INFORMATION FLOW IN PARALLEL PROGRAMS:

AN AXIOMATIC APPROACH

by

Richard Philip Reitman

TR 78-350

This research was supported by National Science Foundation grant MCS 77-07554.

INFORMATION FLOW IN PARALLEL PROGRAMS:
AN AXIOMATIC APPROACH

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

Richard Philip Reitman

August, 1978

INFORMATION FLOW IN PARALLEL PROGRAMS: AN AXIOMATIC APPROACH

Richard Philip Reitman, Ph.D.
Cornell University 1978

The information flow problem is concerned with controlling the transmission of information in computer systems. This thesis addresses this problem by developing an axiomatic logic that captures the information flow semantics of a program. Using this technique the scope of information flow analysis is extended from terminating sequential programs to parallel programs in which non-termination, synchronization and deadlock are possible. Once the information flow generated by a program has been determined, it is easy to check whether or not the program satisfies a given security policy.

The main contribution of the thesis is an axiomatic proof system for determining the flow of information produced by sequential or parallel programs. Just as proofs of correctness capture the effect of program execution upon the values in variables, proofs of information flow capture the effect of program execution upon the information in variables. An advantage of this approach is that once a

flow proof of a program has been generated, various security policies, such as high water mark or final value, can be verified readily.

Although flows in parallel programs need to be determined so that confidentiality in shared systems can be maintained, current information flow techniques are limited to terminating sequential programs. The thesis addresses this problem by capturing the flows generated by programs containing independent processes that synchronize with each other. The practicability of the method is demonstrated by developing the flow semantics for Concurrent Pascal.

Biographical Sketch

Richard Philip Reitman was born in New York City on August 14, 1952. He was graduated from The State University of New York at New Paltz in May of 1974 with a Bachelor of Science degree in Mathematics, awarded Summa Cum Laude. In May of 1976, he received a Masters of Science degree in Computer Science from Cornell University. He is a member of the Association for Computing Machinery.

To my parents, Norman and Lillian,

and to my wife-to-be, Martha

Acknowledgements

I am deeply grateful to my advisor, Greg Andrews. His contagious enthusiasm and constructive criticism made this thesis possible. His unfailing support and personal warmth made the long nights bearable.

I would also like to thank Carl Hauser, Bob Constable, and David Gries for their interest in my work and their valuable and enlightening insights. Discussions with valuable friends are appreciated and will be rememebered.

Special thanks go to Juris Hartmanis for his help and support throughout my years at Cornell.

Table of Contents

	page
1. Introduction	1
2. Overview of Computer Security	6
2.1 Access Control	6
2.2 Information Control	8
2.2.1 Domain Restriction	9
2.2.2 Run-time Flow Detection	12
2.2.3 Compile-time Security Certification	17
2.3 Data Integrity and Program Correctness	20
2.4 Our Work	22
3. The Use of Axiomatic Logic in Information Flow	25
3.1 Flows of Information in Programming Languages	26
3.1.1 Classification of Information	26
3.1.2 Types of Flows	30
3.1.2.1 Direct Flows	30
3.1.2.2 Indirect Flows	31
3.1.2.3 Covert Flows	34
3.2 Flow Proofs	37
3.2.1 Information States	37
3.2.2 State Transformations	38
3.2.3 Additional Notation	40

3.3 Proof Rules 41
3.3.1 Null Statement 43
3.3.2 Assignment 43
3.3.3 Alternation 46
3.3.4 Iteration 49
3.3.5 Compostion and Consequence 52
3.4 Example Flow Proofs 54
3.4.1 Proofs of Program Statements 54
3.4.1.1 Proofs of Assignment Statements 56
3.4.1.2 Proofs of Alternation Statements 57
3.4.1.3 Proofs of Iteration Statements 60
3.4.2 Program Flow Proofs 65
3.5 Certification of Program Security 70
4. Information Flow in Parallel Programs 76
4.1 Additional Flows of Information in Parallel Programs 77
4.2 Proof Rules 81
4.2.1 Parallel Execution 81
4.2.2 Semaphores 84
4.2.3 Message Passing 86
4.2.4 Critical Regions 88
4.2.5 Conditional Critical Regions 91
4.3 Example Proofs 94

4.3.1 A Bounded Buffer Example 96
4.3.2 Semaphores and Critical Regions 98
4.3.3 Information Flow and Process Control 106
5. Correctness and Information Flow 110
5.1 The Use of Correctness Proofs Alone 110
5.1.1 The Necessity for Strongest Proofs 111
5.1.2 Interpreting Correctness Proofs 114
5.1.3 Flow From Conditional Termination 119
5.2 Using Correctness with Flow Proofs 124
5.2.1 Specified and Actual Flows 125
5.2.2 The Combined Proof System 126
5.2.3 Examples Using the Combined System 130
6. Information Flow in Concurrent Pascal 134
6.1 Sequential Features 134
6.1.1 Expressions 134
6.1.2 Assignment Statements 136
6.1.3 Alternation 139
6.1.4 Iteration 141
6.1.5 Procedures and Data Encapsulation 143
6.2 Parallellism 146
6.3 Determining Security in Concurrent Pascal 151
6.4 An Example Flow Proof in Concurrent Pascal 152
6.5 Additional Language Features 162
7. Conclusion 164
7.1 Summary 164

7.2 Areas for Future Research 166

Bibliography 168

Chapter 1

Introduction

The proliferation of multi-programmed, multi-user computer systems has increased the need for ensuring the security of programs and data. To provide for the privacy of objects and information, protection mechanisms have been developed. If users can access objects and information only through the protection mechanisms, and if these mechanisms maintain the required degree of privacy, the computer system is said to be secure. Determining whether a computer system is secure has emerged as an important and difficult problem in operating system research [33,44,49,57,60].

In this thesis we consider the problem of ensuring the internal security of computer systems. We assume that they are externally secure, by which we mean that the computer installation exhibits two properties:

(1) the identities of users cannot be forged;

(2) any access to the computer facilities can only be accomplished through the use of system-defined commands.

In particular, we assume that the authentication of users upon entering the system is infallible and that the mechanism for restricting access to the physical machine and its peripherals is impenetrable. Although the problem of

ensuring external computer security is an important one, it is beyond the scope of this thesis. The interested reader can find discussions of external security in [29,55]. In the remainder of this thesis we will use the term "security" to denote "internal computer security".

There are three main aspects of security within multi-user computer systems. First, access to objects must be controlled so that only authorized forms of sharing can occur. Second, transmission (or flow) of information must be controlled so that the privacy of information can be maintained. The first problem is that of access control whereas as the second is known as information control. Third, the programs that implement protection mechanisms must be correct. In this thesis, we focus on information control and present a new approach that synthesizes and extends much of the previous work in this area.

A <u>security policy</u> specifies to what extent sharing of information is permitted within a particular computer system. The goal in information control is to prevent transmission of information unless it is authorized by the policy. Our approach is to prevent the execution of any program which could violate the security constraints. This approach is feasible only if the flow of information produced by a program can be ascertained. In this area we present four main contributions:

(1) An axiomatic logic for reasoning about information flow.

(2) The application of this logic to both parallel and sequential programs.

(3) A method for information control that can be utilized independent of any particular security policy.

(4) The combination of security certification and program verification into a powerful mechanism for information control.

In addition, we extend the scope of information flow to include programs that may not terminate. Our techniques are applicable to programs that may deadlock as well as those that contain possibly unbounded repetition.

The thesis is organized as follows. In Chapter 2 we review computer security in general and information control in particular. In addition, we review program verification and discuss its relation to computer security.

In Chapter 3 we introduce an axiomatic proof system for information control. The notions of information states, direct, indirect and covert flows of information, and security certification are discussed within this formal framework. We then incorporate these notions into an information flow proof system for a simple sequential programming language. Several example proofs are presented to illustrate the use of axiomatic logic in information

control. In addition, we compare our certification technique to others that have appeared in the literature.

The focus of Chapter 4 is on information flow in parallel programs. We begin the chapter by discussing the additional paths of information flow that exist in parallel programs, including shared variables, process synchronization, and process deadlock. We then examine a large number of parallel programming constructs and develop axioms and rules of inference that specify the flow of information produced by each of these constructs. Examples demonstrating both the use of these rules and the relationships among several synchronization primitives are presented at the end of the chapter.

In Chapter 5, we investigate the relationship between proofs of functional correctness and proofs of information flow. After demonstrating the differences between these two systems, we produce a composite system in which theorems concerning both functional correctness and information flow can be produced. We discover that there are theorems that can be proved in this composite system that cannot be proved in the information flow proof system alone.

The techniques and concepts of the previous chapters are applied to the programming language Concurrent Pascal in Chapter 6. As a consequence, additional language contructs, such as encapsulation mechanisms (capsules within a single

process, monitors for sharing between processes), are investigated. The chapter concludes by considering an example proof that demonstrates the proof rules for monitors and processes and captures flows of information that arise from process synchronization.

In Chapter 7 we summarize our results and contributions, and analyze their limitations. In addition, we indicate some areas for future research.

Chapter 2

Overview of Computer Security

There are three main aspects of computer security. First, computer systems must control access to objects so that only permissible forms of sharing can occur; this is known as access control. Second, secure systems must not divulge private information stored within the system; this is known as information control. Third, the programs that implement access and information control must do so accurately; this is known as program correctness. In this chapter we first present an overview of these three components of computer security and discuss the relationships among them; we then focus specifically on information control.

2.1 Access Control

Access control is concerned with controlling the execution of operations by subjects on objects stored within a computer system. Objects are the entities that can be manipulated and that may contain information; typically they include user files and program variables. Subjects are the entities that may manipulate objects; users, programs, processes, and procedures are common examples of subjects. In addition to subjects and objects, computer protection

systems specify a set of access rights that are used to control the invocation of operations. Associated with every operation p is a set of rights Rp. A subject s can perform operation p on object o if and only if for every right r specified in Rp, s has the right r for object o.

An access control policy specifies for each subject-object pair the set of rights that the subject may have for the object. Lampson introduced the access matrix concept as a means for representing the access policy [44]. In this scheme the rows of the matrix represent the subjects, the columns represent the objects, and the entries contain access rights. The (s,o) entry of the access matrix contains the system-defined right r if and only if subject s is permitted to have right r for object o. A graphical representation of the access matrix concept is presented in Figure 2.1.

Access security mechanisms are needed to control access to protected objects. In addition, they are a necessary prerequisite to achieving information security. Information flow mechanisms can be easily subverted by illegitimate access to objects containing protected information. Accordingly, no information security policy can be successfully maintained unless access security is provided.

Figure 2.1 A Graphical Representation of an Access Matrix

```
                          objects
                             o
subjects  +----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+
        s |    |    |    | wr |    |    |    |
          +----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+
```

subject s has access right wr to object o

2.2 Information Control

Although access control is an important element of computer security, it is not the only area of concern. It is often the case that information within an object should remain private, even if a subject has legitimate access to the object. The most common examples of this phenomenon are service procedures provided by operating systems, and programming languages that allow users to view complicated input/output functions as primitive operations. Although access control mechanisms can ensure legitimate access, they cannot prevent a subject from making a copy of an object

that it can read. This was first recognized by Lampson [41], who called it the confinement problem. In this section, we present three approaches for solving the confinement problem. The first is domain restriction, which ensures confinement by limiting a subject to an environment in which only permissable copying is permitted. The second is flow detection, in which illegal copying is detected and eliminated during execution. The last is program certification, in which programs that contain unauthorized copying are discovered prior to execution.

2.2.1 Domain Restriction

The domain restriction approach to information control is based upon the idea that we can utilize the same techniques for information control that we use so successfully for access control. In access control the extent of sharing is limited so that no unauthorized access can occur. In domain restriction the subject's environment, or domain, is limited even further to ensure the privacy of information.

Domain restriction is an attempt to control information by limiting the amount of sharing permitted within the system. The most extreme variation of domain restriction is to ensure the privacy of information by eliminating all sharing of objects. In this case each subject executes as if it were on its own dedicated machine. This form of

information control is extremely restrictive and is not practical for most computer systems. However it has been used in some military computer installations [1].

Some secure sharing is permitted in the UNIX system by permitting users to differentiate between public and private files [56]. Access control is maintained by differentiating between users. Although in the default mode every file is accessible to every user, users can selectively specify which files are to remain private. Some information control can be maintained through the use of the set user-id feature. This feature enables a user to make public a program which manipulates a private file without necessarily divulging the contents of the file. However, no more general form of information control, other than the elimination of sharing, is possible in the UNIX system.

More flexible forms of information control can be obtained by classifying information into security classes. Military security systems typically use the classes unclassified, confidential, secret, and top secret, along with the linear ordering:

unclassified < confidential < secret < top secret.

In this scheme a < b indicates that information of classification a is less sensitive than information of classification b. Bell and LaPadula have used this

classification scheme in devising a mechanism for controlling the transmission of information [8]. In their approach a distinction is made between objects from which information is extracted and objects to which information is transmitted. In addition to requiring access security (that every allowed access is authorized), Bell and LaPadula require what they have termed the *-property. A program satisfies the *-property if and only if for every object v viewed (read) by the program and every object m modified by the program, the security classification of v is less than or equal to the security classification of m. The *-property ensures that the military's security classification system cannot be violated. Similar approaches to information control and the confinement problem may be found in Andrews [3] and Jones [39].

Although the domain restriction technique ensures the secure control of information, it is not clear whether this technique can be applied in practice. It is a real possibility that the class of programs that exhibit the *-property is not sufficiently rich to permit the construction of usable systems. For example, programs that manipulate classified information and indicate their progress (eg. completion) on a system log violate the *-property unless the system log is as classified as any information in the system. In addition, the domain

restriction approach to information control makes the unrealistic assumption that the classification of the information contained in an object remains constant. In many applications the classification of the information contained in files changes dynamically as the result of program execution.

2.2.2 Run-time Flow Detection

In order to relax the constraints imposed by the domain restriction approach to information control, it is necessary to ensure that although a program has the access capabilities to transmit information illegally, it does not do so. One approach to do this is to monitor program execution and guarantee that no undesirable flows occur.

Fenton's data-mark machine is an application of run-time monitoring [30,31]. In his abstract machine, each memory location contains a static mark that indicates the classification of the information that is permitted to reside there. The hardware of the proposed machine is such that instructions of the form

MOVE a TO b

are not legal if the data cell a is marked with a classification that is higher than the data-mark of b.

Fenton also noticed that information could be transmitted

indirectly as a result of conditional execution. For example, the statement

```
IF a = 0 THEN b := 0
```

transmits information about a to b. To handle flows that result from conditional execution, Fenton included a data mark for the machine program counter. This mark indicates the classification of the information that is transmitted by the very fact that the current instruction is executed; it indicates the information that allowed the current execution path to be taken.

Although Fenton was able to handle flows resulting from conditional execution, he was unable to allow the classification of data cells to vary dynamically [30]. This is an undesirable restriction, especially for machines that have high speed registers. Using the data mark approach, the classification of each register must be fixed, so it is difficult to program efficient computations. (If all registers were of the same classification then all modified variables would have to be at least that classified). In addition, the classification of the program counter indicates the entire execution path, whereas it may be impossible to determine which sub-path was taken. For example, in the program

```
IF a = 0 THEN b := 0
c := 1
```

there is no information transmitted from a to c, despite the fact that there is a conditional branch (which is executed only if a = 0) prior to the the assignment to c.

Denning extended the results of Fenton to high level programming languages in which security classes changed dynamically. She developed a run-time technique for systems in which the classification scheme forms a lattice. Unfortunately, her proposal requires significant modifications to both the run-time and compile-time support of programming languages.

As a result of dynamically changing classes, code must be introduced into programs to capture the effect of conditional execution. In the program segment

```
b := 1
IF a = 0 THEN b := 0
```

there is a flow of information from a to b even if the assignment b := 0 is not executed. In Fenton's static protection system this was automatically handled, since if a is classified and b is not then the assignment b := 0 will never be executed. However, in a dynamic system there is no notion of security violation associated with the flow from a to b; rather the classification of b must simply be updated.

Denning's solution is to insert into the code generated by the compiler the instruction

"update the class of b with the class of a".

This ensures that the flow from a to b is indicated in the data mark of b, even if the assignment b := 0 is not executed.

In addition, Denning introduces a hardware stack to facilitate the computation of flow from conditional execution. Her technique is to push onto the stack the classification of conditions that control execution. Thus when an IF statement is encountered, the classification of the guard (boolean expression) of the statement is pushed onto the special hardware stack. The effect of assignment is to mark the modified cell to indicate the transmission of information from both the expression and the hardware condition stack. When the scope of a condition is exited, the stack is popped to indicate that information concerning the guard is no longer implicitly available. Unfortunately, this scheme incorrectly curtails the flow of information from guards of WHILE loops. In the statement sequence

```
WHILE a = 0 DO NULL
b := 0
```

there is a flow of information from a to b as a result of

the conditional termination of the WHILE loop. In the Denning system this flow is disregarded and the hardware condition stack is popped after the loop is exited.

In a related work, Jones and Lipton [40] present a surveillance mechanism for flow-chart programs. Program transformations are performed at compile-time to add security checks to compiled programs. In this scheme, all conditional flow is considered global so that the classification of a program counter, such as employed by Fenton, is sufficient. Jones and Lipton show how to prevent the transmission of information through the program's running time by terminating the program with an error notice as soon as a sensitive guard is evaluated. Their scheme, like Denning's, is an improvement over Fenton's in that security classes of objects are permitted to vary dynamically.

In addition, Jones and Lipton postulate the <u>observability principle</u>. Programs are viewed as functions from a set of inputs to a set of outputs. The observability principle states that the output of a program must encode <u>all</u> of the available information concerning the inputs. This principle requires that security system designers to precisely state any assumptions that have been made in the construction of the system. A typical assumption made by designers is the inability to transmit information across hidden, or <u>covert</u>,

channels. Some of the covert channels usually disregarded include: power consumption, disk head movement, paging rate, and program execution time. The observability principle is an important consideration in evaluating the applicability of security mechanisms.

2.2.3 Compile-time Security Certification

The main deficiency in run-time information flow detection is the cost in program size and execution speed. These difficulties arise from the fact that not executing program statements as well as executing them can cause flows of information. As a result, the run-time approach requires the use of program transformations, which increases program size considerably. In addition, considerable care must be taken in producing security violation notices since these notices may themselves transmit information.

Compile-time certification of information security is an attempt to relax the constraints of the domain restriction approach without resorting to run-time monitoring. This approach has the advantages that:

(1) security violation notices do not transmit sensitive information,

(2) programmers have assistance from the system in debugging programs with security violations, since these violations occur during compilation rather than during execution, and

(3) program size and execution time are unaffected by the security mechanism.

