value* is the smallest element in the submatrix D'_r at the beginning of this reduction step. Finally, the loop from 1-5 is executed R times, where R is the rank of the original matrix A.

Thus the complexity of computing the Smith normal form and the corresponding unimodular matrices for matrix $[A]_{m\times n}$ of rank R is $O(mnR\log(value))$ where *value* can be approximated by the median of non-zero values in matrix A. Thus if *size* is the higher dimension of A, the complexity of the operation is bounded by $O((size)^3\log(value))$.

end of algorithm

Systems of equations in integers can also be solved by transforming a matrix to a triangular matrix form in which only row or column operations are used in the transformation procedure. However, in those methods a new lowest element in pivot position($[D]_{rr}$ in earlier discussion) can be chosen only from a particular row or column. In computing Smith normal form, the new pivot can be chosen from anywhere in the unprocessed section of the matrix. Hence we expect this procedure to converge faster.

5.1.3 Solving Equation Systems

We show how transformation to Smith normal form is used to solve a system of equations in integers.

Let

$$[A]_{m\times n}.[x]_{n\times 1} = [b]_{m\times 1} \tag{5.1}$$

be a system of linear equations for which we need to determine the integer solution set.

Let

$$[D]_{m\times n} = [U]_{m\times m}.[A]_{m\times n}.[V]_{m\times n} \tag{5.2}$$

where D is in Smith normal form and U and V are unimodular matrices.

We rewrite from equation 5.1

$$[U]_{m\times m}.[A]_{m\times n}.[V]_{n\times n}.[V^{-1}]_{n\times n}.[x]_{n\times 1} = [U]_{m\times m}.[b]_{m\times 1} \tag{5.3}$$

$$[D]_{m\times n}.[V]^{-1}_{n\times n}.[x]_{n\times 1} = [U]_{m\times m}[b]_{m\times 1} \tag{5.4}$$

We define

$$[t]_{n\times1} = ([V^{-1}].[x])_{n\times1} \tag{5.5}$$

From equation 5.4 we get

$$[D]_{m\times n}.[t]_{n\times1} = [U]_{m\times m}[b]_{m\times1} \tag{5.6}$$

If r is the rank of matrix A, and hence of matrix D, then $[D_{m\times n}]$ is of the form $\begin{bmatrix} D_{r\times r} & 0 \\ 0 & 0 \end{bmatrix}$. We rewrite equation 5.6 as two equations representing the top r rows and bottom $m-r$ rows of the column matrices being equated.

$$[D_r^r]_{r\times r}.[t_r^r]_{r\times1} = [(([U].[b])_r^r]_{r\times1} \tag{5.7}$$

$$[0]_{(m-r)\times1} = [(([U].[b])_{m-r}^{m-r}]_{(m-r)\times1} \tag{5.8}$$

Thus, for the original system of equations to have an integer solution

$$[t_r^r]_{r\times1} = [D_r^r]_{r\times r}^{-1}[(([U].[b])_r^r]_{r\times1} \tag{5.9}$$

must be formed of integers and

$$[(([U].[b])_{m-r}^{m-r}]_{(m-r)\times1} = [0]_{(m-r)\times1} \tag{5.10}$$

If an integer solution exists, it can be represented by rewriting from equation 4.

$$[x]_{n\times1} = [V]_{n\times n}.[t]_{n\times1} \tag{5.11}$$

We demonstrate the complete procedure of solving systems of equations in integers with an example. Consider the following system of equations:

$$\begin{array}{rcrcrcrcr} 2x_1 & + & 3x_2 & + & 4x_3 & + & 5x_4 & = & 3 \\ x_1 & - & x_2 & + & x_3 & + & 2x_4 & = & 5 \\ 2x_1 & + & 0x_2 & + & 2x_3 & + & 5x_4 & = & -3 \end{array}$$

We restate the system as:

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 1 & -1 & 1 & 2 \\ 2 & 0 & 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ -3 \end{bmatrix}$$

We have

$$A = D^0 = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 1 & -1 & 1 & 2 \\ 2 & 0 & 2 & 5 \end{bmatrix} \quad U^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad V^0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We shall transform these matrices such that

$$D^k = U^k.A.V^k$$

remains an invariant.

First we interchange rows 1 and 2 of D to obtain the smallest non-zero element of the matrix in the top left corner. In the above equation we premultiply D and U by permutation matrix P_{12}. We have

$$D^1 = \begin{bmatrix} 1 & -1 & 1 & 2 \\ 2 & 3 & 4 & 5 \\ 2 & 0 & 2 & 5 \end{bmatrix} \quad U^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad V^1 = V^0$$

To get the smallest possible values in 1st column of D, we premultiply by composite subtraction matrix $S_{21,2}.S_{31,2}$. For the same effect on first row, we postmultiply by $S_{12,-1}.S_{13,1}.S_{14,2}$ We have

$$S^1_{PRE} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \quad S^1_{POST} = \begin{bmatrix} 1 & 1 & -1 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After corresponding multiplications we get

$$D^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 5 & 2 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \quad U^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & -2 & 1 \end{bmatrix} \quad V^2 = \begin{bmatrix} 1 & 1 & -1 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Again to move the lowest nonzero element of D to D_{22} we interchange columns 2 and 4 of D. We postmultiply D by permutation matrix P_{24}. We get

$$D^3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 5 \\ 0 & 1 & 0 & 2 \end{bmatrix} \quad U^3 = U^2 \quad V^3 = \begin{bmatrix} 1 & -2 & -1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

To reduce the row and column 2 of matrix D, we premultiply by subtraction matrix $S_{32,1}$ and postmultiply by composite subtraction matrix $S_{23,2}.S_{24,5}$ and obtain

$$D^4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -3 \end{bmatrix} \quad U^4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ -1 & 0 & 1 \end{bmatrix} \quad V^4 = \begin{bmatrix} 1 & -2 & 3 & 11 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & -2 & -5 \end{bmatrix}$$

Now the lowest nonzero element in the part of D not in Smith normal form is already in the desired position. To reduce the remaining one element, we postmultiply by subtraction matrix $S_{34,1}$ and get:

$$D^5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -1 \end{bmatrix} \quad U^5 = U^4 \quad V^5 = \begin{bmatrix} 1 & -2 & 3 & 8 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & -2 & -3 \end{bmatrix}$$

We interchange columns 3 and 4 to get the smallest element in the desired position by postmultiplying by permutation matrix P_{34} and get:

$$D^6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2 \end{bmatrix} \quad U^6 = U^5 \quad V^6 = \begin{bmatrix} 1 & -2 & 8 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & -3 & -2 \end{bmatrix}$$

Postmultiplying by subtraction matrix $S_{34,2}$ brings D in Smith normal form and we get the final forms:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad U = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ -1 & 0 & 1 \end{bmatrix} \quad V = \begin{bmatrix} 1 & -2 & 8 & -13 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & -1 & 3 \\ 0 & 1 & -3 & 4 \end{bmatrix}$$

And we verify that

$$D = U.A.V$$

The top left square submatrix of D with number of rows equal to rank r of D is the following matrix:

$$[D_{rr}]_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Its inverse can be computed easily since it is a diagonal matrix.

$$[D_{rr}]^{-1}_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

From equation 5.9 we have

$$\begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ -1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \\ -3 \end{bmatrix} = \begin{bmatrix} 5 \\ -7 \\ 6 \end{bmatrix}$$

Since it evaluates to an integer column and equation 5.10 is trivially satisfied, the system must have integer solutions. The solutions of the system can be represented using equation 5.11

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 8 & -13 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & -1 & 3 \\ 0 & 1 & -3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ -7 \\ 6 \\ t_4 \end{bmatrix} = \begin{bmatrix} 67 - 13t_4 \\ 6 - 2t_4 \\ -6 + 3t_4 \\ -25 + 4t_4 \end{bmatrix}$$

We rearrange the solution and replace t_4 by t since there is only one parameter.

$$\begin{aligned} x_1 &= -13t + 67 \\ x_2 &= -2t + 6 \\ x_3 &= 3t - 6 \\ x_4 &= 4t - 25 \end{aligned}$$

5.2 Finding Solutions in a Feasible Region

We first summarize the results of the solution procedure of the last section, in the context of the overall problem. We started with a system of m equations (and say r independent equations) in n variables. If r is greater than or equal to n, any integer solution that exists must be unique, and we would have already determined whether

a non-negative integer solution exists. However, if r is less than n, then we have all possible solutions for the n variables parameterized by $n-r$ new variables which can take any integer value. We need to determine whether there are any non-negative integer solutions. For the example in the last section, we need to determine if the system:

$$\begin{array}{rcrcl} -13t & + & 67 & \geq & 0 \\ -2t & + & 6 & \geq & 0 \\ 3t & - & 6 & \geq & 0 \\ 4t & - & 25 & \geq & 0 \end{array}$$

is feasible for some integer value of t.

This inequality system is particularly easy to solve since there is only one variable involved. The above system simplifies to:

$$\begin{array}{rcll} t & \leq & 67/13 & \quad (1) \\ t & \leq & 3 & \quad (2) \\ t & \geq & 2 & \quad (3) \\ t & \geq & 25/4 & \quad (4) \end{array}$$

The system is infeasible since (2) and (4) cannot be simultaneously satisfied.