By analyzing programs and determining how information is transmitted, a larger class of programs can be validated using a compile-time certification technique than by domain restriction. Although certification mechanisms are more restrictive than run-time monitoring, it is reasonable to prevent programs that specify illegal flows of information from executing. It is the job of the programmer to specify only those flows that are desired and to eliminate programs that could cause a breach of information security.

Denning has developed a compile-time approach where the security classifications of system objects remain static and form a lattice [22,24]. In this approach, compile-time checks are performed to ensure that source programs do not specify a flow of information from a to b unless the static security binding of a is less than or equal to b. Programs which violate this <u>static binding policy</u> are prohibited from executing.

London [48] has devised a technique to ascertain the flow of information specified by a program in a system where the

security classifications of objects vary dynamically. It is an extension of the graphical approach proposed by Moore [51] to determine the flow of information from input variables to output variables. This approach, like that of Denning, assumes the transmission of information as a result of conditional execution is limited in scope. Thus both of these techniques cannot capture the effects of program non-termination. In addition, both approaches are closely tied to a particular type of security policy; the Denning approach is only valid for systems in which a static binding policy (classifications of objects are permanent) is followed, whereas the London approach is of use only in systems where a <u>final value policy</u> (only the final classifications are of interest) is employed.

Cohen has taken a slightly different view of information control [18]. His information theoretic approach can be summarized as follows. Let p(S) denote the final state produced by running program p with an initial S. Then a flow of information from a to b is said to occur if and only if there exist initial states S1 and S2 such that S1 and S2 differ only in the value of a but p(S1) and p(S2) differ in the value of b. The advantage of this view is that impossible execution paths are not considered so that the potential flow, rather than the specified flow, is determined. Unfortunately this approach leads to

undecidability problems and does not seem applicable to a programming language environment.

Although these compile-time approaches have many practical advantages over the domain restriction and execution monitoring techniques, there remain some serious drawbacks. First, each of these approaches is applicable for only a particular class of security policies. Second, flows from conditional execution are considered to be local in nature even when a programming language construct provides the power of non-termination. This is a very serious flaw, since it permits the certification of programs that may transmit classified information directly through program variables. Finally, these approaches have been of limited use in parallel programs in which independent computations communicate through some form of process synchronization. Our view is that the main application of information control is in large shared systems where parallellism and synchronization are the rule rather than the exception. Therefore, we feel that the compile-time approaches to information control need to be extended to programming languages that provide for parallel execution.

2.3 Data Integrity and Program Correctness

Access and information security can be ensured only if the programs that control access to objects and transmission

of information are correct. For example, security cannot be maintained in a system where the file manager allows unauthorized access. Historically, testing has been used as the means of ensuring functional correctness of computer programs. However, when the goal of program correctness is to ensure the adequacy of security mechanisms, testing techniques are unacceptably imprecise. In recent years several related techniques have been devised to precisely determine the functional correctness of computer programs [28,32,50].

The most widely used technique, and the one on which we will concentrate, is the axiomatic approach of Hoare [35]. His approach is to focus upon the state of variables, which he calls the program state. At any given point during program execution not every program state is possible; Hoare uses logical assertions to denote a set of possible program states. These assertions indicate restrictions that any possible program state must satisfy. For example, in every state which satisfies the assertion { x > 0 and y = x + 1 } the value of x is greater than zero and the value of y is one more than the value of x. Assertions may be simplified by using logical derivations. Thus the assertion { y > 1 } can be derived from the previous assertion by using some simple theorems from number theory and mathematical logic. This is reasonable since any program state that satisfies

the first assertion will also satisfy the second. We will subsequently use the notation P ⊢ Q to denote that assertion Q can be derived from assertion P.

In the Hoare approach, programs are viewed as the entities that transform the program state. For example, assignment statements cause a change in the program state as a result of the modification of program variables. The technique proposed by Hoare is to develop a system of logic to reason about these transformations. In this logical system, the transformation produced by each type of program statement is formally specified. Rules for statement composition and logical consequence are devised and the functional effect of a program is shown by composing proofs of smaller program segments. The development of logical systems and program proving are discussed in more detail in Chapter 3, where we produce a logic for reasoning about information flow. The interested reader can find a more detailed discussion of proofs of functional correctness in [28,35,37].

2.4 Our Work

In this thesis, we focus on the problem of controlling information in parallel systems where the classification of objects varies dynamically. We have abandoned the domain restriction approach to information control because it is

too restrictive to be of use in "real" systems. Run-time information flow detection is too difficult and costly in systems where the binding of classifications is dynamic to be a viable alternative. In such systems considerable hardware and software support is needed to handle statements that provide for conditional execution [23,30]. Accordingly, we choose to concentrate on the compile-time approach to information control.

Our technique is to verify the information security of computer programs prior to their execution. We view a program as a set of statements that transmit information between objects accessed by the program. Accordingly, we develop a proof system in which programming language statements are viewed as transformations on the state of information in the system. Rather than specifying these transformations with respect to a particular policy, as is done by Denning [23,24] and London [48], we consider these state transformations independent of any policy. Instead, the effect of program execution on the <u>information state</u> is proved using axioms and rules of inference that capture the flow of information produced by program statements. Once a proof of a program's effect on the information state has been completed, it is easy to determine whether the program violates a <u>particular</u> security policy. This approach has the advantage that the same flow proof is valid regardless

of the security policy; the certification of various types of information security policies differs only in the rules used to interpret the proof. In addition, our approach is applicable to parallel programs, where the need for information security is the greatest. In fact, in Chapter 6 we give a proof for a fairly large Concurrent Pascal program.

Chapter 3

The Use of Axiomatic Logic in Information Flow

In this chapter, we present basic concepts about axiomatic proof systems and information flow. In addition, we apply these concepts and develop a proof system for a simple sequential programming language. The language we consider has simple variables and arrays, expressions, and some basic programming statements. The statements include SKIP (empty statement), assignment, alternation (IF statement), iteration (WHILE loop), and composition (BEGIN). We do not consider procedures or functions in this chapter; we postpone their investigation, as well as that of more complicated alternation and iteration constructs, until Chapter 6 where the programming language Concurrent Pascal is investigated in detail.

Initially we determine the scope of the information flow problem by investigating the production of flows in programs. Next, we introduce the notion of information classification of objects, so that we can focus on information content rather than on data values. Given a classification scheme, we develop the notion of the information state of a system. In this framework, program statements are simply the means by which the information state can be transformed. We then devise axioms and rules of inference that capture the effect of statement execution

upon the information state. We conclude the chapter by discussing the relationship between proofs of information flow and the certification of secure programs. The power of this approach is demonstrated by showing that many common security policies are easily represented in this framework.

3.1 Flows of Information in Programming Languages

3.1.1 Classification of Information

In order to reason about the transmission of information, it is first necessary to define what is meant by information. Intuitively we have a notion of the sensitivity of information; certainly the information contained in the blueprints for Fort Knox is more sensitive than the contents of an article from our local newspaper. The information in our local newspaper is available to the general public and does not warrant protection, whereas the blueprints to Fort Knox are a highly guarded government secret. In this section we develop a means of classifying information that captures these variations in information sensitivity. Our method of classification is similar to that used by others [8,9,48].

A security classification scheme consists of a finite set of security classes C along with a partial ordering <=

between pairs of classes. The classes denote the degrees of confidentiality available in the system. At any given time every object v in the system contains information of some security class. We use the notation $\underline{v}$ to denote this class. As we will discuss in section 3.1.2, the execution of programs can modify the information contained in variable v, thereby affecting its classification $\underline{v}$.

The ordering <= is used in the classification scheme to indicate the relationship between classes. We use the notation $\underline{a}$ <= $\underline{b}$ to denote that the information contained in object b is at least as confidential as the information contained in a. In general the <= ordering is partial, namely for any three security classes $\underline{a}$, $\underline{b}$, and $\underline{c}$ the following three properties hold:

(1) $\underline{a}$ <= $\underline{a}$ (reflexive).

This ensures the internal consistency of security classes; information of a given class must be as confidential as any other information of that class.

(2) $\underline{a}$ <= $\underline{b}$ and $\underline{b}$ <= $\underline{c}$ implies $\underline{a}$ <= $\underline{c}$ (transitive).

Transitivity ensures that the relationships between classes are consistent.

(3) $\underline{a}$ <= $\underline{b}$ and $\underline{b}$ <= $\underline{a}$ implies $\underline{a}$ = $\underline{b}$ (anti-symmetric).

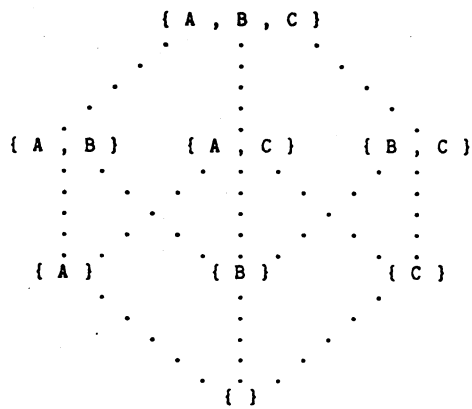
Redundant classes are prohibited by this property.

In addition we assume that for any two security classes a and b their least upper bound, denoted a • b, is defined. Although in some systems this least upper bound operator may not be defined, any information classification scheme can be transformed into one in which the least upper bound is defined and the relationships between the original classes is unchanged. This transformation can be performed by adding a new class (a,b) for each pair of old classes; (a,b) is the least upper bound of a and b.

When the least upper bound operator is defined and there exists a lowest class, the classification scheme (C, <=) forms a complete lattice [23]. We will use the class high to represent the class of the most sensitive information and the class low to represent the class of the least sensitive information. In programming languages, an interesting lattice is produced by considering the information contained in each input variable to be distinct from and incomparable to the information contained in any other input variable. This intuitive notion gives rise to a classification scheme in which each class indicates the set of inputs which could be present in information of that class. The scheme is presented graphically and in fuller detail in Figure 3.1.

Our classification system is applicable to arrays as well as simple variables. The information contained in an array is the sum of all the information contained in each element

Figure 3.1 Powerset Classification Lattice



C	=	powerset of { A , B , C }
<=	=	set inclusion
<u>high</u>	=	{ A , B , C }
<u>low</u>	=	{ } (empty set)
• operator	=	set union

of the array, where sum is the least upper bound operator of the security classification scheme. We do not differentiate between individual elements of an array; to do this it is necessary to reason about the value of subscripts. Since an information proof system only deals with information content and not with values, such a system cannot be used to reason about the information contained in individual elements of arrays. (Systems that combine features of information and correctness can be of use in treating array elements individually; one such proof system is presented in Chapter 5).

Expression are the means by which the values contained in variables are combined. Security classification schemes are used to indicate what information is encoded in the value of objects that appear in programs. Accordingly, the classification of an expression is the least upper bound of all the classes of the components of the expression. Thus the class of the expression a + (b * c) is $\underline{a} \bullet \underline{b} \bullet \underline{c}$.

3.1.2 Types of Flows

3.1.2.1 Direct Flows

In programming languages, the most common form of information transmission occurs as the result of a direct assignment. We use the term assignment here in a very

general sense; an assignment is said to occur if the value of some observable object is modified. Thus input and output, as well as normal assignment statements, cause a direct transmission of information. It should be noted that no flow of information is possible without assignment because otherwise no observable object is altered.

3.1.2.2 Indirect Flows

To be useful, a programming language must provide a means for conditional execution. As we discussed briefly in Chapter 2, conditional execution can produce flows of information. For example, the conditional statement

```
IF a = 0 THEN b := 0
```

causes a flow of information from a to b. More precisely, since execution of the THEN part is conditional on the value of a, b is set to zero only if a is equal to zero.

In the preceding example, the flow of information that resulted from conditional execution was limited in scope. Only objects modified within the THEN or ELSE part of an IF statement receive the information transmitted by the statement's guard (boolean expression). However, a statement that conditionally terminates can transmit information outside of its body. For example, the program segment

```
b := 1;
WHILE a ≠ 0 DO SKIP;
b := 0
```

assigns 0 to b if and only if a = 0. In this case, the information transmitted from the guard of the WHILE loop is not limited to the body of the loop. Instead it is transmitted to every object that could be modified if the loop terminates.

Sequencing of program statements can provide an additional type of information transmission. The program segment:

```
c := a;
b := c
```

copies information from a into b. Yet there is no direct assignment from a to b, nor is there any flow of information as a result of conditional execution. Rather the object c is used as an intermediary. Once the assignment c := a is performed, the transmission of information from c is equivalent to the transmission of information from a. The effects of sequencing can become more subtle when combined with iteration. Consider the loop:

```
FOR i := 1 TO n DO
BEGIN
    t := a;
    a := b;
    b := c;
    c := t
END
```

By looking exclusively at the body of the loop we can only ascertain that there can be a flow of information from a to both t and c, from b to a, and from c to b. Note however that the effect of the body is to perform a cyclic shift of a, b, and c. Thus as a result of the indefinite repetition of the body there can be a flow of information from any of a, b, and c to any of a, b, c, and t. Exactly what information is transmitted is controlled by the termination condition n. Accordingly, in addition to the flow of information caused by the repetition of the body of the loop, there is also a flow of information from n to any of a, b, c, and t as a result of the conditional execution of the loop. Notice that since the FOR loop must eventually terminate, the conditional flow is limited to the loop's body.

Indirect flows are produced as a result of sequencing and conditional execution. The repetition of a program segment can cause subtle indirect flows, since this repetition may

be equivalent to executing the program segment any number of times. In addition, the control of repetition may transmit information as a result of conditional execution. For repetitive constructs that are guaranteed to terminate this flow of information is local, whereas the conditional flow from constructs that conditionally terminate is global.

3.1.2.3 Covert Flows

In the preceding two sections, we have considered information transmission that can be deduced simply by analyzing program text. This type of flow is characterized by the fact that the source of the information, the object which receives the information, and the means of transmission are all represented within the program. The goal of this thesis is to develop a proof system that can capture the effect of this type of information flow.

More covert means of obtaining information are also available in most computer installations. For example many installations indicate program running time and other accounting data to users. The value of this data often reflects information manipulated by the program. For example, the running time of the program

```
FOR i := 1 TO n DO
BEGIN
    "statement sequence"
END
```

is directly proportional to the value of n. Hence allowing the user to view the running time also allows the user to view n.

The difficulty in analyzing this type of flow is that the running time is not represented within the program. It is an externally viewed object that is both difficult to represent in and difficult to eliminate from programs. Similar problems occur in systems where phenomena such as disk head movement, paging rate, and power consumption can be viewed by users.

In the remainder of the thesis we will disregard these covert channels and concentrate instead on those channels that can be represented within programming languages. Previous research has indicated that the elimination of covert channels is extremely difficult [41,46]. Jones and Lipton [40] have shown that information transmitted through a program's running time can be ascertained by considering the information that controls conditional execution. They postulate that any variation in a program's running time is caused because of alternate execution paths. Thus by determining the sensitivity of the information that can

control conditional execution, the sensitivity of the information encoded in the running time of a program can be determined.

Although this approach may be of use in stand-alone systems we feel that it is unrealistic for installations that employ multi-programming. In these systems, the running time of a program may depend upon the load upon the system. For example, consider the following two programs:

Program 1	Program 2
FOR i := 1 TO n DO "read a character"	"compute pi to 200 places"

If the system gives high priority to programs that are i/o bound and Program 1 and Program 2 are initiated at the same time, the running time of Program 2 will reflect the value of n despite the fact that Program 2 does not use the value of n to control its execution. Accordingly, we agree with the approach taken by Lipner [46] that it is futile to attempt to eliminate covert channels. Rather these channels must be identified and made so unreliable that their use in leaking information is prohibitively costly. The remainder of the thesis will focus on overt flows which are specified in programs. These flows may be legitimate; our goal is to precisely specify what flows are generated by a program and to provide people with a way to use this specification to

ensure information security.

3.2 Flow Proofs

3.2.1 Information States

The information state of a computer system at any particular moment is the classification of the information contained in every object in the system. We use logical assertions to specify restrictions on the set of possible information states. These assertions indicate relationships between the security classifications of objects. This is analogous to proofs of correctness that contain assertions indicating relationships between the values of objects. For example, the assertion { a <= b } represents the set of states where the information contained in object b is at least as classified as the information in a.

In addition to the information contained in objects, computer systems contain information as the result of conditional program execution. As discussed in section 3.1, this conditional flow can either be local or global in nature. Accordingly we introduce the auxiliary variables local and global to capture these flows. Thus local indicates the classification of information that is available as a result of conditional execution within a program statement; typical examples of this type of flow

arise in IF, FOR, and WHILE statements. On the other hand, global captures the flow of information between statements that arises from conditional execution. Conditional non-termination and process synchronization are common sources of global flow. The auxiliary variables local and global are needed to obtain meaningful flow proofs. Note that frequently auxiliary variables are also required in proofs of correctness [39,53,54].

In general, assertions concerning the information state include the auxiliary variables local and global, as in { a <= b, local <= low, global <= a }. Note that we use the comma to denote logical conjunction. Any object not mentioned in the assertion is not restricted in any way; it can contain information of any classification. Thus the assertion { a <= b, c <= high } is logically equivalent to the assertion { a <= b }, since the requirement c <= high is always true.

3.2.2 State Transformations

The information state changes as a result of program execution. Suppose the result of an assignment statement is to change the classification of an object from c1 to c2. When c2 is a less confidential class than c1, the object has been declassified as a result of executing the statement. However, if the new classification c2 is not less than c1

the object has received information of a new and additional type. For example, if a variable x in a system using the classification scheme of Figure 3.1 (powerset lattice) has its classification changed from { A } to { B }, it has received information concerning subject B, whereas previously it did not contain any such information. Note that this corresponds to executing the statement sequence x := a; x := b. It is not correct to say that the classification of the object has increased, since { A } and { B } are not comparable classes. Rather, x contains information that is independent of its previous value.

Both the declassification of objects and the propagation of information are important aspects of information control. We will use the term information flow, or simply flow, to denote either of these phenomena. Since a flow of information corresponds directly to a modification of the information state, the effect of information flow can be captured by reasoning about the functional effect of program execution.

Any information proof system consists of two major components. First, since program statements are the mechanisms through which information states can be altered, axioms and rules of inference that capture the effect of statement execution are required. Second, it is necessary to reason about the information states themselves. If an

information state is contained in the set of states { P }, and { P } is a subset of { Q }, then the assertion that the information state is contained in { Q } is also true. This notion of set inclusion can be captured in a logical system that uses the standard axioms and rules of inference from set theory and lattice theory. We will assume throughout the rest of the thesis the existence of such a logical system. The notation { P } |- { Q } will be employed to indicate that { Q } can be derived from { P } in this system.