5.2.1 Fourier-Motzkin Elimination Method

The number of variables in a system of inequalities can be successively reduced by using Fourier-Motzkin elimination method. Any system of inequalities can then be solved by successive reduction and back substitution. In this method, a variable is eliminated by creating a new inequality for each pair of inequalities in which the variable being eliminated has a different sign. We will illustrate this method with an example. A complete treatment can be found in [Duf74]. Consider the following set of inequalities:

$$\begin{array}{rcll} 0x - y + 6 & \geq & 0 & \quad (1) \\ x + 2y - 6 & \geq & 0 & \quad (2) \\ -2x - 3y + 7 & \geq & 0 & \quad (3) \\ -4x - 5y + 40 & \geq & 0 & \quad (4) \end{array}$$

Suppose we decide to eliminate x from this system of inequalities[1]. x has a positive sign in (2) above and a negative sign in (3) and (4) above. By combining inequation pairs (2)-(3) and (2)-(4) to eliminate x, we obtain the following system:

$$\begin{aligned} -y+6 &\geq 0 && (1)\\ y-5 &\geq 0 \quad \mathit{from}\,(2)\,\mathit{and}\,(3) && (2)\\ 3y+16 &\geq 0 \quad \mathit{from}\,(2)\,\mathit{and}\,(4) && (3) \end{aligned}$$

This simplifies to:

$$5 \leq y \leq 6$$

Thus a solution set to the inequalities does exist. We substitute the maximum or minimum possible value of y back in the original equations so as to obtain the range of values x can take. In the form in which our equations are, we substitute such that the y term has the minimum possible value. We get :

$$\begin{aligned} x+4 &\geq 0 && (1)\\ -2x-11 &\geq 0 && (2)\\ -4x+10 &\geq 0 && (3) \end{aligned}$$

These are equivalent to the following inequality:

$$-4 \leq x \leq -5/2$$

Solving systems of inequalities by elimination tells us whether a real solution exists to the system of inequalities. In case such a solution does exist, we can determine the range of the values that the variables may have in the solution space. It still needs to be determined whether an integer solution to the system exists.

Solving a system of inequalities by Fourier-Motzkin elimination techniques can potentially increase the number of inequalities from n to $(n/2)^2$ in every stage. Duffin [Duf74] discusses ways to lower the computation effort needed to reach a solution. They include rules for selection of the variable to be eliminated at each stage, and identification and deletion of redundant inequalities. He further argues that with these modifications, Fourier analysis is a practical method to solve systems of inequations. The number of variables and inequations that we expect in our problems is small and we expect the analysis to be fast and take only a small number of operations.

[1] Heuristics for choosing a variable so as to minimize the work are discussed in [Duf74].

5.3 Identifying Integer Solutions

The system of inequalities obtained by solving diophantine equations defines a convex region in n dimensional space where n is the number of variables in the parametric equations. We are interested in determining the feasibility of an integer solution, if the Fourier analysis shows that the solution space is not empty. In Fourier analysis, we also determine the bounds in each dimension for the convex region defined. Furthermore these bounds are *tight* in the sense that at least one vertex of the convex region must be on each bounding hyperplane.

We define some terms here that we shall use in the rest of this section. An $n - rectangle$ is a closed convex region in n dimensions bounded by hyperplanes that are parallel to the axis hyperplanes (that is, defined by $X = constant$ for some co-ordinate axis X). An n-rectangle *tightly*, or *strictly* bounds a convex region if the the convex region is completely contained in the n-rectangle, and every hyperplane defining the n-rectangle intersects with the convex region. The *centroid* of an n-rectangle is the point with co-ordinates equal to the midpoint of the range of the n-rectangle along each axis. If $X_1, X_2, ...X_n$ are the co-ordinate axes and $(\alpha_1, \alpha_2, ...\alpha_n)$ and $(\beta_1, \beta_2, ...\beta_n)$ are two points in n space, a *line* in n dimensions between them is defined by:

$$\frac{x_1 - \alpha_1}{\beta_1 - \alpha_1} = \frac{x_2 - \alpha_2}{\beta_2 - \alpha_2} = \cdots \frac{x_n - \alpha_n}{\beta_n - \alpha_n}$$

5.3.1 The Case of Two Dimensions

If the system of inequalities obtained has only one parametric variable, the system can be trivially solved as shown earlier. We first describe a solution procedure when there are two parametric variables. Subsequently we will describe the solution method for three or more variables by solving a set of two variable problems. We expect that most practical situations would yield a system in two or fewer variables.

A system of inequalities in two dimensions describes a convex region in two dimensions. For the present we assume that the region is finite. Fourier analysis discussed in the last section would yield a tight bounding rectangle for the convex region. The bounding rectangle is known to contain integer points and we want to determine if the convex region itself has any integer points.

The following theorem holds for convex regions in two dimensions:

Theorem 5.4 If a finite convex region in two dimensions is tightly bounded by a rectangle, then the centroid of the rectangle is always a part of the convex region.

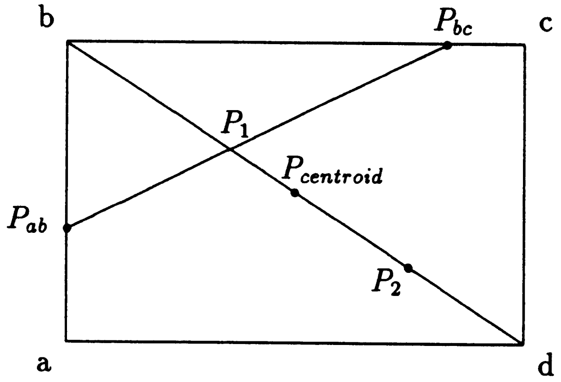


Figure 5.2

Proof:

Let C be a convex region in two dimensions and let R be the rectangle that bounds it tightly. Let *abcd* be the vertices of the rectangle R as shown in Figure 5.2. Let $P_{centroid}$ be the centroid of R, which is also the point of intersection of the diagonals *ac* and *bd*. Since R bounds C tightly, each of the sides *ab*, *bc*, *cd* and *da* must contain at least one point that belongs to C. Let P_{ab} and P_{bc} be points on the sides *ab* and *bc* respectively, that belong to C. Since C is convex, the line segment $P_{ab}P_{bc}$ is contained in C. Also, the line segment $P_{ab}P_{bc}$ is inside the triangle *abc* and must intersect the diagonal *bd* at a point between *b* and $P_{centroid}$ (which is on the diagonal *ac*) , say P_1. Thus there is one point on the diagonal *bd* between *b* and $P_{centroid}$ that is also in C. Similarly we can show that there is one point on the diagonal *bd* between *d* and $P_{centroid}$, say P_2, that is also in C. Therefore, since C is convex, $P_{centroid}$ is part of C.
End of Proof.

This theorem suggests that the neighborhood of the centroid is a good place to look for possible integer solutions. We use this fact in the algorithm presented later

in this section. Furthermore, if none of the closest integer point neighbors of the centroid are in the convex region, the convex region can only be of a special shape. Specifically we have the following results:

Lemma 5.5 Let R (vertices $r_1r_2r_3r_4$) and R' (vertices $r'_1r'_2r'_3r'_4$) be rectangles with parallel sides such that R' is enclosed inside R. Let p be a point inside R'. A set of line segments that connect a point on each side of R to p must intersect at least two sides of R'.

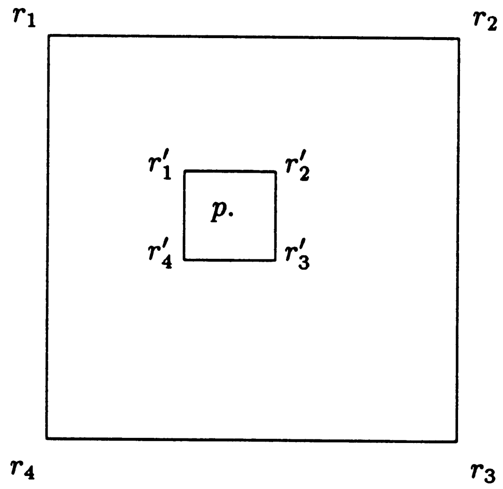


Figure 5.3

Proof:

We will use Figure 5.3 for illustration. Any line segment connecting p to a point on a side of R must intersect at least one side of the rectangle R' since all of R is entirely outside R' and p is inside R'. Suppose $r'_4r'_3$ is the only side of R' that the set of line segments connecting p to each side of R intersect. Now p is a point between parallel lines passing through r_1r_2 and $r'_4r'_3$ and a line segment connecting a point on r_1r_2 to p cannot intersect the line through $r'_4r'_3$. Thus $r'_4r'_3$ cannot be the only side of R' that the set of line segments connecting p to each side of R intersect. Hence, there must be at least two such sides, and that proves the lemma.

End of Proof.

Theorem 5.6 Let C be a convex region in two dimensions. Let R be the smallest rectangle with sides parallel to the axes that encloses C. Assume that the centroid of R is not an integer point and let R' be the smallest rectangle(square) with integer vertices that includes the centroid of R. If none of vertices of R' belong to C, and R is more than 4 integer units in length and width, then C intersects exactly two sides of R'.