3.2.3 Additional Notation

In discussing an information proof system, it is often cumbersome to treat assertions as a single logical expression. A more convenient view is to treat program variables, local flow, and global flow separately. We will occasionally use the notation { V, L, G } to represent these three components of assertions. In this notation, V is used to indicate the relationships between program variables and constants; it does not contain any references to <u>local</u> or <u>global</u>. The role of L is to indicate a restriction of the local conditional flow, therefore it is comprised of terms of the form <u>local</u> <= <u>e</u>, where e is an arbitrary class expression that does not contain <u>global</u>. In a similar manner G is used to indicate the inter-statement conditional

flow; its terms are of the form global <= e, where here e does not contain local. This notation will be used mainly in connection with rules of inference for program statements that can modify local or global, where the separation of assertions into its components is important. The decomposition of assertions into V, L, and G will be avoided elsewhere.

In the following discussion, we consider some common programming language constructs. We use upper case symbols to represent language keywords and lower case symbols for variables. A complete summary of our notational conventions is given in Figure 3.2.

3.3 Proof Rules

In this section we present axioms and rules of inference for a basic sequential programming language. A summary of the constructs included in this language is presented in Figure 3.3. Our goal is to precisely specify the transformations on the information state produced by various types of program statements. In each of the following subsections we discuss the axiom or rule of inference for one sequential programming language feature. A summary of these rules is presented at the end of this section in Figure 3.4.

Figure 3.2 Notation

<u>a</u>	classification of the information in a
<u>low</u>	least sensitive security classification
<u>high</u>	most sensitive security classification
<=	partial ordering between security classes
•	security class least upper bound operator
{ P }	the set of states satisfying assertion P
{ P, Q }	the set of states satisfying both P and Q
{ P | Q }	the set of states satisfying either P or Q
{ V, L, G }	expanded assertion: V - information contained in variables L - local conditional flow G - global conditional flow
P[a <- b]	the assertion P with every occurrence of a syntactically replaced by b
P |- Q	derivation: indicates that Q is derivable from P using set and lattice theory
P1, ... ,Pn ---------- P	rule of inference: given a proof of P1 through Pn one can infer P.

Figure 3.3 Summary of Programming Language Statements

null statement	SKIP
assignment	x := e
alternation	IF b THEN s1 ELSE s2
iteration	WHILE b DO s
composition	BEGIN s1; s2; ... ; sn END

3.3.1 Null Statement

Execution of the null statement, which in our language is denoted by SKIP, does not change the information state. More precisely, it performs the identity transformation. Accordingly the rule for the null statement is:

{ P } SKIP { P }

3.3.2 Assignment

Assignment is the fundamental mechanism for producing flows of information. An observer can only gain additional information as a result of a variation in the object that is being observed. Since objects that are not modified as a result of program execution cannot convey additional information to observers, only the assignment statement can

affect the classification of objects.

In addition to the information transferred directly by the assignment of values to variables, information can flow indirectly as the result of conditional execution. In Section 3.1.2 we discussed the effect of alternation and conditional iteration statements on information transmission. That section made the simple observation that in order for an assignment statement to modify the value of a variable, it must be executed. Since observers can determine whether an object has been modified, any information that can affect whether this modification occurs is at least partially transmitted to the observer.

The assignment axiom must therefore indicate that the information that determines whether the assignment statement is executed flows to the conditionally modified object. It must also indicate that information flows to the modified variable. Accordingly we introduce the axiom

$$\{ P[\ \underline{x} \leftarrow \underline{e} \bullet \underline{local} \bullet \underline{global}\ ] \} \quad x := e \quad \{ P \}$$

to capture the effect of an assignment statement upon the information state. Recall that <- is used to indicate syntactic substitution within assertions. Our assignment rule indicates that the new security class of x is the least upper bound of the classes of e, local, and global. In order for an assertion { P } to be true after the execution

of the assignment statement, the same assertion with the new class of x syntactically substituted for x must be true before the assignment.

Assignment to array elements is similar to that for ordinary variables; both assignments cause a flow of information from the expression assigned, as well as from the local and global flow, to the object being modified. However assignment to array elements has an additional complication. Our view that the classification of an array is at least as great as the least upper bound of all of its elements requires that the classification of an array cannot be decreased by assigning to a single element. Thus arrays can only be declassified when the entire array is assigned unclassified information at once. An additional source of information in array assignment comes from the subscript. Note that in the assignment a[i] := 1 there is a potential flow of information from i to a, since if initially a is entirely 0, the value of i can be ascertained by searching for the non-zero element of a. Accordingly we propose the rule

{ P[$\underline{a}$ <- $\underline{a}$ • $\underline{i}$ • $\underline{e}$ • $\underline{\text{local}}$ • $\underline{\text{global}}$] }

a[i] := e

{ P }

to capture the effect of the assignment to an array element.

3.3.3 Alternation

Conditional execution enables the transmission of information to any object that is conditionally modified. Our goal is to capture the conditional flow of information within a single program statement in the auxiliary flow variable local. Alternation statements are one type of programming language construct that cause a conditional intra-statement information flow. In this section we consider the simple alternation construct:

IF b THEN s1 ELSE s2

In the IF statement presented above, the statement s1 is executed if and only if the boolean expression b evaluates to true. Accordingly there is a flow of information from b to any variable changed within s1. In our axiomatic information proof system this is reflected by adding to the classification of the auxiliary variable local the classification of the information present in the boolean expression, where "adding" is used here to denote the least upper bound operation. Thus we must ensure that no proof of the IF statement is possible unless the associated proof of s1 uses a pre-condition that indicates the potential increase in the local flow. Of course the same reasoning applied to s2 yields a similar requirement.

Our goal in formulating a rule of inference for IF statements is to devise a way to produce a proof of the form

{ V, L, G } IF b THEN s1 ELSE s2 { U, L, G' }

Note that since L reflects the intra-statement conditional flow, this component of the assertion does not change as a result of executing the IF statement.

In order to reason about the transformation to the information state produced by an IF statement, it is necessary to reason about the statements which comprise it, namely s1 and s2. The initial information state of these statements is the same as that of the IF statement except that <u>local</u> has been potentially increased by the class of the boolean expression b. We can precisely state this requirement as follows. Let L' be the second component of the pre-condition for s1 and s2; let L be the second component of the pre-condition of the IF statement. The desired property is that L', with every occurrence of <u>local</u> syntactically replaced by <u>local</u> • <u>b</u>, can be derived from { V, L, G }. The substitution in L' is used here to produce an assertion that, if true before the potential increase of <u>local</u>, implies the validity of L'. Our requirement is to show that this assertion is true by deriving it from an assertion that is true prior to the increase in <u>local</u>.

From the assertion { V, L', G } we must derive, for both

s1 and s2, a final assertion that is compatible with the desired final assertion for the IF statement. As we noted above the second component of the assertion is of no consequence here since by definition it must remain constant between statements. Thus the required post-condition for s1 and s2 is { U, L', G' }. Combining these observations results in the following rule of inference:

```
{ V, L', G }  s1  { U, L', G' }
{ V, L', G }  s2  { U, L', G' }
  V, L , G    |-  L'[ local <- local • b ]
-------------------------------------------
{ V, L, G } IF b THEN s1 ELSE s2 { U, L, G' }
```

Proofs of IF statements with an empty ELSE part need not show the second condition in the rule above. For these statements the simpler rule:

```
{ V, L', G }  s1  { U, L', G' }
  V, L , G  |-  L'[ local <- local • b ]
-----------------------------------------
{ V, L, G } IF b THEN s1 { U, L, G' }
```

is sufficient. In the future will we simply use the rule of inference for IF statements that is most appropriate.

Note that viewed operationally, this rule indicates that local temporarily increases within the body of the IF statement, but is restored to its previous classification

after the IF statement is executed. The permanent effect of this construct is manifested within U, in which the change of the classification of program variables is indicated. Note also that the global flow may be changed as a result of the execution of s1 or s2; accordingly the third component of the post-condition is G'. (In the next section we will see that global can never decrease, since it indicates the class of information available to every subsequently executed program statement).

3.3.4 Iteration

Iterative constructs cause information flow in several ways. First, the repetition of a program statement may result in successive modifications to the information state. Second, the conditional nature of the repetition may produce a local flow of information from the guard of the iterative construct to any variable that is modified within the construct. Third, the conditional termination of the iteration provides a global flow of information to every variable that could be modified subsequent to the statement's termination. In this section we will focus on the iterative construct:

WHILE b DO s

Other iterative constructs will be discussed in Chapter 6.

To reason about the repetitive nature of the WHILE loop, it is necessary to determine an assertion that is always true prior to the execution of its body. This assertion is said to be <u>invariant</u> over the execution of the body since the final state of one iteration is the initial state of the next.

As was the case for alternation, iteration produces a local flow of information from the guard b to any variable modified in the statement s. Accordingly the local flow within the iterative construct is potentially increased. Again we will use L' to denote the assertion dealing with local flow within the construct; the constraints on L' are the same as in the rule for alternation. Thus if the pre-condition of the WHILE loop is { V, L, G }, the invariant that must be shown for s is { V, L', G }. This means that the pre-condition of the loop may have to be weakened so that V and G are invariant with respect to the execution of s. A discussion of the use of derivations to weaken assertions may be found in Section 3.3.5.

Conditional termination produces a global flow of information from the guard of WHILE loops. In addition, it can cause flows that were previously only local in scope to become global. This results from the fact that WHILE loops can only fail to terminate if they are executed. Thus any

information that can determine whether a conditionally terminated loop is executed becomes a component of the global flow. For example, the two statement sequences:

```
IF a ≠ 0                          WHILE a ≠ 0 DO SKIP;
    THEN WHILE true DO SKIP;      b := 0
b := 0
```

produce the same flow of information. In both cases the assignment b := 0 is executed if and only if the value of a is 0.

Our goal is to produce a rule that indicates that the global flow to statements that are executed after the WHILE loop includes the local flow prior to the execution of the loop and the information contained in the guard of the WHILE statement, as well as the global flow indicated by the loop invariant. We will use G' to denote the assertion concerning this increased global flow. As was the case in increasing local flow, we desire an assertion that, if true before the increase, will ensure that G' is true after the increase. The desired assertion is G' with every occurrence of <u>global</u> syntactically replaced by <u>global</u> • <u>local</u> • <u>b</u>; to ensure that this new assertion is true prior to the execution of s, we require that it be derivable from { V, L, G }, the assertion concerning the initial information state. This requirement, together with those

concerning local flow and the loop invariant, yield the following rule for the WHILE loop:

```
{ V, L', G }  s  { V, L', G }
  V, L , G   ¦-  L'[ local  <-  local • b ],
                 G'[ global <- global • local • b ]
-------------------------------------------------------
{ V, L, G } WHILE b DO s { V, L, G' }
```

Note that any flow that is contributed by s is captured by (and present within) the loop invariant. Also note that although execution of the WHILE loop may increase the global flow, the local flow remains unchanged; the local flow can only be increased within a single statement.

3.3.5 Composition and Consequence

The rule of inference for statement composition and the use of logical derivations in an information proof system are analogous to the corresponding rules in functional correctness proof systems. In statement composition, the final state resulting from one statement's execution is the initial state for the next statement. With respect to assertions this means that the post-condition for the first statement is the pre-condition for second. The result of this observation is the following rule:

```
{ P }     s1 { P1 }
{ Pi-1 } si { Pi }  ( 1 < i < n )
{ Pn-1 } sn { Q }
--------------------------------
{ P } BEGIN s1; ... ;sn END { Q }
```

Logical derivations are the means for reasoning about assertions concerning the information state, rather than reasoning about transformations on this state. Derivations provide a way to weaken assertions; the ability to do this is crucial in proofs involving either IF statements or WHILE loops. In order to produce a proof of an IF statement it is necessary to produce the same proof for both the THEN and the ELSE parts. The usual technique is to weaken a proof of one of these parts to produce a proof that is common to both. In proofs of WHILE loops, the loop invariant must be derived from the pre-condition of the loop.

Valid logical derivations may be used within an information flow proof system to produce a proof { P } s { Q } if there is a proof for s with a weaker pre-condition or a stronger post-condition or both. This requirement is stated precisely in the rule:

```
{ P' } s { Q' }
P  |-  P'
Q' |-  Q
---------------
{ P } s { Q }
```

A summary of all of the proof rules is given in Figure 3.4.

3.4 Example Flow Proofs

In this section we apply the above axioms and rules of inference to particular programs. First we consider some simple program statements that demonstrate the basic types of information transmission found within programming languages. We then consider the flows present in a few short programs.

3.4.1 Proofs of Program Statements

Each of the following sub-sections concentrates on flow proofs for a particular programming language construct. We assume that there are three sources of information, denoted a, b, and c, and that the classification scheme corresponds to the powerset lattice illustrated in Figure 3.1.

Figure 3.4 Information Flow Proof System

null

```
{ P }    SKIP    { P }
```

assignment

```
{ P[ x <- e • local • global ] }
x := e
{ P }

{ P[ a <- a • i • e • local • global ] }
a[i] := e
{ P }
```

alternation

```
{ V, L', G }  s1  { U, L', G' }
{ V, L', G }  s2  { U, L', G' }
  V, L, G  |-  L'[ local <- local • b ]
-------------------------------------------
{ V, L, G } IF b THEN s1 ELSE s2 { U, L, G' }

{ V, L', G }  s  { U, L', G' }
  V, L, G  |- L'[ local <- local • b ]
-----------------------------------------
{ V, L, G } IF b THEN s { U, L, G' }
```

iteration

```
{ V, L', G } s { V, L', G }
V, L, G |- L'[ local  <-  local • b ],
               G'[ global <- global • local • b ]
-------------------------------------------------
{ V, L, G } WHILE b DO s { V, L, G' }
```

composition

```
{ P }     s1 { P1 }
{ Pi-1 } si { Pi }  ( 1 < i < n )
{ Pn-1 } sn { Q }
--------------------------------
{ P } BEGIN s1; ... ;sn END { Q }
```

consequence

```
{ P' } s { Q' }
P  |- P'
Q' |- Q
---------------
{ P } s { Q }
```

3.4.1.1 Proofs of Assignment Statements

As we have already seen, information can be transmitted directly through variable assignment. The assignment axiom indicates that the source of the information includes the global and local flows, as well as the expression to be assigned. More precisely, the axiom states that in order for the information state after the execution of the assignment to satisfy the assertion P, the state prior to the assignment must satisfy the assertion obtained by substituting the least upper bound of local, global, and the class of the expression for the class of the variable being assigned. Thus we can give the proof:

{ a • local • global <= a • b, local <= b, global =low }
x := a
{ x <= a • b, local <= b, global = low }

Note that a • local • global <= a • b is implied by the fact that local <= b and global = low. Thus the more intuitive proof:

{ local <= b, global = low }
x := a
{ x <= a • b, local <= b, global <= low }

can be shown by using the rule of inference in addition to the assignment axiom.

3.4.1.2 Proofs of Alternation Statements

The production of indirect local flows is captured by the rule of inference for alternation statements. This rule indicates the increase in local flow that is produced by conditional execution. The rule for the IF statement requires that within the body of the IF, the class of the local flow include the class of the guard of the IF statement. We can produce the proof:

{ local = low, global = low }
IF b > 0 THEN x := a
{ x <= a • b, local = low, global = low }

by using the rule of inference for IF statements in combination with the assignment axiom and the rule of consequence.

To obtain the proof we first consider the local flow within the body of the IF. Intuitively we realize that local will be increased by b, thus the local flow assertion is local <= local • b, or more simply local <= b (initially local = low). Note that this assertion, which we denote by L', satisfies the requirement of the IF rule that:

L ⊢ L'[local <- local • b > 0]

where L is the assertion concerning the local flow outside of the IF statement. In this case the desired derivation is

that:

{ local = low } ⊢ { local • b > 0 <= b }

which is trivially true since local • b > 0 = b when local = low. Note that this also implies that any weaker assertion cannot satisfy the requirement of the IF rule, for if L' is local <= c we need that b <= c.

Using the new local flow assertion L' we next prove that:

{ L', global = low }
x := a
{ x <= a • b, L', global <= low }

Note that since L' is local <= b this proof has already been given in our previous discussion of assignment statements. Accordingly we have shown that all of the requirements of the IF rule are satisfied so that we may infer that:

{ local = low, global = low }
IF b > 0 THEN x := a
{ x <= a • b, local = low, global = low }

A complete and detailed flow proof for this simple IF statement is presented in Figure 3.5.

Figure 3.5 Proof of an IF Statement

Using the assignment axiom:

(1) { a • local • global <= a • b,
local <= b, global = low }

x := a

{ x <= a • b, local <= b, global = low }

Using the rule of consequence:

(2) local <= b, global = low ¦-

a • local • global <= a • b,
local <= b, global = low

proof of (1)

{ local <= b, global = low }

x := a

{ x <= a • b, local <= b, global = low }

Using the alternation rule

(3) local = low ¦- local • b > 0 <= b

proof of (2)

{ local = low, global = low }

IF b > 0 THEN x := a

{ x <= a • b, local = low, global = low }

3.4.1.3 Proofs of Iteration Statements

Flows from iterative constructs arise from both repetitive and conditional execution. The rule of inference for WHILE loops captures both the conditional flow within the loop and the global conditional flow to the remainder of the program. In addition, the loop invariant captures the flows that result from statement repetition.

We first consider proofs that capture the effect of statement repetition. Suppose the statement:

```
BEGIN
    t := x; x := y; y := z; z := x; i := i + 1
END
```

is the body of a WHILE loop. In this case the simple proof of the body:

```
{ x <= a, y <= b, z <= c, local = low, global = low }
BEGIN
    t := x; x := y; y := z; z := t; i := i + 1
END
{ x <= b, y <= c, z <= a, local = low, global = low }
```

is not sufficient to show the effect of executing the loop on the information state. Note that the above proof only shows the effect of the first iteration of the loop. The necessary proof is one that shows that an assertion is

invariant with respect to the execution of the loop body. Such a proof is:

Let V = { x <= a • b • c, y <= a • b • c,
z <= a • b • c, t <= a • b • c }

Let L = { local = low } Let G = { global = low }

```
{ V, L, G }
BEGIN
    t := x; x := y; y := z; z := t; i := i + 1
END
{ V, L, G }
```

The proof of this invariant uses the rules for assignment, composition, and consequence and is presented in Figure 3.6.

Using the proof of the loop invariant a proof for the statement:

```
WHILE i < k DO
    BEGIN
        t := x; x := y; y := z; z := t; i := i + 1
    END
```

can be produced. We assume that initially local, global, i, and k are all bounded by low; this will allow us to eliminate conditional flows from consideration. Accordingly the class of the local flow within the body of the loop is

bounded by low, so that the invariant we presented above is valid. A detailed proof of:

Let V = { x <= a • b • c, y <= a • b • c, z <= a • b • c,
t <= a • b • c }

Let L = { local = low } Let G = { global = low }

```
{ x <= a, y <= b, z <= c, i = k = low, L, G }
WHILE i < k DO
    BEGIN
        t := x; x := y; y := z; z := t; i := i + 1
    END
{ V, L, G }
```

is presented in Figure 3.6.

In addition to the effects of statement repetition, the rule of inference for the WHILE loop must capture the effect of conditional termination. In the statement

```
IF a = 0 THEN WHILE b = 0 DO x := 0
```

there is a global flow of information from a and b to any subsequently modified variable. The rule for the WHILE loop captures this flow by requiring that the global flow after a WHILE loop be "incremented" by the class of the guard and local. The proof outline of Figure 3.7 demonstrates the use of this rule. Note that in this program, the final value of

Figure 3.6 Proof of a Conditional Cyclic Shift

Let V = $\underline{x} <= \underline{a} \bullet \underline{b} \bullet \underline{c}$, $\underline{y} <= \underline{a} \bullet \underline{b} \bullet \underline{c}$, $\underline{z} <= \underline{a} \bullet \underline{b} \bullet \underline{c}$,
$\underline{t} <= \underline{a} \bullet \underline{b} \bullet \underline{c}$

Let L = $\underline{local} = \underline{low}$ Let G = $\underline{global} = \underline{low}$

(1) { V[$\underline{t}$ <- $\underline{x}$], L, G } t := x { V, L, G } assignment

(2) V, L, G ⊢ V[$\underline{t}$ <- $\underline{x}$], L, G
proof of (1)

{ V, L, G } t := x { V ,L, G } consequence

(3) { V, L, G } x := y { V, L, G }
(4) { V, L, G } y := z { V, L, G }
(5) { V, L, G } z := t { V, L, G }
(6) { V, L, G } i := i + 1 { V, L, G }
(3) - (6) by assignment and consequence, as in (1) - (2)

(7) proof of (2), (3), (4), (5), (6)

{ V, L, G } composition
BEGIN
t := x; x := y; y := z; z := t; i := i + 1
END
{ V, L, G }

(8) proof of (7)
V, L, G ⊢ L[$\underline{local}$ <- $\underline{local} \bullet \underline{i < k}$]
G[$\underline{global}$ <- $\underline{global} \bullet \underline{local} \bullet \underline{i < k}$]

{ V, L, G } iteration
WHILE i < k
BEGIN
t := x; x := y; y := z; z := t; i := i + 1
END
{ V, L, G }

(9) proof of (8)
$\underline{x} <= \underline{a}$, $\underline{y} <= \underline{b}$, $\underline{z} <= \underline{c}$, $\underline{t} = \underline{low}$, L, G ⊢
V, L, G

{ $\underline{x} <= \underline{a}$, $\underline{y} <= \underline{b}$, $\underline{z} <= \underline{c}$, $\underline{t} = \underline{low}$, L, G } consequence
WHILE i < k
BEGIN
t := x; x := y; y := z;
z := t; i := i + 1
END
{ V, L, G }

Figure 3.7 Global Flow from WHILE Loops

```
Let L   =  local = low
Let La  =  local <= a
Let Lab =  local <= a • b

Let G   =  global = low
Let Ga  =  global <= a
Let Gab =  global <= a • b


    { L, G }
    BEGIN
        { L, G }

        x := 1;
        { x = low, L, G }

        y := 1;
        { x = y = low, L, G }

        IF a = 0
        THEN
             { x = y = low, La, G } ;-

             { x <= a • b, y = low, La, G }
             WHILE b ≠ 0 DO

                  { x <= a • b, y = low, Lab, G }
                  x := 0;
                  { x <= a • b, y = low, Lab, G }

             { x <= a • b, y = low, La, Gab }

        { x <= a • b, y = low, L, Gab }
        y := 0
        { x <= a • b, y <= a • b, L, Gab }

    END
    { x <= a • b, y <= a • b, L, Gab }
```

x = 0 if and only if a = 0 and b ≠ 0, and that the final value of y = 0 if and only if a = 0 and b = 0; hence both x and y depend on a and b as the proof shows.

3.4.2 Program Flow Proofs

In this section we investigate the information flow generated by some common programming examples. Our focus will be on interpreting these proofs rather than on the technical details involved in the proofs. Accordingly, we present proof outlines rather than complete mathematical proofs.

The first program we consider finds the index of the maximum element in A[1] to A[n]. This simple function can be performed by:

```
BEGIN
    maxindex := 1;
    i := 2;
    WHILE i <= n DO
        BEGIN
            IF A[i] > A[maxindex] THEN maxindex := i;
            i := i + 1
        END
END
```

Note that there is no global flow from the array A; any global flow produced reflects only the information contained in n. In addition, there is a local flow of information from n to the variables modified within the WHILE loop, since n appears in the guard of the loop. Thus i and

maxindex receive information from n. Also note that information flows from A to maxindex indirectly as a result of the conditional execution of the IF statement. The outline of a flow proof that captures these flows is given in Figure 3.8.

The program to compute the maximum element of an array produces a flow of information from the array in question to the variable max. This is not surprising since the function of that program is to compute a value based on the value of the array. Other programs manipulate arrays but do not compute values based upon them; a very common and useful example of this type of program is sorting. In sorting the goal is to transform an array to a desirable state, namely sorted. However no value is computed by sorting programs, thus no flow of information from the array should occur. The program in Figure 3.9 uses the technique of successive maxima to sort the elements A[1] to A[n]. Note that the correctness invariant is that A[1] to A[first-1] are sorted and A[first] to A[n] remain to be sorted.

Figure 3.8 A Proof Outline for a Max Program

```
Let L  =    local  <= low
    L' =    local  <= n
    L" =    local  <= n • A

    G  =    global <= low
    G' =    global <= n

    I  =    maxindex <= n • A,
            i        <= n

    BEGIN

        { L, G }
        maxindex := n;

        i := 2;

        { maxindex = i = low, L, G }
        WHILE i <= n DO

            { I, L', G }
            BEGIN

                { I, L', G }
                IF A[i] > A[maxindex]
                    THEN

                        { I, L", G }
                        maxindex := i;
                        { I, L", G }

                { I, L', G }
                i := i + 1

                { I, L', G }

            END
        { I, L', G }

    END
    { I, L, G' }
```

Figure 3.9 Successive Maxima Sort

```
BEGIN
    first := 1;
    maxindex := 0;
    i := 0;
    temp := 0;

    WHILE first < n DO
        BEGIN
            maxindex := first;
            i := first + 1;

            WHILE i <= n DO
                BEGIN
                    IF A[i] > A[maxindex]
                        THEN maxindex := i;
                    i := i + 1
                END;

            temp := A[first];
            A[first] := A[maxindex];
            A[maxindex] := temp;
            first := first + 1
        END;

    temp := 0;
    maxindex := 0;
    i := 0
END
```

Note there is a direct flow of information from A to temp, maxindex, and i. However this transmission is eliminated by the unconditional assignment of 0 to these variables after the array is sorted. Since the global flow generated by this program is bounded by $\underline{n}$, the final classification of temp, maxindex, and i is also bounded by $\underline{n}$. The outline of a proof of this sort is presented in Figure 3.10. In this proof we use the symbol $\underline{c}$ to denote an initial upper bound for $\underline{A}$.

Figure 3.10 Proof Outline for a Successive Maxima Sort

```
Let L  = local <= low
    L' = local <= n
    L" = local <= n • A

    G  = global <= low
    G' = global <= n

    I1 = maxindex <= c • n, temp <= c • n, i <= c • n
    I2 = first <= n, A <= c • n

    { A <= c, L, G }
    BEGIN
        { A <= c, L, G }
        first := 1; maxindex := 0;
        i := 0; temp := 0;
        { A <= c, first = maxindex = i = temp = low, L, G }
        WHILE first < n DO
            BEGIN
                { I1, I2, L', G' }
                maxindex := first;
                i := first + 1;
                { I1, I2, L', G' }
                WHILE i <= n DO
                { I1, I2, L', G' }
                    BEGIN
                        { I1, I2, L', G' }
                        IF A[i] > A[maxindex]
                            THEN { I1, I2, L", G' }
                                maxindex := i;
                                { I1, I2, L", G' }
                        { I1, I2, L', G' }
                        i := i + 1
                        { I1, I2, L', G' }
                    END;
                { I1, I2, L', G' }
                temp := A[first];
                A[first] := A[maxindex];
                A[maxindex] := temp;
                first := first + 1
                { I1, I2, L', G' }
            END;
        { I1, I2, L, G' }
        temp := 0; maxindex := 0; i := 0
        { temp <= n, maxindex <= n, i <= n, I2, L, G' }
    END
    { temp <= n, maxindex <= n, i <= n, I2, L, G' }
```

The above proof demonstrates the importance of initializing variables before they are referenced. Uninitialized variables effectively contain information of class high, since no bound can be placed on their classification. Erasing the value of temporary variables is also important; otherwise these temporaries retain sensitive information and cannot be reused. Note that in our sort program the variables temp, maxindex, and i are initialized before and erased after they are used.

3.5 Certification of Program Security

Since our goal is to certify the information security of computer systems, it is not sufficient to simply determine the flow of information specified by a program. We must also demonstrate that the flows do not violate the information security policy.

An information security policy specifies the set of acceptable information states. In some types of policies, such as those employing a final value strategy [48], only the final information state must satisfy the policy. In others, such as those that use a high water mark criteria [23], all of the intermediate states of the program must satisfy the policy constraints. Policies that combine aspects of both of these types are also possible.

Once the flow of information specified by a program has

been determined, the certification of the program with respect to a particular information security policy can be performed by a simple mechanism. First, the policy must be defined by an assertion { P } that captures the set of secure information states. Next, every point in the program where the security requirements must hold is identified. Finally, the mechanism must show that { A } |- { P } for every assertion { A } that holds at a point in the program identified during the second step. In particular, for policies that are based on the final value concept, all that is required is that the final assertion be a subset of { P }. On the other hand, high water mark policies require that the policy assertion { P } can be derived from every intermediate assertion.

A common policy for the sort program of Figure 3.9 is defined by { A <= c • n, temp <= n, maxindex <= n, i <= n } This assertion requires that all the information present in the array A has been derived from the original information in A and n alone. (The value of n indicates how many elements of the array were to be sorted). In addition, it specifies that no information concerning the original array has been transmitted to any of the local variables used in the sorting process. Note that it is not possible to satisfy the policy { temp = maxindex = i = low } since the potential non-termination of the WHILE loops produces a

global flow from n. An iterative construct that is guaranteed to terminate (and hence avoids this unnecessary form of global flow) is discussed in Chapter 6. The use of proofs of terminations in conjunctions with flow proofs is discussed in Chapter 5.

The sort program can be easily certified with respect to the final value security policy that corresponds to the above assertion. Certification proceeds in two steps. First, the proof of Figure 3.10 is produced. Second, the policy assertion is derived from the post-condition of this proof. This step is trivial since the post-condition is

{ A <= c • n, temp <= n, maxindex <= n, i <= n,
first <= n, L, G' }

However the sort program cannot be certified when our policy assertion is used as a high water mark policy, since there are intermediate information states that violate the policy, in particular within the body of the WHILE loop.

Note that with respect to certification not all flow proofs are useful. The proof

{ a <= low }
S
{ a <= high }

is true for any program S. Just as proofs concerning the

functional effect of program execution are guided by the correctness assertion, proofs of information flow must be directed by the policy assertion. In fact, for systems that use a final value policy the notion of weakest pre-condition developed by Dijkstra can be successfully employed [28]. Given a final value policy assertion { P } the security goal of the programmer is to develop a program S such that:

(1) there is an initial assertion { I } such that a proof of { I } S { P } can be produced, and

(2) the initial information state satisfies { I }.

Many previous compile-time mechanisms for information control are special cases in our axiomatic approach. In the domain restriction approach [4,7,39], only programs that exhibit the *-property (every object read is of a lower classification than any object written) are certified. Such programs are easy to certify with our logic, since no variable can have its classification increased. Accordingly, the policy assertion is satisfied by each of the information states that occur during program execution.

Denning has generalized domain restriction to programs that do not exhibit the *-property [22]. Her technique is to determine whether a program specifies unauthorized flows of information with respect to a static security policy. In our system, this corresponds to a policy that is invariant

over the execution of any statement in the program. Unfortunately, this technique is not sufficient to determine that a violation of the policy is not specified (ie. the program is secure). For example, the program segment

```
a := 0;
b := a
```

is not certified with respect to the static policy { a <= high, b = low } using the Denning mechanism. However, it is obvious that the policy is maintained, as indicated by the flow proof outline:

```
{ a <= high, b = local = global = low }
a := 0;
{ a = b = local = global = low }
b := a
{ a = b = local = global = low } |-
{ a <= high, b = low }
```

Restrictions on security policies themselves have also been proposed; two of the most common ones are high water mark and final value [23,48]. High water mark policies correspond to an policy that always holds; however the binding of security classifications to objects is dynamic in nature. The flow proof presented above shows that the high water mark policy { a<= high, b <=low } is maintained. As

with static binding policies, it is easy to represent this type of policy in our logic. In final value policies only the final classifications of objects are of interest; once again this is a special case of our notion of policy.

One important contribution of this thesis is the unification of many diverse concepts and approaches to compile-time information control. Our work forms a foundation from which these ideas can be reconciled and evaluated. Of course, programs cannot be developed with only information security in mind; functional correctness is also of utmost importance. Just as programs that maintain security but are incorrect are of little use, programs that are correct but violate security are unacceptable for security-minded installations. It is our belief that programs for secure systems must be designed with <u>both</u> security and correctness in mind. In Chapter 5 we consider a combined logic in which both functional correctness and information security can be shown. The proof system presented in the remainder of this thesis will hopefully aid the development of a methodology for designing secure and correct programs.

Chapter 4

Information Flow in Parallel Programs

Contemporary computer systems handle a variety of concurrent activity. We view such a system as a parallel program, where some of the processes correspond to users. Thus it is necessary to reason about the flow of information in parallel programs to ensure the confidentiality of critical user or system information. In this chapter we focus our attention on information transmission in parallel programs and develop axioms and rules of inference for many parallel programming language constructs that have been proposed in the literature. In particular, we consider the COBEGIN statement for parallel execution as well as synchronization via semaphores, message passing, critical regions, and conditional critical regions. In section 4.1, we discuss the origin of flows unique to parallel programs. In section 4.2, axioms and rules of inference that capture the effect of parallel programming language features upon the information state are introduced. Section 4.3 is devoted to examples that demonstrate both the validity and the utility of these rules.

4.1 Additional Flows of Information in Parallel Programs

In sequential programs, information is transmitted either directly through object modification or indirectly from conditional execution. The same types of flow arise in parallel programs. However, the detection of these flows is made more difficult by the possibility of simultaneous execution and access. In addition, independent processes may transmit information through synchronization primitives such as semaphore signals and file locks. Although not handled by previous information flow techniques, we find that the effect of synchronization mechanisms can be ascertained when the mechanisms are expressed in a parallel programming language.

Independent processes can transmit information to one another directly by using either shared variables or message passing. A direct flow of information can occur if one process stores some information in a shared variable and another process later retrieves it. Thus the program segment

Process 1	Process 2
store := x	y := store

may assign the value of x to the variable y. Note that since the relative execution speeds of the two processes is

not known in general, we must assume that information flows from x to y.

Message passing features allow a similar type of direct information flow to occur. Although many similar message passing mechanisms have been proposed [12,58], with respect to information flow they are all equivalent. In this discussion we will focus upon a representative message passing mechanism, called message classes, in which a class contains messages of one type [5]. The statement

SEND(Mclass, message_value)

deposits a message value in Mclass for any process that wants a message from that class. The message can be obtained by a process by an invocation of

RECEIVE(Mclass, message_buffer)

It is obvious that this language feature provides for a direct flow of information from the sender to the receiver, since the value of the message is transmitted. Thus, just as assignment is the fundamental mechanism for information transmission within processes, modification of shared variables and message passing are the fundamental mechanisms for information transmission between processes.

In addition to direct flows, there can be indirect flows of information between processes. One obvious example of

this is deadlock. Just as potential non-termination causes information to flow between objects within a process, potential deadlock produces a flow of information between objects in different processes. For example, the program segment

```
Process A                    Process B

IF a = 0                     b := 1;
THEN SEND(M, 0)              RECEIVE(M, b);
```

transmits information from a to b as a result of the potential blocking of process B. Note that the final value of b is 0 if and only if a = 0.

The above program is an example of information transmission through process synchronization. Although the deliberate programming of potential deadlock is a rare occurrence in real programs, the use of synchronization primitives to control the relative speeds of processes is not. Unfortunately this type of synchronization can also indirectly transmit information. For example, the following program transmits one bit of information from object a (in process A) to object d (in process D) without any direct assignment (via the SEND primitive) from a to any of the message classes.

```
Process A                          Process B

IF a = 0                           RECEIVE(leakB,x);
    THEN BEGIN                     SEND(m,0);
        SEND(leakB,0);             SEND(synch,0);
        RECEIVE(synch,x);
        SEND(leakC,0);             Process C
    END
    ELSE BEGIN                     RECEIVE(leakC,x);
        SEND(leakC,0);             SEND(m,1);
        RECEIVE(synch,x);          SEND(synch,0);
        SEND(leakB,0);
    END;                           Process D

RECEIVE(synch,x);                  RECEIVE(m,d);
```

Note that the same values get transmitted along the message classes leakB and leakC, regardless of the of the value of a. However the order in which the messages are sent is determined by the value of a. Thus when a = 0 the message to process B is sent first. Although process B cannot determine that it has received the first message from A (and therefore that a = 0), it transmits this information through message class m to process D. Process D is able to determine whether a = 0 by examining the first message sent to it along message class m. If process B sent the first message the value assigned to d by the RECEIVE statement is

0; if C sent the first message it is 1. Thus the final value of d is equal to 0 if and only if the value of a is zero. Note that information is transmitted as a result of process A controlling the relative execution speeds of processes B and C. It is through this use of process synchronization that information flows from a to d. We will return to this example and examine it in detail in section 4.3.

4.2 Proof Rules

In this section, we present axioms and rules of inference that capture the effect of many parallel programming language constructs. The parallel features considered are: the cobegin statement, semaphores, message passing, critical regions, and conditional critical regions. Each rule is developed and discussed in the following sections; they are summarized in Figure 4.1.

4.2.1 Parallel Execution

As discussed above, flows between processes arise as a result of synchronization rather than as a result of parallel execution. Two completely independent and non-communicating processes cannot transmit information to one another. Therefore, the rule for parallellism need only

specify the mechanism for combining proofs of several processes into one proof for the entire program. This has been done for proofs of functional correctness by Owicki and Gries [53].