Proof:

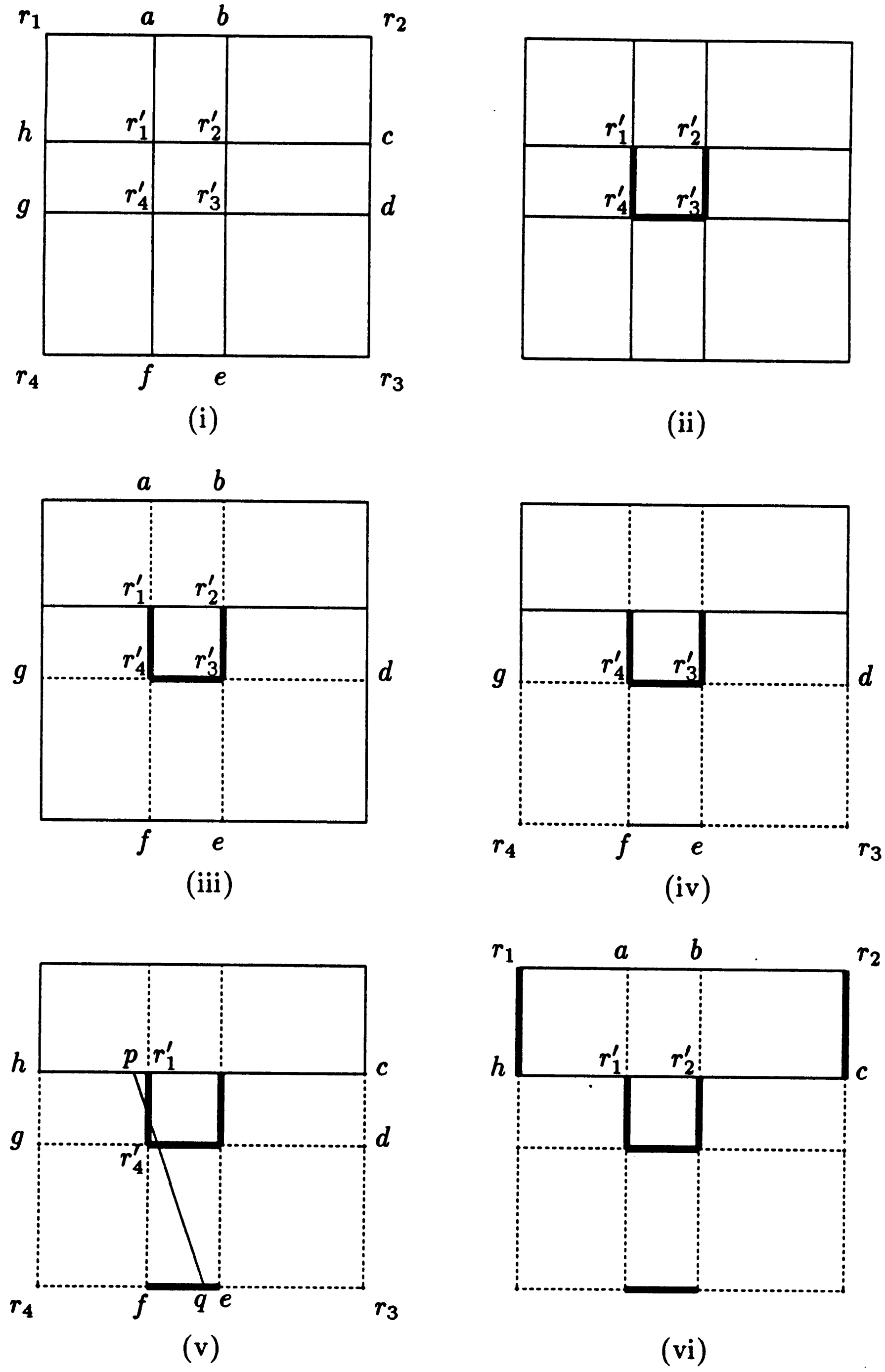
Let R be the rectangle $r_1r_2r_3r_4$ in Figure 5.4(i) that tightly bounds a convex region C (not shown in the figure). R' is the rectangle shown as $r'_1r'_2r'_3r'_4$ and the centroid of R is an interior point of R'. The sides of R' are extrapolated until they meet a side of R.

Since R tighly bounds C, there is at least one point belonging to C on each side of rectangle R. Also there is at least one point belonging to C inside R' (centroid of R). Since C is convex, there is a line segment from a point on each side of R to a point inside R' which is completely contained in R. From Lemma 5.5, C must intersect at least two sides of R'.

We now prove that C can intersect at most two sides of R'. Suppose C intersects three sides of R'. Without loss of generality we assume the sides are $r'_2r'_3$, $r'_3r'_4$ and $r'_4r'_1$. We represent this by drawing these sides with thick lines in Figure 5.4(ii). In Figure 5.4 we will convert every line segment that we prove has a point belonging to C, to thick lines. Every line segment which is proved *not* to contain any points in C is marked by dashed lines.

We have assumed that there is at least one point on the line segment $r'_4r'_1$ that belongs to C. If any point on the line segment r'_1a belongs to C, then the point r'_1 must belong to C, since C is convex. But r'_1 is a vertex of R' and does not belong to C. Thus we conclude that the line segment r'_1a does not intersect C. Similarly we can show that line segments r'_1a, r'_4f, r'_2b, r'_3e, r'_4g and r'_3d cannot intersect C. These inferences are shown in Figure 5.4(iii).

If there is any point of C inside the rectangle gr'_4fr_4, then a line segment from that point to a point inside R' must be contained in C. But such a line segment must intersect gr'_4 or r'_4f, and we have established that these line segments do not intersect C. Thus there can be no points inside the rectangle gr'_4fr_4 that belong to C. In particular, line segments fr_4 and r_4g do not intersect C. Similarly we can show that line segments dr_3 and r_3e do not contain any points belonging to C. This is shown in Figure 5.4(iv).



(i) (ii) (iii) (iv) (v) (vi)

Figure 5.4 Intersection of a convex region and its enclosing rectangle

Since R tightly bounds C, there must be at least one point on r_3r_4 that is in C. Since we have proved that line segments r_3e and fr_4 do not contain any points belonging to C, the line segment ef must intersect C, as shown in Figure 5.4(v).

Now since the sides of R are more than 4 integer units (or 4 times the sides of R'), line segments gr'_4, hr'_1, r'_4f, r'_4e are all longer than one integer unit (or longer than sides of R'). Also sides of R' and line segments ef and gh are of unit length.

It can be seen that any line through a point on the line segment ef that does not intersect the line segment fr'_4 cannot intersect the line through points h and c at a point more than 1 unit left (towards h) of point r'_1, and hence cannot intersect the line segment gh (see pq in Figure 5.4(v)). Thus there cannot be points belonging to C on both line segments ef and gh, since a line segment connecting those points would have to intersect the line segment fr'_4 and we have proved that the line segment fr'_4 does not intersect C. Since we have also proved that the line segment ef has at least one point belonging to C, the line segment gh cannot intersect C. Similarly we can show that the line segment cd cannot intersect C. Thus we reach the situation shown in Figure 5.4(v).

Now since the side r_4r_1 of rectangle R must have at least one point that is in C and the line segment r_4h has been shown to not have any points belonging to C, the line segment r_1h must intersect C. Similarly we can show that the line segment r_2c must intersect C (see Figure 5.4(vi)). Thus there must be a line segment from a point on the line segment r_1h to a point on the line segment r_2c which is contained in C. But every such line segment must intersect the line segment ar'_1 (and br'_2) which we have proved does not intersect C. Therefore we have a contradiction.

Hence the convex region C cannot intersect three or more sides of R'. Since we have already shown that it must intersect at least two sides of R', the convex region C must intersect exactly two sides of R'.

End of Proof.

Furthermore, we find that a convex region that satisfies the requirements of this theorem must be located around one of the diagonals of the bounding rectangle. Specifically we have the following theorem:

Theorem 5.7 Let C, R and R' be defined as in Theorem 5.6. Let R_1, R_2, R_3 and R_4 be the rectangles formed by the extensions of the sides of R' and the rectangle R as shown in Figure 5.5. Then under the conditions of Theorem 5.6, exactly two diagonally opposite rectangles R_i intersect with C.

Proof:

We first show that at least two of the rectangles R_i intersect C. Suppose none of the rectangles R_i intersect C. Since R strictly bounds C, all sides of R intersect C.

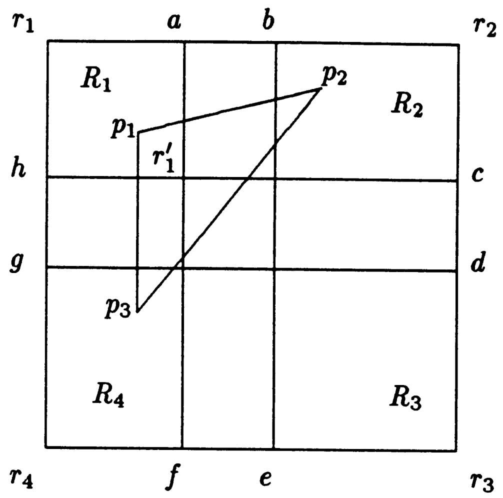


Figure 5.5

Thus C must intersect each of the segments ab, cd, ef and gh, since those are the only parts of the sides of R that do not belong to any of R_i. But a line segment connecting any point on ab to any point on bc must intersect R_1, which contradicts that none of the rectangles R_i intersects C. Hence there must be at least one rectangle R_i that intersects C.