Owicki and Gries have observed that executing a collection of parallel processes is the same as executing each one seperately provided that their proofs do not interfere with one another. There are many ways to ensure non-interference of parallel processes. A first, but unappealing method, is to prohibit sharing. Another method is to ensure that operations on shared variables are indivisible and that every operation preserves an invariant state. If the invariant is maintained, and if the proofs of the operations on the shared variables assume only that the invariant is always preserved, then the proofs do not interfere. Owicki and Gries noticed a generalization of these requirements, namely that in order for a process A to invalidate a proof of process B there must be some statement in A and some assertion in the proof of B, such that executing the statement of A invalidates the assertion concerning B, assuming that statements are indivisible.

The notion of non-interference can be applied to proofs concerning the information state as well as those dealing with functional correctness. All of the observations concerning the value of variables also apply to their

classification. However, the definition of non-interference must be modified slightly to indicate that the auxiliary variables <u>local</u> and <u>global</u> exist seperately for each process (the fact that a process is executing a while loop does not affect the global conditional flow within another process). We define non-interference of flow proofs in two parts as follows. First, given a proof { V, L, G } S { V', L, G' } and a statement T with pre-condition pre(T) we say that T does not interfere with the above proof if:

(1) { V', pre(T) } T { V' }, and

(2) For any statement S' in S with precondition { U, L, G } { U, pre(T) } T { U }.

Second, let T be an assignment statement within Si. We say that the proofs { P1 } S1 { Q1 },...,{ Pn } Sn { Qn } are interference-free if for all j, j ≠ i, T does not interfere with { Pj } Sj { Qj }.

Since a COBEGIN statement terminates only if each of its processes terminate, the global flow from any process can be no greater than that produced by the COBEGIN itself. In addition, the assertion about the class of program variables from each process is valid. Accordingly, we propose the following rule of inference for the cobegin statement:

```
{ Vi, Li, G } Si { Ui, Li, Gi' }     (1 <= i <= n)
          are interference free

  Gi' |- G'                          (1 <= i <= n)
-------------------------------------------------

{ V1, ... , Vn, L, G }
COBEGIN S1 // ... // Sn COEND
{ U1, ... , Un, L, G'}
```

Note that the local flow L cannot be changed between statements, only within a statement, so that the final local flow is the same as the initial one. However, the global flow can be transformed from G to G'. Recall that the comma is used to denote logical "and" and that V1 through Vn and U1 through Un contain no references to <u>local</u> or <u>global</u>.

4.2.2 Semaphores

The legal operations on semaphores [27] are WAIT and SIGNAL; they can be thought of as request and release of a resource. A WAIT operation is an attempt to acquire the resource; if none is available the process is blocked. A SIGNAL releases one unit of the resource; if there are processes waiting, one of them is unblocked (ie acquires the resource).

As mentioned in section 4.1, the indefinite delay produced by synchronization primitives gives rise to an additional source of information flow. Although the SIGNAL operation completes immediately, the WAIT operation can

delay the process executing it. The effect of the WAIT operation upon the information state is the same as that of the following sequential program segment:

```
WHILE s = 0 DO SKIP;
s := s - 1;
```

Note, however, that a semaphore WAIT is an indivisible instruction and that it is seldom implemented as busy loops.

By examining the sequential program that corresponds to the WAIT operation, we see that the global flow will be incremented by the class of the semaphore. In addition, there is an assignment involving s; therefore, its classification is also affected. Note that the effect of the WAIT upon <u>s</u> is the same as that of assignment. Accordingly, we devise the following axiom for the semaphore WAIT operation:

$$\{ P[\ \underline{s} \leftarrow \underline{s} \bullet \underline{local} \bullet \underline{global},\ \underline{global} \leftarrow \underline{s} \bullet \underline{local} \bullet \underline{global}\ ] \}$$

WAIT(s)

{ P }

The SIGNAL operation is more straightforward. Its affect is the same as that of the simple assignment statement s := s + 1. It is this assignment that may allow another process to proceed by causing the boolean s = 0 to become false. The proof rule for the SIGNAL statement is the same

as that for the simple increment, namely:

{ P[s <- s • local • global] }
SIGNAL(s)
{ P }

To be of use in producing flow proofs, s must be the least upper bound of all the conditional (local and global) flows it received from within any process. Failure to do this will cause the proofs of these processes to interfere.

4.2.3 Message Passing

In the discussion of section 4.1, we showed that information can be transmitted indirectly through message passing either by conditionally causing deadlock or by controlling the relative speeds of processes. With respect to information flow, it is convenient to view message passing as an extended form of semaphore signalling. With semaphores only one bit of information is transmitted between processes, namely whether or not the value of the semaphore is 0. With message passing this is augmented by the information contained in the message transmitted. Thus the axiom for the message class operation RECEIVE is:

$$\{ P[\ \underline{x} \leftarrow \underline{m} \bullet \underline{local} \bullet \underline{global},\ \underline{m} \leftarrow \underline{m} \bullet \underline{local} \bullet \underline{global},\ \underline{global} \leftarrow \underline{m} \bullet \underline{local} \bullet \underline{global}\ ] \}$$

RECEIVE(m, x)

{ P }

Notice that the information that is contained in the message class flows to the received message x as well as the global flow. As usual there are local and global flows to all of the objects changed. The only difference between this rule and the one for the WAIT operation on semaphores is the role of x.

Just as RECEIVE is very similar to WAIT, SEND is very similar to SIGNAL. An additional complexity comes from the fact that a value v is transmitted as well as the one bit signal. The effect of this is to produce a flow from v to the message class m. This indicates that any RECEIVE operation could receive the information in v. We introduce the following axiom to capture the effect of SEND statements upon the information state.

$$\{ P[\ \underline{m} \leftarrow \underline{m} \bullet \underline{v} \bullet \underline{local} \bullet \underline{global}\ ] \}$$

SEND(m, v)

{ P }

Note that the classification of v is unchanged since the value of v is unchanged. As was the case for semaphores, $\underline{m}$ is an upper bound that is valid in every process.

4.2.4 Critical Regions

Another common mechanism for process communication and process synchronization is shared variables. However the unrestricted use of shared variables leads to programs that are both difficult to understand and difficult to reason about. The critical region construct

REGION r DO s

has been proposed to ensure that these difficulties do not arise [15]. A process executing a region statement is delayed until the shared variable r is free; at that time it executes the statement s. In parallel programs that use critical regions, no shared variable can be accessed except within one of these regions. The definition of a critical region ensures exclusive access to the shared variable protected by the region; thus shared variable invariants can be used in obtaining proofs.

The flows from critical regions are proved in three stages. First, for each shared variable r, an invariant assertion I(r) is established that indicates an upper bound on the information in r. Next, for each region statement REGION r DO s, a proof for s is obtained using no information concerning r other than the fact that I(r) is true when the region is entered (the proof can utilize assertions about the local variables of the process).

Finally, a proof is given that shows that s preserves the invariant; that is that after the execution of s, I(r) will be true.

It is obvious that r can contribute to the global flow, since it is possible for a process to be blocked at a region statement. In addition, since the state of the region is changed from free to busy upon entry into the region, and from busy to free upon exit from the region, the effect is that of an assignment to r at the beginning and at the end. For example the program segment

```
Process A                    Process B

REGION r DO                  b := 0;
    WHILE a = 0 DO           REGION r DO b := 1;
        SKIP;
```

produces a flow from a to b as a result of the conditional blocking of process B at the entry to its region statement. The flow occurs since if A enters its region first and a = 0, process B is blocked indefinitely.

Thus, any valid assertion concerning the information state prior to the execution of s must reflect this implicit modification of r as well as the updated local flow that results from entering the region. If this state satisfies the assertion { U, L', G } and the state prior to the entire REGION statement satisfies { V, L, G }, the following

constraint applies:

$$I(r),\ V,\ L,\ G \ \vdash\ U\ [\ \underline{r} \leftarrow \underline{r} \bullet \underline{local} \bullet \underline{global}\ ],$$
$$L'[\ \underline{local} \leftarrow \underline{r} \bullet \underline{local}\ ]$$

The first part of the constraint ensures that U reflects the possible increase in r due to its implicit assignment from free to busy. The second part of the constraint is used to update the auxiliary flow variable local within the scope of the region statement. Now consider the final state after executing s; let us denote it by { U', L', G' }. Exiting the region causes two implicit assignments to occur, one to r and one to global. Accordingly we need the additional constraint:

$$U',\ L',\ G' \ \vdash\ I(\ \underline{r} \leftarrow \underline{r} \bullet \underline{local} \bullet \underline{global}\ ),$$
$$W,$$
$$G''[\ \underline{global} \leftarrow \underline{r} \bullet \underline{local} \bullet \underline{global}\ ]$$

in order to produce a final state { W, L, G" }, where neither W nor G" contains any reference to r. By combining all of these constraints we arrive at the following rule:

```
{ U, L', G }  s  { U', L', G' }
I(r), V, L, G  ¦-  U [ r     <- r • local • global ],
                   L'[ local <- r • local ]

U', L', G'  ¦-  I( r <- r • local • global ),
                W,
                G"[ global <- r • local • global ]
            where W and G" do not refer to r
-------------------------------------------------------
{ V, L, G }   REGION r DO s   { W, L, G"}
```

The first constraint captures flows that occur within the body of the region statement. The second constraint indicates what additional information is obtained by the executing process from entering the region. The third constraint captures the information that is transmitted when the process exits the region. The last constraint also ensures that I(r) cannot be assumed outside of a critical region for r. This is necessary because some other process may be temporarily changing r within a critical region to a state which violates the invariant. All that is ensured is that when a process enters a critical region, the invariant is true.

4.2.5 Conditional Critical Regions

The conditional critical region is an extension of critical regions that allows explicit process synchronization [15]. The statement has the form

```
WITH r WHEN b DO s
```

and causes the executing process to block until the region r is free and the boolean b is true. Once unblocked, the process executes the statement s. For example, execution of the statement

```
WITH r WHEN r > 0 DO r := r - 1
```

is delayed until r > 0 and the region is free. Note that the above with-when statement is an implementation of the semaphore operation WAIT(r).

Obviously the effect of the with-when statement upon the information state is very similar to that of the region statement. The only difference arises from the boolean condition that occurs in the when clause. This condition contributes to the local flow within the body to the with-when since it controls whether the body is executed. In addition, other flows involving the boolean occur. First, since the remainder of the program is executed only if the with-when statement completes, there is a flow of information from the boolean to the auxiliary flow variable global. Second, there is a flow from the boolean to the shared variable r as a result of the implicit assignments to r at the beginning and at the end of the with-when statement. Note however that these flows are captured simply by increasing (incrementing) the local flow within

the body of the with-when to include the class of the boolean.

We can change the rule for the region statement into a rule for the with-when statement simply by changing the constraint:

$$I(r),\ V,\ L,\ G\ \vdash\ U\ [\ \underline{r} \leftarrow \underline{r} \bullet \underline{local} \bullet \underline{global}\ ],\ L'\ [\ \underline{local} \leftarrow \underline{r} \bullet \underline{local}\ ]$$

to the constraint:

$$I(r),\ V,\ L,\ G\ \vdash\ U\ [\ \underline{r} \leftarrow \underline{r} \bullet \underline{b} \bullet \underline{local} \bullet \underline{global}\ ],\ L'\ [\ \underline{local} \leftarrow \underline{r} \bullet \underline{b} \bullet \underline{local}\ ]$$

where { V, L, G } is the assertion prior to the statement

WITH r WHEN b DO s

and I(r) is the invariant associated with the shared variable r. By substituting the new requirement concerning local flow into the proof rule for the region statement we have the following rule of inference for with-when statements:

```
{ U, L', G }  s  { U', L', G' }

I(r), V, L, G  |- U [ r <- r • local • global ],
                   L'[ local <- r • b • local ]

U', L', G'  |-  I( r <- r • local • global ),
                W,
                G"[ global <- r • local • global ]

            where W and G" do not refer to r
-------------------------------------------------------------

{ V, L, G }   WITH r WHEN b DO s   { W, L, G"}
```

Note that since L' includes the flow from b, the flow from b to G" and to r is captured by the second and fourth constraints. A table summarizing all of the rules for information flow in parallel programs is given in Figure 4.1.

4.3 Example proofs

In this section we use the information flow proof rules for parallel programs to ascertain the flow of information generated by a sample program. We then focus on the implementation of critical regions using semaphores, and show that the proof rules for critical regions that are implemented with semaphores are analogous to those presented for critical regions in section 4.2. We then demonstrate this relationship by presenting a simple example in which the proof of the critical region is the same as the proof of its semaphore implementation. We conclude by examining the

Figure 4.1 Rules for Parallel Programs

```
PARALLEL EXECUTION:
    { Vi, L, G } Si { Ui, L, Gi' }        (1 <= i <= n)
            are interference free
    Gi' |- G'                             (1 <= i <= n)
    ---------------------------------------------------
    { V1, ... Vn, L, G }
    COBEGIN S1 // ... // Sn COEND
    { U1, ... , Un, L, G'}

SEMAPHORES:
    { P[ s       <- s ⊕ local ⊕ global,
         global  <- s ⊕ local ⊕ global ] }
    WAIT(s)
    { P }

    { P[ s <- s ⊕ local ⊕ global ], }
    SIGNAL(s)
    { P }

MESSAGE PASSING:
    { P[ x       <- m ⊕ local ⊕ global,
         m       <- m ⊕ local ⊕ global,
         global  <- m ⊕ local ⊕ global ] }
    RECEIVE(m, x)
    { P }

    { P[ m <- m ⊕ v ⊕ local ⊕ global ] }
    SEND(m, v)
    { P }

CRITICAL REGIONS:
    { U, L', G }  s  { U', L', G' }
    I(r), V, L , G  |-  L'[ local <- r ⊕ local ],
                        U[ r <- r ⊕ local ⊕ global ]
    U', L', G'  |-  I( r <- r ⊕ local ⊕ global ), W,
                    G"[ global <- r ⊕ local ⊕ global ]
                 where W and G" do not refer to r
    ---------------------------------------------------
    { V, L, G }   REGION r DO s   { W, L, G"}

    { U, L', G }  s  { U', L', G' }
    I(r), V, L , G  |-  L'[ local <- r ⊕ b ⊕ local ]
                        U [ r <- r ⊕ b ⊕ local ⊕ global ]
    U', L', G'   |-  I( r <- r ⊕ local ⊕ global ), W,
                     G"[ global <- r ⊕ local ⊕ global ] }
                 where W and G" do not refer to r
    ---------------------------------------------------
    { V, L, G }   WITH r WHEN b DO s   { W, L, G"}
```

example program of section 4.1. This is a more substantial example that uses message passing and contains subtle indirect flows caused by the control of process execution speed.

4.3.1 A Bounded Buffer Example

A standard problem in parallel programming occurs when a producer process sends a data stream to a consumer process. Typically, a buffer is provided between the processes to allow variation in their relative execution speeds. However, the buffer is limited to only N data values. In Figure 4.2 we consider a program presented in [53] in which an array of M values is sent from a producer to a consumer via a bounded buffer.

Global flow from the semaphores empty and full is produced by the WAIT statements within the WHILE loops. However, no such flow occurs from buffer, A, or B. Thus, although information concerning the number of values sent and received flows to local variables, no information about the actual values is transmitted. Information concerning M also flows to the local variables. Intuitively this makes sense, since if the producer sends less than M values the consumer deadlocks, and if the consumer receives less than

Figure 4.2 A Bounded Buffer

```
/* shared variables:
    in, out - integers indicating the number of values that
              have been produced and consumed, respectively.
    buffer  - buffering array, going from 0 to N-1.
    empty   - semaphore indicating the number of empty slots
              in the buffer.
    full    - same as empty, except for full slots.

    A       - array of M values to be transmitted
    B       - array to receive M values
*/

BEGIN
    in    := 0;  out  := 0;
    empty := N;  full := 0;

    COBEGIN /* producer:
                    local variable i - index into A
            */
        BEGIN
            i := 0;
            WHILE i <= M DO
                BEGIN
                    WAIT(empty);
                    buffer[in MOD N] := A[i];
                    in := in + 1;
                    i  := i  + 1;
                    SIGNAL(full)
                END
        END
    //
        /* consumer:
                local variable j - index into B
        */
        BEGIN
            j := 0; B := 0;
            WHILE j <= M DO
                BEGIN
                    WAIT(full);
                    B[j] := buffer[out MOD N];
                    out := out + 1;
                    j := j + 1;
                    SIGNAL(empty)
            END
        END
    COEND;
END
```

M-N values the producer deadlocks. An outline of a flow proof for the bounded buffer program is outlined in Figure 4.3.

Note that although shared variables are accessed without the use of critical regions, non-interference is preserved. None of the statements of the producer process invalidate any assertion in the consumer, since the assertion { in <= M, out <= M, empty <= M, full <= M, buffer <= A • M} remains valid throughout the body of the producer. Similar reasoning can be applied to show that the consumer does not invalidate the proof of the producer.

4.3.2 Semaphores and Critical Regions

A critical region is used to ensure that processes do not interfere. This is enforced by allowing a shared variable r to be accessed only within a critical region for r, and by ensuring that at most one process is ever in a critical region for r.

A critical region r can be easily implemented by a semaphore sem, which is initially 1. In this scheme entry to the critical region corresponds to WAIT(sem) whereas exit from the region is implemented by SIGNAL(sem). This implementation of critical regions corresponds to the following program transformation:

Figure 4.3 Flow Proof Outline of a Bounded Buffer

```
Let V  =  buffer = in = out = empty = full = low

    Ip =  buffer <= A • M, out <= M, in <= M,
          empty <= M, full <= M

    Ic =  out <= M, in <= M, empty <= M, full <= M

Let L  =  local  = low          Let G  =  global  = low
    Lm =  local <= M                Gm =  global <= low

BEGIN
    { buffer <= low, L, G }
    in  := 0; out := 0; empty := N; full  := 0;
    { V, L, G } |- { Ip, Ic, L, G }
    COBEGIN /* producer */
        BEGIN { Ip, L, G }
            i := 0;
            { Ip, i = low, L, G } |- { Ip, i <= M, L, Gm }
            WHILE i <= M DO
                BEGIN
                    { Ip, i <= M, Lm, Gm }
                    WAIT(empty);
                    buffer[in MOD N] := A[i];
                    in := in + 1; i  := i + 1; SIGNAL(full)
                    { Ip, i <= M, Lm, Gm }
                END
        END
        { Ip, L, Gm }
    //
        /* consumer */
        BEGIN { Ic, L, G }
            j := 0; B := 0;
            { Ic, B = j = low, L, G } |-
            { Ic, B <= A • M, j <= M, L, Gm }
            WHILE j <= M DO
                BEGIN
                    { Ic, B <= A • M, j <= M, Lm, Gm }
                    WAIT(full);
                    B[j] := buffer[out MOD N];
                    out := out+1; j := j+1; SIGNAL(empty)
                    { Ic, B <= A • M, j <= M, Lm, Gm }
            END
        END
    COEND;
    { Ip, Ic, B <= buffer • M, L, Gm } |-
    { Ip, Ic, B <= A • M, L, Gm }
END
```

Region Statements	Semaphores
REGION r DO	WAIT(sem);
s;	s;
	SIGNAL(sem);

Our goal in this section is to examine the implementation of critical regions using semaphores and discover whether this implementation produces the same transformation on the information state as we described in section 4.2. The rule presented in section 4.2 was produced by viewing region statements as a primitive synchronization mechanism. In this section we examine the standard implementation of regions and compare the information flow proof rule for regions presented in section 4.2 to the transformation of the information state specified by this implementation.

Let us consider the flows produced by the semaphore implementation of the statement REGION r DO s. We will assume that a proof of

{ U , L, GO }
s
{ U', L, G' }

has been obtained, where s is the body of the region statement. If the state prior to the execution of the

region is { V, L, G } and the state after this execution is { W, L, G" } the following proof outline is valid:

```
{ V, L, G } |-
{ U [ sem <- sem • local • global],
  L, G0[ global <- sem • local • global] }
WAIT(sem)

{ U , L, G0 }
s
{ U', L, G' } |-

{ W'[sem <- sem • local • global], L, G' }
SIGNAL(sem)
{ W', L, G' } |-
{ W , L, G" }
```

Note that although the execution of s cannot effect the classification of sem, it can change the classification of the shared variable. This is due to the fact that the shared variable really has two components, one that corresponds to the semaphore sem and one that corresponds to the shared global variable. However, for the purpose of this discussion the second component can be disregarded since any modification to it is captured by the proof of the statement s.

In order to obtain a proof of the form outlined above a

few simple requirements must be met. Obviously the proof of { U, L, G0 } s { U', L, G' } is a necessary component of a proof of the entire statement sequence. In addition the derivations indicated in the proof outline must be produced. Combining the constraints we arrive at the following list of conditions:

(1) { U, L, G0 } s { U', L, G' }

(2) V , L, G ⊢ U [sem <- sem • local • global]

(3) V , L, G ⊢ G0[global <- sem • local • global]

(4) U', L, G' ⊢ W'[sem <- sem • local • global]

(5) W', L, G' ⊢ W, L, G"

Note that this rule is less restrictive than the rule given for critical regions in section 4.2. The reason for this is that although critical regions are required to preserve the invariant state of their associated shared variable, the semaphore implementation need not. In addition, assertions concerning shared variables are not permitted outside of critical regions, but may be present anywhere in our semaphore implementation.

The consequence of these differences is that proofs of non-interference are necessary in the semaphore implementation, whereas critical regions ensure non-interference. The following additional constraints guarantee that the semaphore implementation of critical

regions also ensures non-interference:

(6) W' ⊢ I(r)

(7) neither W nor G" can contain a reference to r

(8) requirements 2 and 3 include I(r) in the hypothesis

Note that any rule concerning critical regions should not refer to W', since this assertion arises only from the fact that the exit from the region is implemented by SIGNAL(sem). W' can be eliminated by replacing references to it with U' whenever it is used as an hypothesis in a derivation, and by replacing it by the shared variable invariant whenever it occurs as the conclusion of a derivation. This substitution together with constraints (1)-(8) yield the following rule of inference:

```
{ U, L, GO }   s   { U', L, G' }

I(r), V, L, G  ⊢   U [sem    <- sem • local • global],
                   GO[global <- sem • local • global] }

U', L, G'      ⊢   I(sem <- sem • local • global),
                   W,
                   G" }

                 where W and G" do not refer to r
-------------------------------------------------------

{ V, L, G } REGION r DO s { W, L, G" }
```

Recall that the rule for critical regions formulated in section 4.2 is:

```
{ U, L', G }   s   { U', L', G' }
I(r), V, L , G  |-  L'[ local <- r • local ]
I(r), V, L', G  |-  U [ r <- r • local • global ]
U', L', G'      |-  I( r <- r • local • global ),
                    W,
                    G"[ global <- r • local • global ] }
                 where W and G" do not refer to r
-------------------------------------------------------
{ V, L, G } REGION r DO s { W, L, G" }
```

Note that the only difference in these rules is the fact that in the semaphore implementation there is a global flow produced by WAIT(sem) upon entry to the region, whereas in the rule for regions this flow is considered local. The effect is the same since:

(1) any assignment within the region receives information from both local and global, and

(2) global is increased to reflect the local flow when the region is released.

Thus our rule of inference for regions is the same as that produced by its standard semaphore implementation. In Figure 4.4 we present proofs for an simple program segment that demonstrate this equivalence. In the proofs given we assume that c1 • c2 • c3 <= c0 in order to preserve the shared variable invariant.

Figure 4.4 Comparison of Semaphores and Critical Regions

Critical Regions

```
assume:  c1 • c2 • c3 <= c0

I(r) =  r        <= c0
V    =  x        <= c1
L    =  local    <= c2
G    =  global   <= c3

L'   =  local    <= c0 • c2
U    =  r        <= c0 • c2 • c3,        x <= c1
U'   =  r        <= c0 • c1 • c2 • c3, x <= c1

W    =  x        <= c1
G"   =  global   <= c0 • c1 • c2 • c3 = c0

Note:  I(r), V, L , G  |-  L'[ local <- r • local ] ,
       I(r), V, L', G  |-  U[ r <- r • local • global]

    { V, L, G }
    REGION r DO
        { U, L', G } r := x { U', L', G } |-
        { I( r <- r • local • global ), W, L', G }
    { W, L, G" }
```

Semaphore Implementation of Critical Regions

```
assume c1 • c2 • c3 <= c0

V   =  r <= c0, sem <= c0, x <= c1
L   =  local <= c2
G   =  global <= c3

G'  =  global <= c0 • c2 • c3 = c0
U   =  r <= c0,          sem <= c0 • c2 • c3, x <= c1
U'  =  r <= c0 • c1 • c2 • c3, sem <= c0 • c2 • c3, x <= c1

    { V, L, G }
    WAIT(sem);
    { U, L, G' } r := x; { U', L, G' }
    SIGNAL(sem)
    { U', L, G' }  |-  { r <= c0, sem <= c0, L, G }
```

4.3.3 Information Flow and Process Control

In section 4.1 we presented a simple parallel program that produced a flow of information only as a result of the conditional control of the order of process execution. This program, which uses message passing as its only synchronization primitive, is summarized in Figure 4.5.

Figure 4.5 A Program with Flows due to Process Timing

```
Process A                          Process B

IF a = 0                           RECEIVE(leakB,x);
    THEN BEGIN                     SEND(m,0);
         SEND(leakB,0);            SEND(synch,0);
         RECEIVE(synch,x);
         SEND(leakC,0);            Process C
         END
                                   RECEIVE(leakC,x);
    ELSE BEGIN                     SEND(m,1);
         SEND(leakC,0);            SEND(synch,0);
         RECEIVE(synch,x);
         SEND(leakB,0);
         END;                      Process D

RECEIVE(synch,x);                  RECEIVE(m,d);
```

Any flow proof of this program must show that the execution of one process cannot interfere with the proof of another process. We will guarantee non-interference by finding classifications for the message channels that are not exceeded within any process. Since there are no shared variables in this program, no modification to any of the

message classes can cause these bounds to be breached. As a result, these classifications will be invariant throughout the execution of the program, so that non-interference is ensured.

Let us first consider the transmission of information within process A. Note that since the conditional modification of the message classes leakB, leakC, and synch is controlled by the value of the local variable a, there is a flow of information from a to these classes. This flow occurs despite the fact that messages to leakB and leakC are produced regardless of the value of a. As we shall see shortly, the relative ordering of the two SEND operations is of utmost importance in producing a flow of the information contained in a.

Processes B and C both send messages to channel m after receiving a message from leakB and leakC respectively. Since RECEIVE statements produce a global flow of information within the receiving process, and the class of these channels is at least a, the class of m is also at least a. Accordingly, no invariant for the message channels in this parallel program can be stronger than:

leakB <= a, leakC <= a, synch <= a, m <= a

Using this invariant, a proof of the above program is outlined in Figure 4.6. Note that the proof indicates that

there is a flow of information from the variable a in process A to the variable d in process D. A careful examination of this program will substantiate this result, for the final value of d is zero if and only if the initial value of a is also zero. The reader is invited to generalize this program to one that transmits an arbitrary number of bits from a to d. We will present such a program along with its proof in Chapter 6.

Figure 4.6 A Proof of Program 4.5

```
I   =   leakB <= a, leakC <= a, synch <= a, m <= a

L   =   local   = low
La  =   local   <= a

G   =   global  = low
Ga  =   global  <= a
```

Process A

```
{ I, L, G }
IF a = 0
    THEN BEGIN
        { I, La, G }
        SEND(leakB,0);
        { I, La, G }
        RECEIVE(synch,x);
        { I, x <= a, La, Ga }
        SEND(leakC,0);
        { I, x <= a, La, Ga }
        END
    ELSE BEGIN
        { I, La, G }
        SEND(leakC,0);
        { I, La, G }
        RECEIVE(synch,x);
        { I, x <= a, La, Ga }
        SEND(leakB,0);
        { I, x <= a, La, Ga }
        END;

    { I, x <= a, La, Ga }
    RECEIVE(synch,x);
    { I, x <= a, La, Ga }
```

Process B

```
{ I, L, G }
RECEIVE(leakB,x);
{ I, x <= a, L, Ga }
SEND(m,0);
{ I, x <= a, L, Ga }
SEND(synch,0)
{ I, x <= a, L, Ga }
```

Process C

```
{ I, L, G }
RECEIVE(leakC,x);
{ I, x <= a, L, Ga }
SEND(m,1);
{ I, x <= a, L, Ga }
SEND(synch, 0)
{ I, x <= a, L, Ga }
```

Process D

```
{ I, L, G }
RECEIVE(m,d)
{ I, d <= a, L, Ga }
```

Chapter 5

Correctness and Information Flow

Proofs of correctness reveal a great deal concerning the effect of program execution. The goal of this chapter is to examine the extent to which correctness proofs are of use in solving the information flow problem. We first consider using correctness proofs in lieu of flow proofs. After showing that when applied to information flow, proofs of correctness are not an acceptable alternative to flow proofs, we indicate how the correctness and flow proof systems can be combined to form a more powerful system for determining information flow in programs.

5.1 The Use of Correctness Proofs Alone

There are three main problems in using correctness proofs for information flow. First a valid correctness proof does not always completely indicate a program's effect on the information state. Second, there is no easy way to interpret correctness assertions and proofs with respect to information flow. Third, the possibility of non-termination is disregarded in correctness logics. In this section we discuss each of these difficulties in detail.

5.1.1 The Necessity for Strongest Proofs

An arbitrary proof need not indicate all the effects of a program's execution upon its variables. In particular, information may flow between two variables even though a valid proof of correctness does not indicate that the variables are related in value. For example, consider the following proof outline of a simple program that computes both factorial and sigma (sigma(n) = 1 + 2 + ... + n):

```
{ n>0 }
i := 1;
{ i=1 & n>0 }
fact  := 1; sigma := 1;
{ i=1 & fact=1 & n>0 } |- { 0<i<=n & fact=i! }
WHILE i < n DO
    BEGIN { 0<i<=n & fact=i! & i<n } |- { 0<i<n & fact=i! }
        i := i + 1;
        { 0<i<=n & fact=(i-1)! }
        fact  := fact  * i; sigma := sigma + i;
        { 0<i<=n & fact= i! }
    END
{ 0<i<=n & fact=i! & NOT i<n } |-
{ i=n & fact=i! } |-
{ fact=n! }
```

loop invariant = { 0<i<=n & fact=i! }

Part of the functional effect of this program is captured by the final assertion { fact = n! }. Although this is an accurate statement, it does not indicate the final relationship between the variables i, sigma and n. An even simpler proof that neglects the program's effect on both fact and sigma can be outlined as follows:

```
{ n>0 }
i := 1;
{ i=1 & n>0 }
fact  := 1;
sigma := 1;
{ i=1 & n>0 } |- { i<=n }
WHILE i < n DO
    BEGIN
        { i<n }
        i := i + 1;
        { i<=n }
        fact  := fact  * i;
        sigma := sigma + i
        { i<=n }
    END
{ i<=n & NOT i<n } |- { i=n }

loop invariant = { i<=n }
```

This indicates that it is not the case that any proof of correctness can be used when one is concerned with the flow of information produced by programs. In order for the proof to be valid in this context it must always make the strongest possible assertion about the program state. As is evident from the second proof of the factorial program, this includes finding the strongest possible loop invariant.

In general the problem of finding the strongest post-condition, given a pre-condition and a program statement, is not solvable. Note that the strongest post-condition of the pair < {true},y=f(x) > is {false} if and only if f(x) does not halt. In developing his proof system for a mini-language, Dijkstra avoided this problem by taking the view that we can only reason about programs that are guaranteed to terminate [28]. Taking a different approach, Cook addressed this problem by moving questions of undecidability into an <u>expressive</u> assertion language [21]. By hypothesizing the existence of a complete proof system for the assertion language, Cook is able to show <u>relative completeness</u>. Although from results in computability we know that no such complete system for an expressive assertion language exists, the goal of this theoretical approach is to ascertain properties about the proof system independent of the assertion language. However in practice this has the effect of producing proofs that are as

difficult to understand as the original programs. For example the proof:

```
{true}
WHILE b(x) DO s(x)
{ x = f(x') }

where
    f(a) = IF b(a) THEN f(s(a)) ELSE a
and
    x' = initial value of x
```

is not particularly useful in understanding the effect of program execution.

In the usual use of axiomatic systems, the proofs are goal directed and use derivations from the base logic to simplify assertions about the program state. Unfortunately, this goal directed approach must be abandoned when using correctness proofs to derive information flow; instead the strongest possible proof must be obtained. This results in proofs that are both difficult to obtain and difficult to understand.

5.1.2 Interpreting Correctness Proofs

Although using correctness proofs for determining information flow is cumbersome, this is not a sufficient

reason to abandon the approach entirely. There are several advantages to using correctness techniques to determine information flow. First, correctness proofs are already established, at least in the academic community, as a fruitful method for understanding programs. Second, a more precise determination of information flow is possible using the correctness approach, since some impossible execution paths can be eliminated from consideration. Finally, there already exist automatic verification systems for correctness proofs [10,20,32,59].

However there are more objections to the correctness approach to information flow than the difficulty of producing proofs that always derive the strongest post-condition, given a statement and a pre-condition. In order for correctness proofs to be of use in determining information flow, there must be an easy transformation from assertions about the functional effect of program execution to assertions about the transmission of information between program variables. Unfortunately, it appears that no easy transformation exists.

Assertions concerning the program state indicate the relationship between program variables and constants. These constants may be either logical constants that indicate previous values of program variables or the constant values that variables may denote. The goal in interpreting these

assertions is to determine which of the initial values of variables have affected which current values the variables contain.

In Figure 5.1 we present two programs that ensure that the variables x and y contain the same value. In addition, these programs indicate, through the variable eq, whether x and y were equal initially. Although the final assertions of the proofs for these programs are very similar, very different flows have been produced.

As we have observed in previous chapters, information can flow either directly across storage channels, or indirectly as a result of conditional execution. The assertion $x = y'$ in the first program indicates that there is a direct flow from y to x. Notice that there is an indirect flow from both x' and y' to eq in this program; it is reflected by the fact that $x' = y'$ if and only if $eq = 1$. One can see that this requirement is stronger than necessary; even if the statement eq := 1 is deleted from program1 there is still flow from x' and y' to eq. In that case eq is set to 0 if and only if $x' \neq y'$; thus flow can occur as long as initially $eq \neq 0$.

Figure 5.1 A Problem in Using Correctness for Flow

Program 1

```
eq := 1
IF x≠y THEN
   BEGIN
       x  := y;
       eq := 0
   END
```

Program 2

```
eq := 1;
IF x≠y THEN
   BEGIN
       y  := x;
       eq := 0
   END
```

Proof of Program 1

```
{ x=x' & y=y' }
eq := 1;
{ x=x' & y=y' & eq=1 }
IF x≠y
THEN { x=x' & y=y' & x≠y & eq=1 }
     BEGIN
         x := y;
         { x≠y' & x=y=y' & eq=1 }
         eq := 0
         { x≠y' & x=y=y' & eq=0 }
     END
{ (x≠y' & x=y=y' & eq=0) ¦ (x'=x=y=y' & eq=1) }
```

Proof of Program 2

```
{ x=x' & y=y' }
eq := 1;
{ x=x' & y=y' & eq=1 }
IF x≠y
THEN { y=y' & x=x' & x'≠y' & eq=1 }
     BEGIN
         y := x;
         { x'≠y' & y=x=x' & eq=1 }
         eq := 0;
         { x'≠y' & y=x=x' & eq=0 }
     END
{ (x'≠y' & y=x=x' & eq=0) ¦ (y'=y=x=x' & eq=1) }
```

In Program 2 the value of x is not changed, yet the final assertion seems to indicate a relationship between x and y'. There seems little in the assertion to distinguish between eq and x with respect to indirect flow from y'. The crucial difference is that whereas there is variation in the value of eq that is dependent upon y', the value of x is invariant. Thus although the value of eq indicates whether x' = y', the value of x does not.

The implications of this are that in order to use correctness proofs to determine information flow, it is necessary to compute, for every variable, whether there is a possibility of variation in its value. Next we need to express the assertions in disjunctive normal form. Any variable whose value may vary across the disjuncts receives an implicit flow of information from every term of a disjunct in which it appears. Thus, in the assertion { (x=0 & P) ¦ (x=0 & Q) ¦ R } x receives a flow from the terms of both P and Q if and only if R does not imply x = 0. Unfortunately, this means that even if we are able to produce a proof of correctness that always derives the strongest post-condition, it is still difficult to determine what the assertions indicate about the flow of information produced by the program of interest.

5.1.3 Flow From Conditional Termination

As was made evident in Chapter 3, the key to implicit information transmission by computer programs is variation in the values of variables. The variations that cause implicit flow arise as a result of conditional execution, such as is present in the alternation statement IF b THEN s1 ELSE s2. Additional sources of implicit flow are statements that conditionally terminate. The correctness approach to information flow is doomed to failure here since it either requires or assumes program termination.

In sequential programs, non-termination arises as a result of indefinite looping. Consider the program:

```
a := 0;
IF b THEN WHILE c DO SKIP;
a := 1;
```

The assignment a := 1 will be made if and only if the boolean expression b & c evaluates to false. However a partial correctness proof for this program is:

```
{ true }
a := 0;
{ a=0 }
IF b
THEN { a=0 & b }
     WHILE c DO
         { a=0 & b & c }
         SKIP;
         { a=0 & b & c }
     { a=0 & b & NOT c }

{ (a=0 & b & NOT c) ¦ (a=0 & NOT b) } =
{ a=0 & (true ¦ NOT c) }
a := 1;
{ a=1 & (true ¦ NOT c) }
```

Notice that in the final state there is no variation in the value of a, nor is there a relationship between a and the boolean expression b. This results from the fact that proofs of partial correctness indicate a valid restriction on the program state if execution of the program reaches a particular point. Even though execution of the statement a := 1 is contingent upon the fact that both b and c are false, there is no way for the usual correctness systems to reflect this.