Suppose R_1 is the only rectangle R_i that intersects C. Then C must intersect the line segments cd and ef since those are the only parts of the sides r_2r_3 and r_3r_4 respectively that do not belong to any of the R_i assumed not to intersect C. But a line segment connecting any point on cd to any point on ef must intersect the rectangle R_3, which contradicts that R_1 is the only rectangle that intersects C. We conclude that at least two of the rectangles R_i must intersect C.

We now show that not more than two of the rectangles R_i can intersect C. Suppose at least three of the rectangles R_i intersect C. Let the rectangles be R_1, R_2 and R_3. Let p_1, p_2 and p_3 be points belonging to C in the rectangles R_1, R_2 and R_3 respectively. It is easy to see from Figure 5.5 that point r_1' is inside the triangle $p_1p_2p_3$, and hence inside C which is a contradiction. Hence we can have at most two rectangles R_i that intersect C. We conclude that exactly two of the rectangles R_i intersect C.

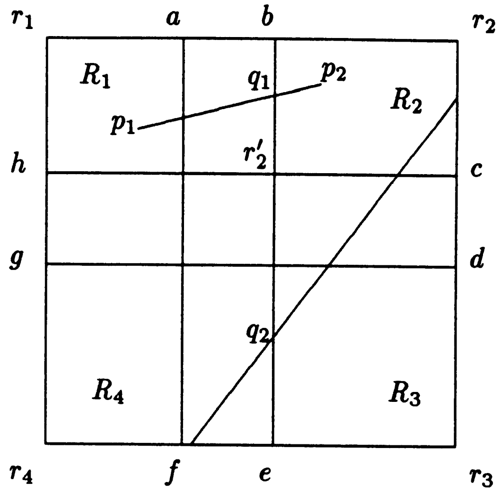


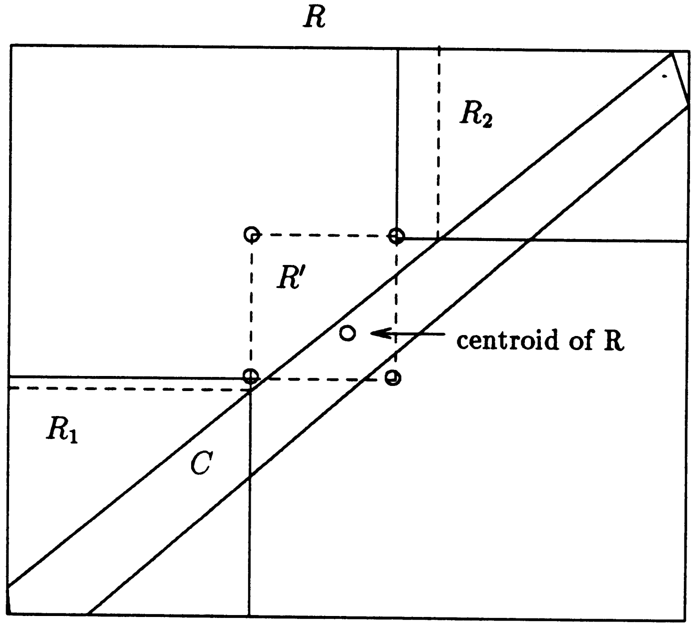
Figure 5.6

We now show that the two rectangles R_i that intersect C, must be diagonally opposite. Suppose two adjacent rectangles R_1 and R_2 intersect C. Then R_3 and R_4 cannot intersect C, since exactly two of R_i intersect C. The line segment ef must intersect C since that is the only part of the side r_3r_4 that is not in the rectangles R_3 or R_4. Let p_1 and p_2 be points in the rectangles R_1 and R_2 respectively that belong to C, as shown in the Figure 5.6. A line connecting p_1 and p_2 will intersect the segment br'_2. Let the point of intersection be point q_1. Also, since C is convex and tightly bounded by R, there must be a line segment contained in C that connects a point on the side r_2r_3 to a point on the line segment fe. Since the line segment ab is shorter than the line segment br_2, and the line segment er'_2 is longer than the line segment r'_2b, any line segment connecting a point on the side r_2r_3 to a point on the line segment ef must intersect line segment r'_2e, say at point q_2. Since q_1 and q_2 belong to C and r'_2 lies on the line connecting them, r'_2 must belong to C. This is a contradiction to the conditions of this theorem. Hence two adjacent R_i cannot intersect C.

Hence, exactly two diagonally opposite R_i must intersect C.

End of Proof.

We conclude that if a convex region (larger than a certain size) contains an integer point, then either an integer point close to the centroid of its bounding rectangle is inside the convex region, or the convex region region can be divided into two smaller regions, at least one of which must contain an integer point. Based on these results, we have the following procedure to determine if a given convex region contains any integer points.



C : Convex region under consideration.

R : Smallest rectangle enclosing C

R' : Smallest integer square enclosing centroid of R

R_1, R_2 : Bounding rectangles for the new problem

Figure 5.7 Nature of Convex Regions with no points in the vicinity of the Centroid of the Bounding Rectangle

procedure: FindIntegers(C, R)

input: Convex region C defined by inequalities in two variables, say x and y. Rectangle R tightly bounding C.

output: Boolean value representing whether an integer solution exists in the region.

1. If R covers more than 4 integers in both dimensions, GOTO step 2. Otherwise, without loss of generality, say x is the variable which can take at most 4 integer values. For each possible integer value of x, determine the range of values y can take. If an integer is found in the value range of y, return TRUE, else return FALSE. This is an exhaustive search in a small space.
2. Find the centroid of R. If it is an integer point, return TRUE. Let R' be the smallest square with integer points as corners enclosing the centroid of R. If any of the four corner points of R' is part of C, return TRUE. Otherwise GOTO step 3.
3. Find the two sides of R' that intersect C. Determine the two rectangles R_1 and R_2 as shown in Figure 5.7. Define C_1 and C_2 by adding the new boundaries of R_1 and R_2 respectively, to C as additional inequality constraints. Find the new strictly bounding rectangles R'_1 and R'_2 for C_1 and C_2 respectively using Fourier-Motzkin elimination.

   Return (FindIntegers(C_1, R'_1) *OR* FindIntegers(C_2, R'_2)).

complexity: This procedure divides the problem into two smaller problems that are half the size. The division stops at problem size 4. Thus the maximum number of times the above steps may have to be repeated is $n/4$, where n is the length of the bounding rectangle for the original convex region C.
end of procedure

5.3.2 Solution for n Dimensions

The results obtained for 2 dimensions cannot be extended to n dimensions in general. However, intuition suggests that the centroid of the bounding rectangle is still a good position to potentially establish an integer point. We use this to divide and shrink the region for analysis, until a solution is found or the problem is reduced to 2-dimensions.
procedure: NdimFindIntegers(n, C, R)
input: A set of inequalities with n variables representing a closed convex region C. Values X_i^{max} and X_i^{min} for $i = 1$ *to* n denoting an $n-$rectangle R strictly enclosing C.
output: Boolean value representing whether an integer solution exists in the region.

1. If range in any dimension does not contain an integer, no integer solutions can exist, so terminate returning FALSE. If the product of the the ranges is a small inte-

ger, perform an exhaustive search for solutions in the integer space. If the number of dimensions is two, Return (FindIntegers(C, R)). Otherwise go to step 2.

2. Determine the centroid of the given n-rectangle R. Let R' be the smallest n-rectangle with all integer vertices that contains the centroid of R. If any of the integer vertices of R' belong to the convex region C, return TRUE, else GOTO step 3.
3. Let X_i be the dimension in which R contains the smallest number of integers. If the number of integers is greater than two, GOTO step 4. Otherwise obtain (at most) two new convex regions in $n - 1$ dimensions by giving occurrences of X_i fixed integer values within this range. Suppose C_1 and C_2 are the new regions obtained. Find the smallest enclosing n-rectangles R_1 and R_2. Return (NdimFindIntegers($n - 1, C_1, R_1$) *OR* NdimFindIntegers($n - 1, C_2, R_2$)).
4. Let x_1 and x_2 be the two values in the center of the range of values with $x_1 < x_2$ that X_i can take. Add additional constraints $X_i <= x_1$ and $X_i >= x_2$ to C to get two new convex regions C_1 and C_2 respectively. Find their smallest bounding n-rectangles R_1 and R_2. Return (NdimFindIntegers(n, C_1, R_1) *OR* NdimFindIntegers(n, C_2, R_2)).

end of procedure

The steps of these solution procedures are organized so that most common cases can be handled quickly, but the less common cases are also handled. It is not possible to do an accurate complexity analysis of this solution procedure since it is dependent on numerical values. However, we expect the procedure to be efficient for problems with very few variables and small ranges, which is what we expect in our applications.

5.3.3 Handling Infinite Regions

The solution procedures so far assumed that the ranges for all variables are finite. This may not be true in general. An infinite convex region would include zero or infinite integer points. It is obvious that an infinite convex region bounded by diverging hyperplanes would include infinite integer points. A necessary condition for an infinite convex region to have no integer points is that there are parallel hyperplanes bounding the region (with an infinite thin strip with no integer points between them). We have the following lemma:

Lemma 5.8 An infinite size convex region must have infinite integer points unless at least two of the hyperplanes defining it are parallel.

For our analysis, we check if any of the hyperplanes defining the convex region are parallel. If no two hyperplanes are parallel, no further analysis is required. If parallel hyperplanes do exist, we generate heuristic values for bounds whenever actual values are not available. A heuristic value that we use for bounds is the largest coefficient in the system of inequations. This is supported by the fact that the occurrence of integers inside infinite convex regions is usually cyclic along the axes in which it is infinite, and the lengths of cycles are determined by the magnitude of the integers in the inequation set. A more exact analysis could be performed to determine a better defined bound, but we do not have evidence that would suggest that our heuristic is inadequate.

Chapter 6

Other Synchronization Mechanisms

In this chapter we discuss the potential and the limitations of extending our methods for analyzing simple event variable synchronization to other synchronization mechanisms.

6.1 Event Synchronization with CLEAR statement

In previous chapters, we have assumed the absence of the CLEAR statement, which has the effect of resetting the value of an event variable to *clear*. The CLEAR statement is commonly used to reuse an event variable for synchronization. In this section we discuss how the CLEAR statements affect program synchronization behavior, and their relevance to our analysis.

If CLEAR statements are ignored when using the methods of the previous chapters to identify potential data races, all true data races will still be recognized. However, some of the potential data races can be proved infeasible by analyzing the use of CLEAR statements. Thus, analysis of CLEAR statements is a critical optimization to reduce the number of false data races reported.

In our synchronized control flow graph, we have an edge from every POST statement on an event variable to every WAIT statement on the same variable, implying that any of the POST statements can set the value of the event variable to *posted* and enable execution past the WAIT statement. However, if we can prove that a CLEAR statement must execute after a POST statement but before a corresponding WAIT statement (whenever both are executed), that particular POST statement cannot be responsible for triggering the execution at that WAIT statement, and hence the synchronization edge between this POST-WAIT pair can be removed. This makes the analysis using the *SCG* more accurate[1].

[1] Inside loops, analysis of CLEAR statements can often help attach more precise synchronization distance vectors, which is in effect analogous to reducing the number of POSTS that can trigger execution past a WAIT.

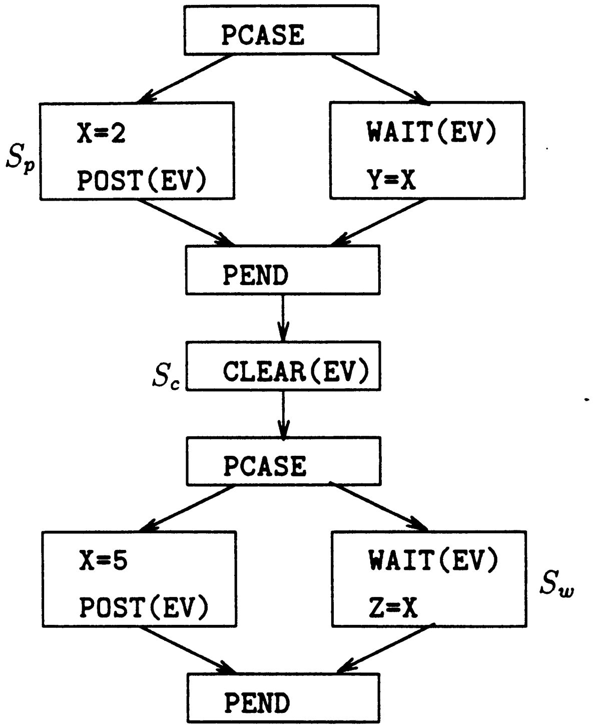


Figure 6.1 A program using a CLEAR statement. The synchronization edge from block S_p to block S_w can be eliminated due to the CLEAR statement in block S_c.

Let ev be a scalar event variable. Let S_p, S_w and S_c represent statements POST(ev), WAIT(ev) and CLEAR(ev) respectively in a program. Suppose we can prove that if S_p is executed before S_w in a program run, then S_c must execute after S_p and before S_w. If this is true, then the value of ev posted by S_p would certainly have been cleared at least once before S_w executes, and hence S_p cannot cause a thread of execution waiting at S_w to resume execution. Thus, the synchronization edge from S_p to S_w is not of significance and can be removed (see Figure 6.1). These facts can be stated formally as the following lemma, which follows from the semantics of the event variable operations:

Lemma 6.1 Let S_p, S_w and S_c be POST, WAIT and CLEAR statements acting on the same scalar event variable ev, respectively. If S_c can never execute before S_p and must execute before S_w whenever S_w executes, then there is no synchronization relationship between S_p and S_w.

Since execution orders in parallel executions are captured by preserved sets defined in Chapter 3, we have the following theorem:

Theorem 6.2 Let S_p, S_w and S_c be POST, WAIT and CLEAR statements acting on the same scalar event variable **ev**, respectively. Given that

$$p \in Preserved(c)$$
$$c \in Preserved(w)$$

There is no synchronization relationship between S_p and S_w.

Proof:

Immediate from the definition of preserved sets and Lemma 6.1.

End of Proof.

Theorem 6.2 defines a criterion for eliminating synchronization edges after preserved sets have been computed. Eliminating synchronization edges and recomputing would make the preserved sets more precise. This process can be repeated to improve the precision of preserved sets, until a fixed point is reached.

6.2 ADA Rendezvous

The methods developed in this thesis can be used to analyze statement orders in programs using rendezvous synchronization as defined in the ADA language. We first informally introduce rendezvous synchronization and then present our approach to analyzing programs using this synchronization model.

Parallel tasks synchronize by issuing *entry call* and *accept* instructions. Figure 6.2 shows two simple ADA tasks. A task T1 issuing an *entry call* to another task T2 of message type P suspends execution until task T2 executes an *accept* statement of the same message type P. Similarly a task T2 executing an *accept* statement of message type P suspends execution until some task executes an *entry call* to T2. After the *rendezvous* completes, both tasks involved can continue execution. Exactly two tasks synchronize in a rendezvous. If multiple tasks are waiting to rendezvous with the same accepting task, the semantics of ADA dictate that the system non-deterministically selects one of them.

We can construct a synchronized control flow graph for programs with rendezvous synchronization by adding a pair of synchronization edges between statements that can potentially rendezvous. Any statement that executes an *entry call* to task T of

```
Task T1
begin
  i = 1
  T2.P
  j = 2
  T2.P
end


Task T2
begin
  accept P
  x = i + 3
  accept P
  y = j + 4
end
```

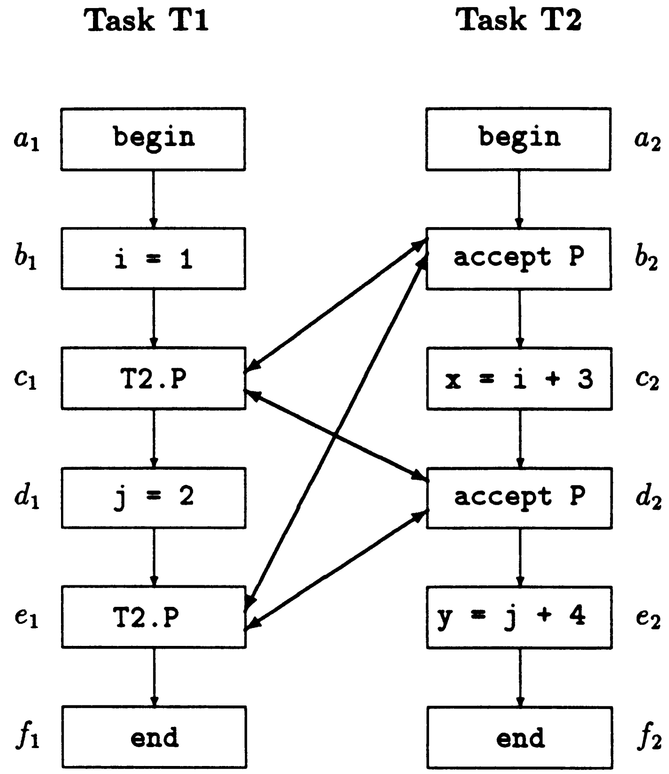


Figure 6.2 Rendezvous Synchronization in ADA

type P is assumed to potentially rendezvous with any *accept* statement in task T of type P. Having a pair of synchronization edges between two statements potentially involved in a rendezvous leads to ambiguous conclusions regarding the execution order of the specific statement pair. In particular if S_{r_1} and S_{r_2} can potentially rendezvous, we may conclude that each of them is in the preserved set of the other. However, preserved sets computed for blocks in the program that do not contain an *accept* or an *entry call* do not have any ambiguity. The information obtained can be used to identify potential data races.

In Figure 6.2, we can infer that i is defined in block b_1 before it is used in block c_1, by computing the preserved set for block c_1. However we cannot directly infer that j is defined in block d_1 before it is used in block e_2. The reason is that the pair of synchronization edges between c_1 and d_2 represent an impossible rendezvous, and the

presence of synchronization edges between them reduces the precision of our analysis. The following theorem can be used to eliminate impossible rendezvous:

Theorem 6.3 Let S_{r_1} and S_{r_2} be program blocks that are potentially involved in a rendezvous. Further, for every control flow predecessor S_{cpred} of S_{r_2} if

$$S_{r_1} \in Preserved(S_{cpred})$$

then the rendezvous between S_{r_1} and S_{r_2} is an impossible rendezvous.