Although most sequential programs are guaranteed to

terminate, many parallel programs are not. Operating systems occasionally allow deadlock to occur; programmers willingly run the risk of deadlock for the sake of efficiency. Since deadlock produces many of the same effects as indefinite looping, implicit flows can be produced here as well. For example, consider the program segment:

```
Process 1                    Process 2

wait(s');
eq := 1;                     IF c THEN signal(s);
IF b THEN wait(s);           signal(s');
eq := 0;
```

where s and s' are semaphores with initial value 0 and neither b nor c contain references to eq.

Note that eq is set to 0 if and only if b is false or c is true; thus there is flow from both b and c to eq. Unfortunately, in order for correctness techniques to capture this flow we need to use history variables that indicate the possible execution sequences of the parallel program. This means that in general the application of correctness proofs to information flow requires the consideration of all possible statement inter-leavings. Since any correctness proof that is valid for information flow must derive strongest post-conditions, given statements

and pre-conditions, proofs of parallel programs must examine time-dependent behavior. This is undesirable, since in doing this we lose the ability to represent time-independent abstractions in our proofs.

Indirect flows can also occur as the result of normal process synchronization. In the following program, information is transmitted from y to x as a result of the conditional sequencing of processes:

```
process1: IF y
          THEN signal(s1); wait(synch);
               signal(s0); wait(synch)
          ELSE signal(s0); wait(synch);
               signal(s1); wait(synch)

process2: wait(s0);
          x := 0;
          signal(synch);

process3: wait(s1);
          x := 1;
          signal(synch);
```

where the semaphores s0, s1, and synch are initially set to 0 and the boolean y contains no reference to x.

Note that no process deadlocks or loops indefinitely, yet there is a flow of information from y to x. Merely ordering

the execution of processes 2 and 3, along with a shared variable, is sufficient to cause a flow of information. The use of functional correctness to determine this flow is difficult. Again, as is the case with parallel programs that may deadlock, history variables must be introduced into the correctness proof in order to capture information flow. On the other hand, our flow techniques permit the proof of upper bounds on variable classification without resorting to time-dependent analysis.

Previous approaches to information flow have not considered the implicit flows that result from conditional termination. These flows have been typically regarded as "covert" despite the fact that both the source and the destination of the transmission of information are representable in the programming language, and in fact are essentially storage objects. Since correctness techniques are not applicable to programs that have these characteristics, we feel that this approach is not sufficient for solving the information flow problem.

The difficulty in using correctness proofs to determine information flow can be summarized as follows. In correctness proofs the initial program state indicates that there is no relationship between any of the program variables. The goal of the program prover is to enhance this view by deriving restrictions of the program state that

capture some of the inter-variable dependencies. However, in flow proofs, the initial information state indicates a relationship between every pair of program variables. The information flow proof eliminates some of the erroneous relationships that are initially present in the state. Thus any flow proof yields an upper bound on the information present in program variables, whereas only the strongest possible correctness proofs can be used for this purpose.

5.2 Using Correctness with Flow Proofs

In this section, we examine ways in which proofs of information flow can be combined with proofs of functional correctness. First we show that a flow proof captures all the flows specified by a program; unfortunately, it captures those arising along impossible execution paths. To eliminate this problem, we consider a combined proof system that incorporates both information flow and functional correctness. After specifying the meaning of assertions, rules, and proofs in the combined system, we present a simple example where the combined system is stronger than the flow system alone.

5.2.1 Specified and Actual Flows

Our approach to information flow has been to consider the flows that are specified by a program. However, it is possible to write programs that specify flows that cannot occur. This point is illustrated by the following program segment:

```
IF     b THEN y := x;
IF not b THEN z := y;
```

where b does not reference x, y, or z.

It should be clear that even though the assignment statements y := x and z := y specify a flow of information from x to y to z, there is no execution path along which this flow can occur. Unfortunately, the following flow proof cannot exploit this fact:

{ $\underline{local}$ = $\underline{low}$ }

IF b THEN { $\underline{local}$ <= $\underline{b}$ }

y := x;

{ $\underline{y}$ <= $\underline{x}$ • $\underline{b}$, $\underline{local}$ <= $\underline{b}$ }

{ $\underline{y}$ <= $\underline{x}$ • $\underline{b}$, $\underline{local}$ = $\underline{low}$ }

IF not b THEN { $\underline{y}$ <= $\underline{x}$ • $\underline{b}$, $\underline{local}$ <= $\underline{b}$ }

z := y;

{ $\underline{z}$ <= $\underline{x}$ • $\underline{b}$, $\underline{y}$ <= $\underline{x}$ • $\underline{b}$, $\underline{local}$ <= $\underline{b}$ }

{ $\underline{z}$ <= $\underline{x}$ • $\underline{b}$, $\underline{y}$ <= $\underline{x}$ • $\underline{b}$, $\underline{local}$ = $\underline{low}$ }

This weakness of flow proofs results from the fact that the information state reflects only the classification of the information contained in program variables, not the actual data value. Thus there is no indication in the state that if the THEN part of the first alternation statement is executed in the above program, it is impossible to execute the THEN part of the second statement. We have no way to state, let alone prove, that information flows from x to y if and only if B is true and from y to z if and only if B is false. In order to have this information reflected in the state we must introduce the concept of value into our proofs and assertions. The combined proof system of the next section does exactly this.

5.2.2 The Combined Proof System

In this section, we present a proof system that uses aspects from functional correctness to reason about values, and aspects from our flow logic to reason about information. The assertions in this system denote a restriction on the set of possible states, where a state is the cross product of the information state and the program state. The rules of the combined state indicate how statement execution transforms this composite state.

We use the standard notation for terms referring to the value of variables and the underlined name notation to refer

to the classification of the information present in that value. There is no problem with terms that are not homogeneous because it does not make sense to write an assertion such as x = y. Quantification is allowed over both values and classifications; since the set of security classes is finite, quantification over classes is merely a notational shorthand.

Terms of assertions can be combined using the usual logical operations such as and, or, and not. These operations can be used over terms that are concerned with different types of objects. For example, to indicate that the state must satisfy the constraints that the value of x is less than 0 and the classification of x is at most c, we would write { x < 0 , x <= c } where , is used as before to denote the and operator. We continue to denote assertions by { V, L, G } where appropriate; in the combined system, V contains references to the values of variables as well as their classifications.

Just as the state for the combined system is constructed from the states of the individual systems, the composite axioms and rules of inference are constructed from the corresponding rules of the flow and correctness logics. Once again the systems do not interfere since each is concerned with its own distinct domain of objects, correctness with values and flow with classifications. The

rules for the combined system are given in Figure 5.2.

The combined proof system has the ability to disregard impossible execution paths and to treat array elements individually. By reasoning about values, we can make valid assertions about A[i], the class of an individual array element. Note that the classification of the expression A[i] is A[i] • i, since information contained in the subscript i is also transmitted. The new assignment rule for array elements is:

{ P[A[i] <- e, A[i] <- e • i • local • global] }

A[i] := e

{ P }

This indicates that although information flows from the subscript to the array element, information contained in the rest of the array does not affect A[i].

Although the combined system of Figure 5.2 is only for sequential programs, there is no difficulty in producing one for parallel programs as well. Again, the independence of the flow and correctness logics permits the formation of a composite system. Non-interference in this system simply means that the execution of a process cannot invalidate the composite proof of another process.

Figure 5.2 Combined Flow and Correctness Proof Rules

```
null            { P }    SKIP    { P }

assignment      { P[ x <- e, x <- e • local • global ] }
                x := e
                { P }

alternation     { V &     b, L', G }  s1  { U, L', G' }
                { V & not b, L', G }  s2  { U, L', G' }
                L ¦- L'[ local <- local • b ]
                ----------------------------------------
                { V, L, G } IF b THEN s1 ELSE s2 { U, L, G' }

iteration       { V & b, L', G } s { V, L', G }
                L ¦- L'[ local  <-  local • b ]
                G ¦- G'[ global <- global • local • b ]
                ----------------------------------------
                { V, L, G } WHILE b DO s { V & not b, L, G' }

composition     { P }     s1 { P1 }
                { Pi-1 } si { Pi }    (1 < i < n)
                { Pn-1 } sn { Q }
                ------------------------------------
                { P } BEGIN S1; S2; ... ; Sn END { Q }

consequence     { P' } S { Q' }
                P  ¦- P'
                Q' ¦- Q
                ---------------
                { P } S { Q }
```

Note that our information logic can also be combined with a logic for total correctness. In this case global flows from unbounded iteration (WHILE loops) are avoided. However, global flows due to process synchronization are not eliminated, since these flows occur even in programs that are guaranteed to terminate.

5.2.3 Examples Using the Combined System

In this section we consider programs that illustrate the advantages of the combined proof system. To demonstrate the elimination of impossible execution paths, reconsider the program segment:

```
IF     b THEN y := x;
IF not b THEN z := y;
```

We observed before that although any flow proof must indicate that a flow of information from x to z will result from program execution, no such flow can occur. However, in the combined system we are able to reason about values as well as information classifications. Accordingly, we can now give the following proof:

{ y = z = local = global = low }

IF b THEN

{ b, y = z = global = low, local <= b }

y := x;

{ b, y <= x • b, z = global = low, local <= b }

{ not b, y = z = global = low, local <= b } -- else part

{ (y <= x • b, b) or (y = low, not b),

z = local = global = low }

IF not b THEN

{ (false or y = low), z = global = low, local <= b } |-

{ y = z = global = low, local <= b }

z := y;

{ z <= b, y = global = low, local <= b }

{ y <= x • b, z = global = low, local <= b } -- else part

{ y <= x • b, z <= b, local = global = low }

Note that the proof assumes that initially y, z, local, and global contain only information of the lowest sensitivity. This is reflected in the pre-condition { y = z = local = global = low }.

The above proof does not consider flows that would be produced if both assignment statements were executed. Notice that if the THEN part of the second alternation statement is about to be executed, we know that none of the

program variables have been changed, and in particular, that no information from x has been transmitted to y. As a result, the pre-condition of the statement z := y contains the fact that y = low. Accordingly, the final assertion indicates that although information can flow from x to y, no information can flow from x to z. Of course, there exist initial values for the boolean b such that information flows from b to either y or z. This flow is also reflected in the final assertion.

In addition to eliminating impossible execution paths, the combined system permits reasoning about individual array elements. The following program segment declassifies an array by setting every element to zero:

```
i := 1;
WHILE i <= n DO
    BEGIN
        A[i] := 0;
        i := i + 1
    END
```

Although in the flow logic we cannot show that the array elements A[1], A[2], ... A[n] have been declassified, in the combined proof system we can produce the proof of Figure 5.3.

Figure 5.3 A Proof Involving Array Elements

```
Let P  =  n = local = global = low
    I  =  A[j] = low (0 < j < i),   i = low, i <= n + 1, P
    I' =  A[j] = low (0 < j < i-1), I = low, i <= n + 1, P

{ P }
i := 1;
{ P, i = low }
WHILE i <= n DO
    BEGIN
        { I, i <= n }
        A[i] := 0;
        { I, i <= n, A[i] = low }
        i := i + 1
        { I', A[i-1] = low } |- { I }
    END
{ I , i > n } |- { I, i = n + 1 } |-
{ A[j] = low (1 <= j <= n) }
```

Note that we have assumed that no classified information is contained in n, local or global; information contained in these objects flows to every element of A that is modified. The final assertion indicates that every element of A that was set to zero no longer contains any sensitive information.

Chapter 6

Information Flow in Concurrent Pascal

The true test of our technique is in its application to real languages and programs. In this section we apply the axiomatic approach to information flow to programs written in Concurrent Pascal [13]. We have chosen Concurrent Pascal because it is a high level parallel programming language that has been used in the implementation of real operating systems. The rules presented in this chapter assume that proofs are only attempted for syntactically valid programs. Accordingly, type checking and scope restrictions are disregarded within the proof system.

6.1 Sequential Features

6.1.1 Expressions

The security classification of an expression indicates the sensitivity of the information encoded in the value of the expression. Most expressions in Concurrent Pascal are very similar to those found in our idealized language of Chapters 3 and 4. Expressions that are formed using standard operators, such as plus or times, produce a value that is an encoding of the information contained in each term of the expression. Array and set expressions also

follow this rule; the security class of A[i] is the least upper bound of the class of A and the class of i, whereas the classification of { a, b, c } is $\underline{a} \bullet \underline{b} \bullet \underline{c}$. Record expressions are slightly more complicated. Since by evaluating a record the value of any of its components may be obtained, we define the classification of a record to be the least upper bound of the classification of each of its components. Note that these components may be of any type, including record.

In Concurrent Pascal, as in most programming languages, functional expressions are allowed. Since these functions need not be total, the global flow between statements may be affected by the evaluation of expressions. Thus the effect of expression evaluation is two-fold: a value is returned that encodes the information present in the components of the expression, and the global flow is increased by the class of the information that determined whether or not the expression evaluation terminated. Accordingly, the class of an expression consists of these two components. For simple expressions the contribution to the global flow is low, but for function expressions it may be of a higher classification.

In the rules presented in the remaining sections of this chapter, we disregard the possibility of increasing global flow by evaluating expressions. More complete rules that

reflect this potential increase can be easily obtained by explicitly defining the class of an expression to have the two components presented above. The assignment rule now becomes:

```
{ P[ x      <- E.info • local • global,
     global <- E.inc  • local • global ] }
x := E
{ P }
```

where E.info is the classification of the value of the expression and E.inc is the contribution to global flow. Although similar rules for more complicated statements can be easily obtained, these rules are cumbersome. Therefore, simpler rules that assume termination of expression evaluation are used throughout the rest of this chapter. The simpler version of the information flow proof rules for expressions are summarized in Figure 6.1.

6.1.2 Assignment Statements

Recall that in a simple assignment statement, the class of the modified variable becomes the least upper bound of the classes of the assigned expression and the auxiliary variables local and global. Arrays are considered as single objects; in order to reason about individual array elements a combined proof system, similar to the one presented in

Chapter 5, is needed.

Assignment to record components affects not only the class of the component, but the class of the record as well. The class of any record that contains the modified component is potentially updated. Rather than introduce a special rule for assignment to record components, we require the re-evaluation of record classifications. Thus in any proof { P } S { Q }, Q can refer to record componenets, but not to entire records. However, the class of record variables may occur in P by using the axiom for record expressions presented in Figure 6.1. Note that requiring the application of this axiom immediately before the class of the record is used ensures that the correct classification will be obtained. Accordingly, both record and record component assignment are straightforward and are special cases of the standard assignment rule. The rules for assignment are summarized in Figure 6.2.

Figure 6.1 Classification of Expressions

class of < constant > = low

class of a op b = a • b

class of A[i] = A • i

class of r = X1 • ... • Xn

where r is of type:

```
RECORD
    X1 : T1
    :
    :
    Xn : Tn
END
```

Figure 6.2 Assignment Axioms

standard assignment:

{ P [x <- e • local • global] }

x := e

{ P }

array assignment:

{ P [A <- A • i • e • local • global] }

A[i] := e

{ P }

6.1.3 Alternation

Concurrent Pascal contains two forms of alternation; a simple conditional statement and a more general CASE statement. Both forms evaluate an expression and conditionally execute one of a sequence of statements depending upon the value of the expression. The flow arising from this conditional execution is local, that is, it only affects the statements within the alternation construct. Since any assertion about the effect of executing the alternation statement must be valid regardless of the value of the guard, each statement that is conditionally executed must transform the initial information state into one that satisfies the constraints of the proof of the entire alternation statement. In particular, we must show that all of the conditionally executed statements transform the initial state, with the local flow augmented by the class of the guard, into the final state. Note that there is only a temporary increase of local flow within the body of the alternation statement, since information from the guard is not available outside of it. The rules for the simple IF and more general CASE statements are summarized in Figure 6.3.

Figure 6.3 . Rules of Inference for Alternation Features

alternation:

```
{ V, L', G }  s1  { U, L', G' }
{ V, L', G }  s2  { U, L', G' }
L |- L'[ local <- local • guard ]
-----------------------------------------
{ V, L, G } IF guard THEN s1 ELSE s2 { U, L, G' }
```

case statement:

```
{ V, L', G }  si  { U, L', G' }  for 1 <= i <= n
L |- L'[ local <- local • guard ]
-----------------------------------------------
{ V, L , G }
    CASE guard OF
        l1: s1
            :
            :
        ln: sn
    END
{ U, L, G' }
```

6.1.4 Iteration

Concurrent Pascal contains four iterative constructs: while loops, repeat until loops, for loops and cycles. Since while and repeat until loops conditionally terminate, the guards of these loops can affect the information transmitted in program segments that are executed only if the while or repeat loop terminates. Thus their execution augments the global flow by the class of the guard of the loop. Global is not affected by for loops and cycles, however. For loops always terminate and cycles never terminate (hence there is no subsequent flow).

The body of iterative statements may be executed any number of times. In order to reason about the flow from these statements, it is necessary to produce an information state that is invariant with respect to the execution of the body. In this way, we can determine valid constraints of both the final information state and the intermediate states that arise during the execution of the loop. Note that in cycle statements, the final state of the loop is irrelevant since the loop can never terminate; however the intermediate states may still be of interest. The rules for the iterative constructs of Concurrent Pascal are presented in Figure 6.4.

Figure 6.4 Rules of Inference for Iterative Constructs

while loop:

```
{ V, L', G }  s  { V, L', G }
{ V, L , G } ¦- { L'[local  <- guard • local ] }
{ V, L', G } ¦- { G'[global <- guard • local • global] }
-------------------------------------------------------
{ V, L, G } WHILE guard DO s { V, L, G' }
```

for loop:

```
{ V', L', G }  s  { V', L', G }
{ V, L, G } ¦-
     { V'[ i     <- start • finish • local • global ],
       L'[ local <- start • finish • local ] }
-------------------------------------------------------
{ V, L, G } FOR i = start TO finish DO s { V', L, G }
```

repeat statement:

```
{ V, L , G }  s  { U, L , G' }
{ U, L', G'}  s  { U, L', G' }
{ U, L , G } ¦- { L'[local  <- local • guard ] }
{ U, L', G'} ¦- { G"[global <- local • guard • global] }
-------------------------------------------------------
{ V, L, G } REPEAT s UNTIL guard { U, L, G" }
```

cycle statement:

```
{ P }  s  { P }
-----------------------------------------
{ P } cycle s { Q }  for any assertion Q
```

6.1.5 Procedures and Data Encapsulation

Concurrent Pascal contains two constructs, classes and monitors, that encapsulate a data structure and its associated operations. These constructs are declared in a manner similar to simple variables and are initialized dynamically through use of the INIT statement. Monitor and class operations are called by specifying both the name of the component and the name of the operation. Both the INIT and call statements may be parameterized, with the restriction that initialization parameters must be passed as constants.