Proof:

If S_{r_1} is guaranteed to complete execution before any control flow predecessor of S_{r_2} begins execution, then clearly S_{r_1} must complete execution before S_{r_2} begins execution. This implies that S_{r_1} and S_{r_2} cannot be involved in a rendezvous.

End of Proof.

We can use this theorem to eliminate the synchronization edges between nodes c_1 and d_2 (and between nodes f_1 and b_2) in Figure 6.2. Analysis after eliminating these edges would conclude that j must be defined in block d_1 before it is used in block e_2.

Note that the elimination of impossible rendezvous is achieved in linear time after the preserved sets have been computed. However eliminating some impossible rendezvous can lead to collection of more precise information, which in turn may lead to labeling of other potential rendezvous as impossible. Thus the process can be repeated to achieve better precision. We do not have sufficient knowledge of characteristics of ADA programs to know how effective this method is for real programs, and how it compares with other algorithms to eliminate impossible rendezvous [Tay83b].

6.3 Semaphores

Use of semaphores is a common synchronization mechanism in parallel and concurrent programming. A *semaphore* is an integer variable on which only the following two atomic operations are defined.

```
P(S): while S ≤ 0 do skip;
      S = S - 1

V(S):S = S + 1
```

Semaphores can be used to ensure mutual exclusion.

```
loop
   P(mutex)
     critical section
   V(mutex)
     non critical section
forever
```

If a set of processes is sharing the above code, and the semaphore **mutex** is initialized to 1, the semaphore operations ensure that only one process can execute the code in the critical section at any given time. Using semaphores for mutual exclusion is not directly relevant to our analysis methods and the discussion of other mutual exclusion mechanisms in the next section is relevant here.

Semaphores can also be used to enforce statement execution ordering. If a process p_1 has the statement sequence:

```
$S_1$
V(synch)
```

and process p_2 has statements:

```
P(synch)
$S_1$
```

S_2 will execute only after S_1 if the semaphore **synch** is initialized to 0 and no other statements act on it.

When binary semaphores are used to enforce ordering of statements, methods in this thesis can be used to analyze their effect on program execution. We shall show that a *binary* semaphore can be interpreted in terms of operations on event variables. A binary semaphore is a semaphore that can take values 0 and 1 only. For our analysis we translate the operations on a binary semaphores to event operations:

```
P(semaphore): WAIT(event)
              CLEAR(event)

V(semaphore): POST(event)
```

Although the corresponding statements are not semantically equivalent, (semaphore P operation is atomic, while the sequence of WAIT and CLEAR statement used to mimic it is not) the information collected by analyzing equivalent event operations is still correct. Thus, we can determine execution ordering enforced by semaphore operations by analyzing an equivalent program with event variables. However, when semaphores

are used as means of enforcing mutual exclusion, we cannot derive any useful information.

6.4 Locks and Critical Sections

Locks and *critical sections* are synchronization mechanisms used to ensure mutual exclusion between processes for access to shared code or data. These synchronization mechanisms do not directly enforce any ordering between statements, and thus do not cause any dependences to be preserved. The information contained in them is not of any direct significance for our analysis procedures, whose objective is to identify the program dependences that are protected by synchronization.

However, using mutual exclusion mechanisms with other synchronization mechanisms can enforce execution orders. For instance, if we can establish that any statement in a program section S_1 protected by a lock is executed before a statement in another program section S_2 protected by the same lock, we can infer that all of S_1 must be executed before any part of S_2 can begin execution. We have not investigated this effect and are not aware if it is of significance in real applications.

Analysis of mutual exclusion mechanisms can be used to help development of applications in other ways. Static analysis can reveal when accesses to a particular variable are not guaranteed to be mutually exclusive. This could result from incorrect use of locks or critical sections. Also, when concurrent accesses to a particular variable are made mutually exclusive (say by using critical sections), often the actual order of execution is not critical. A programming environment can use this information to selectively ignore dependences that are irrelevant for program analysis.

6.5 Summary

The analysis methods in this thesis were intended for use on programs with event synchronization. However, we have shown that the methods have the potential to be useful for several other models of synchronization. These methods can be used to discover useful information about programs that use synchronization to constrain possible execution orders. In addition to event synchronization, these include rendezvous and semaphore synchronization. We have not studied these synchronization mechanisms in detail, but we do present evidence that suggests the fundamental similarity in analyzing these synchronization mechanisms.

Synchronization mechanisms enforcing mutual exclusion do not have information pertaining to program dependence and are not relevant to our analysis. However, certain anomalies associated with enforcing mutual exclusion are detectable by static program analysis and should be reported.

Chapter 7

Conclusions and Future Work

7.1 Research Contribution

In this thesis we have developed a framework for analyzing synchronization in programs for a shared memory multiprocessor. We presented an algorithm to analyze "schedule-correctness" of parallel programs by stating the problem in a data flow framework. We presented another algorithm to transform the "schedule-correctness" problem to the problem of finding a non-negative integer solution to a system of equations. We have presented a practical way of solving such systems using existing known mathematical methods, and heuristic techniques developed in the thesis. We have characterized the types of synchronization mechanisms that can be analyzed by our methods.

7.2 Significance of Research

The immediate problem that we have solved is finding the set of potential anomalies in a parallel program by identifying data dependences that can lead to data races. If no such anomalies are found, the program is "schedule-correct". We consider this information to be of significant importance to a programmer developing a parallel program, or a programmer trying to convert an existing sequential program to parallel form. The set of potential anomalies is also a starting point for a debugger for parallel programs. If a set of potential anomalies can be established statically, a debugger to detect parallel access anomalies needs to trace references only to relevant memory locations, and not to all of the shared memory.

The algorithms developed in this thesis have other applications in a parallel programming environment. Optimization and transformation of explicitly parallel programs requires that the programming environment understand the meaning of synchronization in parallel programs. Mechanisms to automatically insert necessary

synchronization, and to remove redundant synchronization statements, can use the analysis presented in this thesis as their basis.

We consider this thesis to be a significant step towards increasing the power and flexibility of parallel programming environments.

7.3 Implementation

We have developed a prototype implementation of synchronization analysis in the context of the ParaScope programming environment being developed at Rice University. In ParaScope a programmer can examine the data dependences which may cause potential data races if a program loop executes in parallel. This analysis is sharpened with synchronization analysis. In particular, a programmer can find which data dependences are protected by the synchronization in the program, and hence cannot lead to data races. Figure 7.1 and Figure 7.2 give an illustration of the implementation of our synchronization analysis. The interface is that of the ParaScope editor.

The implementation has several limitations, partly because of the environment in which it is developed. Since ParaScope computes dependences only for individual loop nests (and not across loop nests), synchronization analysis can be applied only to statements inside the same loop nest. Synchronization and dependence distance vectors are assumed to have all simple constant components, although it is possible to extend the analysis to some other cases. Finally the analysis is not integrated with other transformations and is only used to present advisory information on which dependences are protected.

Our implementation has convinced us that synchronization analysis is extremely useful in developing parallel programs. Figure 7.2 demonstrates that the effect of even relatively simple use of synchronization can be tricky to understand. However we have not done extensive experimentation. It remains to be established whether the payoffs of static analysis methods are worth the cost of the analysis. Some of the methods developed in this thesis are polynomial approximations to NP-hard problems. We believe that our methods would be sufficient to handle practically all the cases in real programs, but we have only limited experimental evidence to support this claim. Also, it is not known if simpler and less accurate methods would do as good a job in practical situations. All these issues can be resolved only by collecting empirical results by extensive experimentation.

7.4 Synchronization in Parallel Programs

We made an effort to understand what synchronization patterns are common in real programs and whether they are analyzable by the methods presented in this thesis. Many real programs could not be run through our prototype analysis tool because of the limitations of the tool or the syntactic differences between parallel Fortran dialects. However our main objective was to verify the applicability of our methods in broad terms, so we manually analyzed codes whenever necessary. The discussion in this section is based on about 10 benchmark programs obtained from Los Alamos National Labs, other literature on parallel program analysis (particularly [Hen87]) and conversations with several scientists involved in development of parallel scientific applications.

The first observation was that the use of event synchronization is not very common even when it is supported in a language. The main reason was that most of the parallelism in user applications was exploited in a straightforward way using *parallel case*, *parallel loop* or analogous constructs. The other reason was the programmer's reluctance to use event variables. Often programmers would use locks and additional variables to simulate operations on events, even though the language directly supported event operations. We discovered that programmers often found event variable operations complex and hard to use. Another reason for avoiding event synchronization was the apparent or real belief that it is an inefficient method of synchronization.

We believe that use of event operations is a natural way to implement synchronization and lack of widespread use just reflects the infancy of parallel computing. Event operations can be used to implement software pipelining which is an important source of additional parallelism. There is no reason why use of event operations should be less efficient than using alternate mechanisms to do the same work, although specific implementations may lead to such behavior.

The use of event synchronization in programs examined was always very simple. Analysis of these programs leads to fairly simple synchronized control flow graphs. However, often the complete meaning of a program can be understood only with some symbolic analysis. For instance, in one program that we analyzed, a set of tasks would start executing concurrently and each of them would increment a counter before terminating. There was code inside the tasks to ensure that the last task to terminate would post an event variable, effectively implementing a barrier. Clearly the complete meaning of this construction cannot be inferred without symbolic analysis. In other

instances we observed that locks were used to implement event style synchronization, again requiring symbolic execution to make meaningful inferences.

Overall we feel that the methods in this thesis are sufficient to solve the problem statement that we presented, which is to analyze parallelism and synchronization using only the synchronized control flow graph. However, the solution of this problem may not be sufficient for us to verify accurately whether a programs execution is guaranteed to be schedule-correct and to identify the potential data races. Additional symbolic analysis can be important to solve the overall problem.

7.5 Future Work

The analysis procedures developed in this thesis can be extended and applied to several related problems.

7.5.1 Other Synchronization Mechanisms

This thesis develops methods to analyze *event* style synchronization. In Chapter 6 we discuss how we can do synchronization analysis for ADA rendezvous, semaphores, and other synchronization mechanisms. The discussion is preliminary and a more thorough analysis is needed. Our analysis is oriented towards numerical applications where parallelism is essentially a performance issue and its applicability to environments that are oriented towards inherently parallel or realtime applications is not known.

7.5.2 Removal of Redundant Synchronization

In Chapter 7 we presented how synchronization edges in a program representation can be removed when they are not meaningful due to the presence of CLEAR statements. Synchronization statements are redundant when the execution ordering that they enforce is already enforced by other synchronization statements. By removing a synchronization edge, and analyzing the program to determine if the source of the edge is guaranteed to execute before the sink, we can detect redundant synchronization statements. However, repeating this analysis for each synchronization edge can be overly expensive. Also the order in which redundant synchronization edges are removed may be critical to the overall effectiveness of this procedure. It would be useful to have a general purpose algorithm to minimize synchronization in a whole program.

7.5.3 Insertion of Synchronization

A programmer would like to have just the right amount of synchronization in programs. It should be enough to ensure correct execution, but with minimal runtime overhead. Optimal synchronization is a function of machine parameters, particularly the timings of various machine operations. However, in many cases it can be statically determined which of the different possible correct synchronization structures will result in the smallest execution overhead. A parallel programming environment should be able to advise and help the programmer in managing synchronization.

The simplest way to ensure that a program is schedule-correct (relative to sequential execution) is to synchronize each data dependence. This can be achieved by inserting a `POST` statement after the source of the dependence and a corresponding `WAIT` statement before the sink of the dependence. This would ensure maximal parallelism (under the assumption that each data dependence must be preserved) but may have a high and unreasonable synchronization overhead. Other strategies may yield low synchronization overhead, but with some loss of parallelism. Midkiff and Padua [MP86] suggest a way of synchronizing DO loops. Wolfe [Wol87] discusses various strategies for inserting synchronization for a variety of synchronization constructs. These are mainly based on inserting sufficient synchronization and then removing synchronization that is considered unnecessary by examining the loops for some particular synchronization pattern. Li and Abu-Sufah [LAs85] present a similar approach for synchronization by locks.

These methods to insert synchronization look for specific patterns for optimizing synchronization. The results of this thesis can be used to develop a general framework in which the effect of adding and deleting synchronization can be examined in the context of correctness and efficiency. It is desirable to have a general mechanism to find a set of synchronization points that would satisfy a set of dependence constraints with minimal execution overhead.

7.5.4 Program Transformations

The process of parallelizing a program usually involves many program transformations. These transformations may be done manually by the programmer, or automatically by a parallelizing compiler. Most algorithms used for these transformations assume that programs do not contain synchronization. Extending these algorithms to accept "partially parallel" programs that contain synchronization would add more

power to a parallel programming environment. We believe that this thesis provides a foundation for research in this direction.

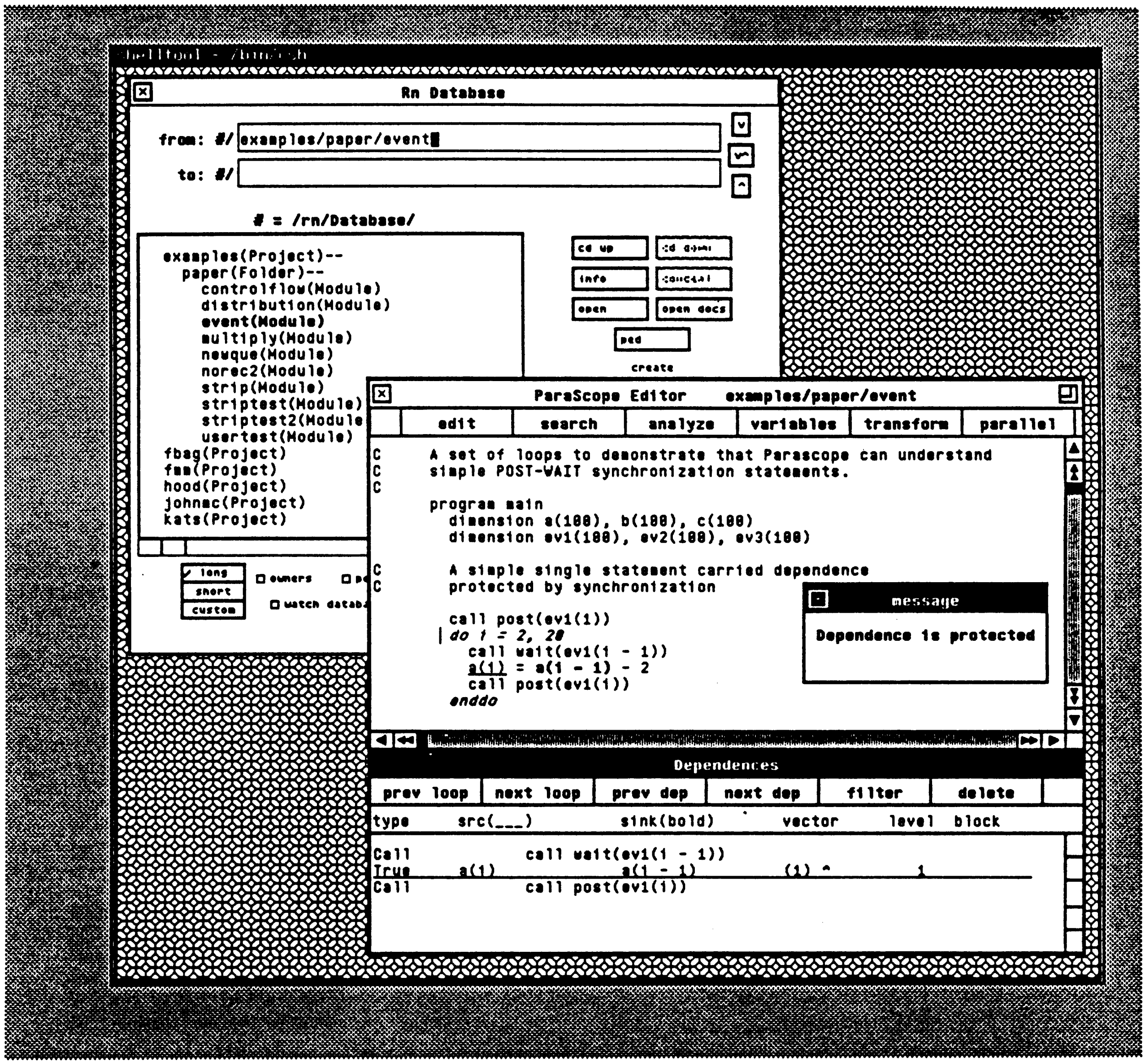
7.5.5 Synchronization in Distributed Memory Multiprocessors

Distributed memory multiprocessors use message passing for synchronization. A process issues a *send(msg)* instruction when it has data to send to another process. A process executes a *receive(msg)* to obtain data from another process. These message passing primitives may be blocking or non-blocking, depending on the implementation, leading to synchronous or asynchronous communication. Send and receive primitives are used to exchange data between processes running on different processors and provide synchronization between them.

Communication costs are an important execution overhead on distributed memory multiprocessors. The analysis of synchronization due to process communication is similar to the analysis of synchronization on a shared memory machine. The results of analysis of interprocess communication can be used for debugging, and for optimizing the communication to minimize execution overhead. We believe that our methods can be extended and brought to use in the distributed memory model of programming.

7.5.6 Combining Static and Dynamic Analysis

This thesis is geared towards statically finding memory references that may lead to data races during execution. Methods have been developed to identify data races on the fly during execution [Sch89, DS90] and by tracing memory references [EGP89, MC88]. One of the drawbacks of these dynamic debugging methods has been the high overhead associated with collecting information at runtime. Static analysis can be used to reduce the overhead of dynamic methods by finding a small set of potential data races before execution begins [HKMC90]. Only information that is relevant to these potential data races need be collected during execution. Thus combining static and dynamic analysis can be a powerful and efficient approach to identifying data races in parallel programs.



from: #/ examples/paper/event

to: #/

= /rn/Database/

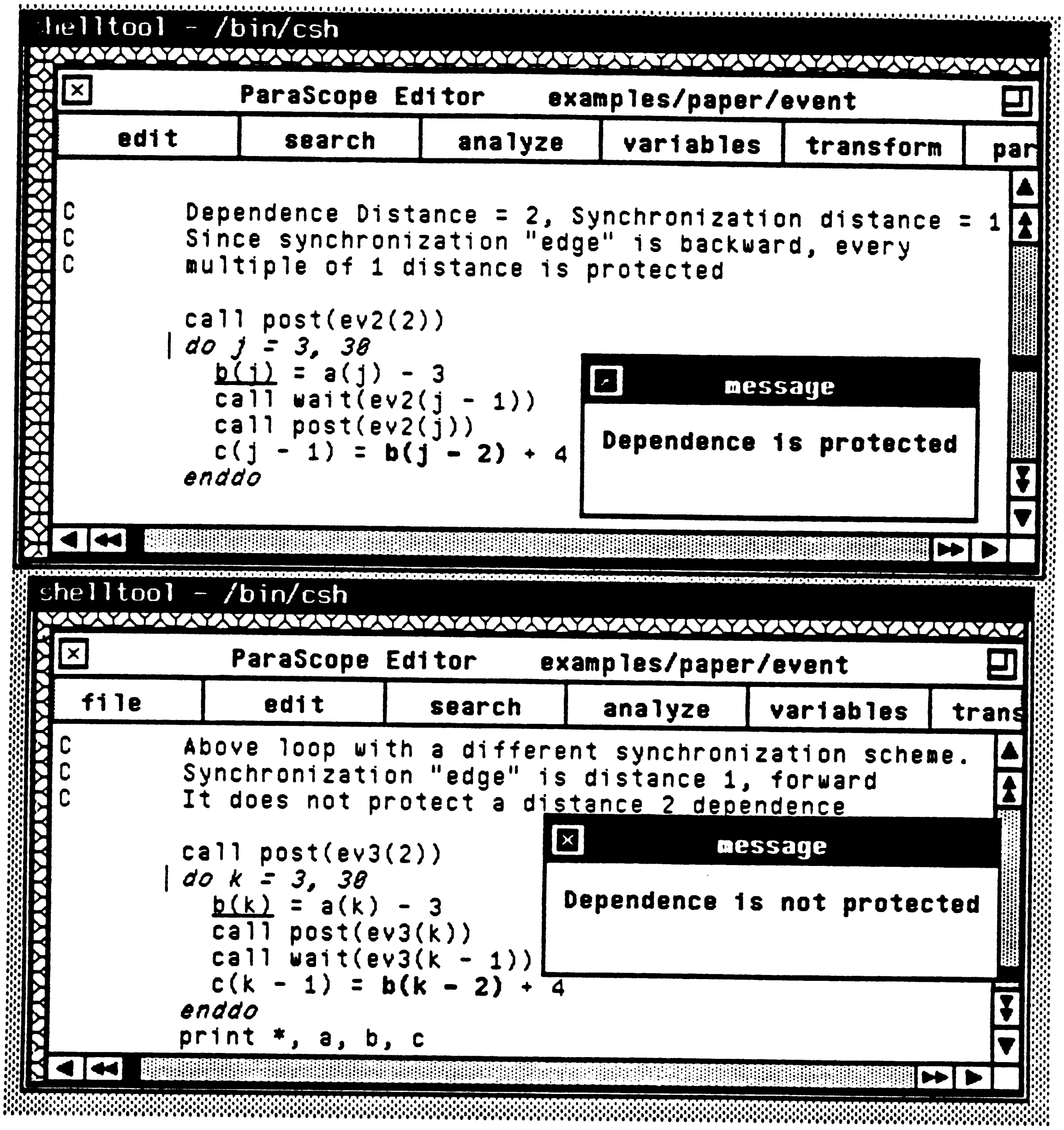
```
examples(Project)--
  paper(Folder)--
    controlflow(Module)
    distribution(Module)
    event(Module)
    multiply(Module)
    newque(Module)
    norec2(Module)
    strip(Module)
    striptest(Module)
    striptest2(Module
    usertest(Module)
fbag(Project)
fmm(Project)
hood(Project)
johnmc(Project)
kats(Project)
```

```
C     A set of loops to demonstrate that Parascope can understand
C     simple POST-WAIT synchronization statements.
C
      program main
        dimension a(100), b(100), c(100)
        dimension ev1(100), ev2(100), ev3(100)

C       A simple single statement carried dependence
C       protected by synchronization

        call post(ev1(1))
       |do i = 2, 20
          call wait(ev1(i - 1))
          a(i) = a(i - 1) - 2
          call post(ev1(i))
        enddo
```

Dependence is protected

```
type    src(___)         sink(bold)         vector      level  block
Call             call wait(ev1(i - 1))
True    a(i)               a(i - 1)         (1) ^         1
Call             call post(ev1(i))
```

Figure 7.1 View of Synchronization Analysis in ParaScope



```
C       Dependence Distance = 2, Synchronization distance = 1
C       Since synchronization "edge" is backward, every
C       multiple of 1 distance is protected

        call post(ev2(2))
        do j = 3, 30
          b(j) = a(j) - 3
          call wait(ev2(j - 1))
          call post(ev2(j))
          c(j - 1) = b(j - 2) + 4
        enddo
```

Dependence is protected

```
C       Above loop with a different synchronization scheme.
C       Synchronization "edge" is distance 1, forward
C       It does not protect a distance 2 dependence

        call post(ev3(2))
        do k = 3, 30
          b(k) = a(k) - 3
          call post(ev3(k))
          call wait(ev3(k - 1))
          c(k - 1) = b(k - 2) + 4
        enddo
        print *, a, b, c
```

Dependence is not protected

Figure 7.2 Example to demonstrate the effect of Event Synchronization on Data Dependences

Bibliography

[ABKP86] J. R. Allen, D. Bäumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing.* IEEE Computer Society Press, August 1986.

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

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AM85] W. F. Applebe and C. E. McDowell. Anomaly reporting — a tool for debugging and developing numerical algorithms. In *First International Conference on Supercomputers*, Florida, December 1985.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BBC+88] V. Balasundaram, D. Bäumgartner, D. Callahan, K. Kennedy, and J. Subhlok. PTOOL: A system for static analysis of parallel programs. Rice COMP TR88-71, Dept. of Computer Science, Rice University, June 1988.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: An interactive parallel programming tool. In *Proceedings SUPERCOMPUTING '89*, Reno, NV, November 1989.

[CCH+87] D. Callahan, K. Cooper, R. Hood, K. Kennedy, L Torczon, and S. Warren. Parallel programming support in ParaScope. In *Proceedings of the 1987 DFVLR Conference on Parallel Processing in Science and Engineering*, Koln, West Germany, June 1987. Available as Rice University, Dept. of Computer Science Technical Report TR87-59.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 1–10, Seattle, WA, March 1990.

[Duf74] R. J. Duffin. On fourier's analysis of linear inequality systems. *Mathematical Programming Study*, 1:71–95, 1974.

[EGP89] P. Emrath, S. Ghosh, and D. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings SUPERCOMPUTING '89*, pages 580–588, Reno, NV, November 1989.

[EP88] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison,WA, May 1988.

[FJS85] P. O. Frederickson, R. E. Jones, and B. T. Smith. Synchronization and control of parallel algorithms. *Parallel Computing*, (2):255–264, 1985.

[GGGJ88] V. Guarna, D. Gannon, Y. Gaur, and D. Jablonowski. Faust:an environment for programming parallel scientific applications. In *Proceedings of Supercomputing'88*, pages 3–10, Orlando,FA, November 1988.

[GN72] R. Garfinkel and G. Nemhauser. *Integer Programming.* John Wiley and Sons, New York, 1972.

[Hen87] L. Henderson. The usefulness of dependecy-analysis tools in parallel programming: Experiences using PTOOL. Technical Report Preprint LA-UR-87-3135, Los Alamos National Laboratory, September 1987.

[HKMC90] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Supercomputing 1990*, November 1990. (to appear).

[KHL77] A. Kaufmann and A. Henry-Labordere. *Integer and Mixed Programming.* Academic Press, New York, 1977.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eigth ACM Symposium on the Principles of Programming Languages*, pages 207–218, Williamsburgh, VA, January 1981.

[KMR87] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in BLAZE. In S. K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, August 1987.

[Kuc78] D. Kuck. *The Structures of Computers and Computation.* John Wiley and Sons, New York, 1978.

[LAs85] Z. Li and W. Abu-sufah. A technique for reducing synchronization overhead in large scale multiprocessors. In *Proceedings of the 12th International Symposium on Computer Architecture*, May 1985.

[LMC87] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[MC88] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, pages 135–144, Atlanta,GA, June 1988.

[MP86] S. P. Midkiff and D. A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 544–551, August 1986.

[MR85] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. Technical report, ICASE, NASA Langley Research Center, 1985.

[Par88] The Parallel Computing Forum. *PCF Fortran: Language Definition*, 1 edition, August 1988.

[Sch89] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, pages 285–297, Portland, OR, June 1989.

[Smi61] H. Smith. On systems of linear indeterminate equations and congruences. *Philosophical Transactions of the Royal Society of London*, A(151):293–326, 1861.

[Tar81] R. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3):594–614, July 1981.

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

[Tay83b] R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[Wol87] M. Wolfe. Multiprocessor synchronization for concurrent loops. Technical report, Kuck and Associates, June 1987.