The information flow effect of invoking an operation of a system component is similar to procedure invocation in proofs of correctness; there is a substitution of actuals for formals and, given a flow proof of the body of the operation with a pre-condition P and post-condition Q, Q (with appropriate substitutions) is true after executing the call given that the pre-condition P (with appropriate substitutions) was true before the call. In order to represent the "reasonable" states of the data shared by the operations, there is an invariant that must be preserved by every component operation.

The global flow of the calling program is increased by the class of the information that controls the termination of the invoked routine. In order to capture this potential

increase to the global flow, as well as produce a proof of the calling program, we use the pre-condition global <= g in the proof of the body of the routine, where g is a previously unused logical variable. This enables the proof of the body to be stated in terms of g, rather than the greatest global flow that satisfies the pre-condition. The post-condition Q reflects the change to actual parameters in terms of the logical constant g; thus by substituting the global flow before the call for g, the effect upon the global flow for each particular call is accurately captured. We use an additional logical variable c to denote the potential increase of global by the operation. Thus proofs of the body of operations are of the form:

{ P, global <= g } s { Q, global <= g • c }

Note that c is used simply as a placeholder in our rule; in a proof it has a specific classification.

Components of the same type vary with respect to the initialization parameters. In order to reason about the operations of a particular component, we must know the classification of its parameters. We consider the class of a component to be the sequence of the classes of its initialization parameters. For example, the classification of a component v that has been initialized by the statement INIT v(a1, a2) is the sequence < a1, a2 >. Note that

although these sequences can be nested (components themselves may be passed as initialization parameters), they can only be finitely nested, since Concurrent Pascal requires definition before use. Thus, if a component of system type T1 is an initialization parameter to components of type T2, T2 must be declared after T1. This implies that T1 cannot use T2, so that recursively defined types are not possible.

The effect of component initialization is two-fold. First the component's block is invoked. This results in the execution of the initialization code once the component's variables have been allocated and initialized with information of low classification. In addition, the correspondence between component name and actual initialization parameters is produced. When subsequent operations are invoked, occurrences of formal initialization parameter classes are substituted by the class of the actuals, which have been saved in the component's classification. The rules for component initialization and routine invocation are presented in detail in Figure 6.5.

A short example that demonstrates the use of these rules is given in Figure 6.6. The proof of the class t shows that an invariant is maintained for its permanent variable store. Note that since the invariant is in terms of the class of the initialization parameter, different instances of the

class can have different invaraints. The INIT statement determines the classification of the parameters, and hence of the class instance itself. The proofs of class operations require that their parameters be bounded by the the class of the initialization parameter, otherwise the invariant cannot be maintained. Accordingly, a proof of a call statement can be produced only if its parameters satisfy this requirement.

6.2 Parallellism

A facility for concurrent execution is provided in Concurrent Pascal by the system type PROCESS. Processes are initiated by an INIT statement, just as monitors and classes are initialized. The initialization parameters to a processes include the monitors that the process may invoke. Since process variables are local, the classification of these variables are only of interest within the proof of the process. Therefore process initiation does not affect the parent process. However, in order for a program to satisfy a security policy, every process variable must satisfy the policy. The relationship between flow proofs of Concurrent Pascal programs and security policies is discussed more fully in Section 6.3.

Figure 6.5 Rules of Inference for System Components

Let v be of type:

```
MONITOR [or CLASS] (y)
    < variable declarations >
        :
    PROC ENTRY p( w ; VAR x )
        < variable declarations >
        S
        :
    S'
END
```

where y and w are sequences of formal value parameters,
x is a sequence of formal variable parameters,
p is an operation with body S, and
S' is the initialization statement.

Let INV be the invariant for permanent variables of v.

variable declaration:

$$\{ P \} \text{ VAR } z \{ P, \underline{z} \Leftarrow \underline{low} \}$$

component initialization:

$$\frac{\{ P, \underline{global} \Leftarrow \underline{g} \} \; S' \; \{ Q, INV, \underline{global} \Leftarrow \underline{g} \bullet \underline{c} \}}{\{ P[\underline{y} \leftarrow \underline{a}], R[\underline{global} \leftarrow \underline{global} \bullet \underline{c}, \underline{v} \leftarrow \underline{a}] \} \; \text{INIT } v(a) \; \{ Q[\underline{y} \leftarrow \underline{a}], R \}}$$

where a is a sequence of expressions

operation invocation:

$$\frac{\{ P, INV, \underline{global} \Leftarrow \underline{g} \} \; S \; \{ Q, INV, \underline{global} \Leftarrow \underline{g} \bullet \underline{c} \}}{\{ P[\underline{w} \leftarrow \underline{a}, \underline{x} \leftarrow \underline{b}, \underline{y} \leftarrow \underline{v}], R[\underline{global} \Leftarrow \underline{global} \bullet \underline{c}] \} \; v.p(a ; b) \; \{ Q[\underline{x} \leftarrow \underline{b}, \underline{y} \leftarrow \underline{v}], R \}}$$

where a is a sequence of expressions,
b is a sequence of variables, and
R contains no variable in b.

Figure 6.6 An Example Proof Using System Components

```
VAR t : CLASS(initval)
    BEGIN
        VAR store;

        " Let INV = store <= initval "

        " Let P = val   <= initval,
                  local <= initval , global <= initval "

        PROCEDURE ENTRY inc(val);

            { INV, P, global <= g }
            store := store + val;
            { INV, P, global <= g }

        PROCEDURE ENTRY dec(val);

            { INV, P, global <= g }
            store := store - val;
            { INV, P, global <= g }

        { store = low, local <= initval,
          global <= initval, global <= g }

        store := initval

        { INV, local <= initval,
               global <= initval, global <= g }

    END;

{ b <= a, t = local = global = low }

INIT t(a);
{ b <= a, t = < a >, local = global = low }

t.inc(b);
{ b <= a, t = < a >, local = global = low }
```

Process communication is possible in Concurrent Pascal through the use of monitors. This construct, as well as providing a data encapsulation mechanism as discussed in the previous section, ensures that operations on this data do not interfere. The invariant of a monitor captures the class of the information contained in its shared variables. In addition to the explicit transmission of information through shared data, process synchronization can also cause information to be transferred. In Concurrent Pascal, process synchronization is provided within monitors by the language type QUEUE.

Objects of type QUEUE contain at most one delayed process; they are either empty or full. The operations on queues are DELAY, which blocks the current process on a queue, and CONTINUE, which unblocks the process on the specified queue if the queue is non-empty. Since the execution speed of a delayed process is affected, DELAY(q) increases both the class of q and the class of the global flow. The statement CONTINUE(q) always terminates immediately; therefore it has the same effect as that of assignment. Since both of these statements release the monitor and enable other processes to enter it, the monitor invariant must be true before the execution of either of these queue operations. Since queue operations cannot invalidate this invariant, it is true after a queue

operation is completed. The rules and axioms dealing with queues are presented in Figure 6.7.

Figure 6.7 Axioms for Queues

determine the status of a queue:

$\underline{QUEUE(q)} = \underline{q}$

block a process on a queue:

$\{ INV, P[\ \underline{q} \leftarrow \underline{q} \bullet \underline{local} \bullet \underline{global},$
$\underline{global} \leftarrow \underline{q} \bullet \underline{local} \bullet \underline{global}\ ]\ \}$

DELAY(q)

{ INV, P }

resume a blocked process from a queue:

$\{ INV, P[\ \underline{q} \leftarrow \underline{q} \bullet \underline{local} \bullet \underline{global}\ \}$

CONTINUE(q)

{ INV, P }

where INV is the invariant of the monitor in which q is declared.

6.3. Determining Security in Concurrent Pascal

In Concurrent Pascal, as in our idealized parallel programming language, information policies are restrictions on the information state. A policy indicates requirements on the shared variables within monitors and on the local variables of processes. Since these variables can only be changed by the system component that created them, proofs concerning the entire information state of the system are unnecessary.

System components are initialized with "permanent" parameters that may affect the class of variables declared within the component. In monitors the class of the encapsulated data is usually dependent upon the class of the initialization parameters. Monitor operations cause a flow of information that is bounded by this class. The class of the actual monitor component reflects the class of its initialization parameters (see Section 6.1.5 and Figure 6.5).

A Concurrent Pascal program is certified by certifying each of the program's components. Since processes can communicate only through the use of shared variables, encapsulated by monitors, a process may transmit information to another process only by invoking monitor operations. The proof of a process reflects this by ensuring that the monitor call will not invalidate the monitor's invariant.

Therefore, as long as process proofs correctly capture the effect of monitor calls, proofs of processes and monitors can be produced independently. There is no need to represent the entire information state within the proof of the parallel program; rather, each component is proved with respect to the variables that it can affect. To verify that the class of a variable never exceeds its security constraint one need only verify those components that directly alter its class.

6.4 An Example Flow Proof in Concurrent Pascal

To illustrate the above concepts, we consider a generalization of the leak program presented in Figure 4.5. In this program, the array A is copied to the array B by using the ability to control relative process execution speeds. An outline of this program is given in Figure 6.8. The corresponding proof outline is presented in Figure 6.9. In this outline we use the class constant C to denote a bound on the information read into A.

Figure 6.8 Outline of a Program with Synchronization Flows

```
TYPE switch = MONITOR
    VAR ok : BOOLEAN; synch : QUEUE;
    PROCEDURE ENTRY wait;
    PROCEDURE ENTRY signal;

TYPE msg = MONITOR
    VAR store,  full      : BOOLEAN;
        sender, receiver : QUEUE;
    PROCEDURE ENTRY send   (b : BOOLEAN);
    PROCEDURE ENTRY receive(VAR b : BOOLEAN);

TYPE source = PROCESS(size : INTEGER; one, zero, done : switch)
    FOR i = 1 TO size DO
        IF A[i]
        THEN BEGIN
            one.signal; done.wait; zero.signal; done.wait
            END
        ELSE BEGIN
            zero.signal; done.wait; one.signal; done.wait
            END

TYPE destination =
PROCESS(size : INTEGER; done : switch; leak : msg)
    FOR i = 1 TO size DO
        BEGIN leak.receive(x); done.signal; B[i] := x;
              leak.receive(x); done.signal
        END

TYPE helper =
PROCESS(size : INTEGER;  bit : BOOLEAN;
        sig  : switch;  leak : msg)
    FOR i = 1 TO size DO
        BEGIN sig.wait; leak.send(bit) END

VAR one, zero, done  : switch;
    leak             : msg;
    sender.          : source;
    receiver         : destination;
    helper0, helper1 : helper;
BEGIN
    INIT leak, one, zero, done;
    INIT sender  (file_size, one, zero, done);
    INIT receiver(file_size, done, leak);
    INIT helper0 (file_size, false, zero, leak);
    INIT helper1 (file_size, true,  one,  leak)
END.
```

The proof of the switch monitor indicates that its shared variables ok and synch are bounded by the security class C. The proof shows that each operation of switch, as well as its initialization, preserves this invariant. Note that the global flow of a caller of the operation wait is increased by C. This is because termination of the wait is dependent upon the completion of the statement: IF NOT ok THEN DELAY(synch). Since ok may contain information of classification C, there can be an increase of the global flow by this amount. The post-condition global <= C • g captures this fact.

The proof of the monitor msg follows along the same lines. Here both operations may cause an increase in the global flow. Initially the shared variables are undefined, therefore their classification is low. As a result of monitor operations their classes may be increased; the invariant state indicates that these shared variables may contain information of classification C.

A proof outline for each process type is also given in Figure 6.9. Note that prior to each FOR loop the invariant for the loop is given. The invocation of monitor operations have a significant effect upon the proof of each process. Each monitor call must satisfy the operation's pre-condition. In addition, the effect of the call upon the var (result) parameters and upon the global flow must be

indicated in the post-condition of the invocation. In particular, the wait operation on switch monitors and the receive operation on msg monitors enable information of class C to be transmitted via global flow.

System initialization determines the number of each type of component in the system, as well as the parameters of each particular component. To certify an entire program, it is necessary to show that each component satisfies the security policy. In the program of Figure 6.8, each component only has constant initialization parameters (the monitor component parameters are essentially constants since they themselves are parameterless), thus the proofs of the types are sufficient. The policy that the classification of every object (variable) be bounded by C is satisfied by the program. In fact, no stronger (more restrictive) policy can be satisfied since the values in the array A are used to control process synchronization. The effect of this is to produce a global flow that transmits information of class C, where A <= C.

Figure 6.9 A Proof Outline for Program 6.8

```
CONSTANT file_size = 1000;

TYPE switch =
MONITOR

    " Let INV = ok <= C, synch <= C "

    VAR ok    : BOOLEAN;
        synch : QUEUE;

    PROCEDURE ENTRY wait;
    BEGIN
        { INV, local <= C, global <= C, global <= g }

        IF NOT ok THEN DELAY(synch);
        { INV, local <= C, global <= C, global <= g • C }

        ok := false
        { INV, local <= C, global <= C, global <= g • C } ;-

        { INV, local <= C, global <= C • g }
    END;

    PROCEDURE ENTRY signal;
    BEGIN
        { INV, local <= C, global <= C, global <= g }

        ok := true;
        { INV, local <= C, global <= C, global <= g }

        CONTINUE(synch)
        { INV, local <= C, global <= C, global <= g } ;-

        { INV, local <= C, global <= g }
    END;

    BEGIN
        { ok    <= low, synch <= low,
          local <= C, global <= C, global <= g }

        ok := false
        { ok    <= C, synch <= low,
          local <= C, global <= C, global <= g } ;-

        { INV, local <= C, global <= g }
    END;
```

Figure 6.9 (continued)

```
TYPE msg =
MONITOR
    " Let INV  =  store    <= C, sender <= C,
                  receiver <= C, full   <= C "

    VAR store, full       : BOOLEAN;
        sender, receiver  : QUEUE;

    PROCEDURE ENTRY send(b : BOOLEAN);
    BEGIN
        { INV, b <= C, local <= C, global <= C, global <= g }

        IF full THEN DELAY(sender);
        { INV, b <= C, local <= C,
          global <= C, global <= g • C }

        store := b; full  := true; CONTINUE(receiver)
        { INV, b <= C, local <= C,
          global <= C, global <= g • C } ⊢

        { INV, b <= C, local <= C, global <= g • C }
    END;

    PROCEDURE ENTRY receive(VAR b : BOOLEAN);
    BEGIN
        { INV, local <= C, global <= C, global <= g }

        IF NOT full THEN DELAY(receiver);
        { INV, local <= C, global <= C, global <= g • C }

        b    := store; full := false; CONTINUE(sender)
        { INV, b <= C • global, local <= C,
          global <= g • C, global <= C } ⊢

        { INV, b <= C • global, local <= C, global <= g • C }
    END;

    BEGIN
     { store <= low, sender <= low, receiver <= low,
       full <= low, local <= C, global <= C, global <= g }

     full := false
     { store <= low, sender <= low, receiver <= low,
       full <= C, local <= C, global <= C, global <= g } ⊢

     { INV, local <= C, global <= g }
    END;
```

Figure 6.9 (continued)

```
TYPE source =
PROCESS(size : INTEGER; one, zero, done : switch);

    VAR i : INTEGER;
        A : ARRAY [1..size] OF BOOLEAN;

    " Let P =  i <= C, size = low, A <= C,
               local <= C, global <= C "

    BEGIN
        { i = A = size = local = global = low }
        " assign information of class C to A "

        { i = size = local = global = low } ;-
        { i <= C, A <= C, size = local = low, global <= C }

        FOR i := 1 TO size DO

            { i <= C, A <= C, size = local = low,
              global <= C }
            IF A[i]
            THEN BEGIN
                 { P }
                 one.signal;
                 { P }
                 done.wait;
                 { P }
                 zero.signal;
                 { P }
                 done.wait
                 { P }
                 END
            ELSE BEGIN
                 { P }
                 zero.signal;
                 { P }
                 done.wait;
                 { P }
                 one.signal;
                 { P }
                 done.wait
                 { P }
            END
        { i <= C, A <= C, size = local = low, global <= C }

    END;
    { i <= C, A <= C, local <= low, global <= C }
```

Figure 6.9 (continued)

```
TYPE destination =
PROCESS(size : INTEGER;
        done : switch;
        leak : msg);

    VAR i : INTEGER;
        x : BOOLEAN;
        B : ARRAY [1..size] OF BOOLEAN;

    "Let P  =   i <= C, x <= C, B <= C, size = local = low,
                global <= C "

    BEGIN

        { i = x = B = size = local = global = low } |-

        { P }

        FOR i := 1 TO size DO

            BEGIN

                { P }
                leak.receive(x);

                { P }
                done.signal;

                { P }
                B[i] := x;

                { P }
                leak.receive(x);

                { P }
                done.signal;

                { P }
            END

        { P }
    END;
```

Figure 6.9 (continued)

```
TYPE helper =

PROCESS ( size : INTEGER;
          bit  : BOOLEAN;
          sig  : switch;
          leak : msg    );

    VAR i : INTEGER;

    " Let P =   i <= C, size = local = low, global <= C "

    BEGIN

        { i =  size = local = global = low } ;-

        { P }

        FOR i := 1 TO size DO

            BEGIN

                { P }
                sig.wait;

                { P }
                leak.send(bit);

                { P }

            END

        { P }

    END;
```

Figure 6.9 (continued)

```
"*****************************
 *                           *
 *       initialization      *
 *                           *
 *****************************"

VAR one  : switch;
    zero : switch;
    done : switch;
    leak : msg;

    sender   : source;
    receiver : destination;
    helper0  : helper;
    helper1  : helper;

" Let P =   one = zero = done = leak = < > "

BEGIN
    { one = zero = done = leak = low }

    INIT leak;
    { P }

    INIT one;
    { P }

    INIT zero;
    { P }

    INIT done;
    { P }

    INIT sender  (file_size, one, zero, done);
    { P }

    INIT receiver(file_size, done, leak);
    { P }

    INIT helper0 (file_size, false, zero, leak);
    { P }

    INIT helper1 (file_size,  true,  one, leak);
    { P }

END.
```

6.5 Additional Language Features

Although we have presented the flow semantics for most of Concurrent Pascal, there remain three types of language features whose effect we have not yet specified. Some features, such as the attribute function, provide a representation of what is normally a covert channel directly in Concurrent Pascal. For example, the value returned by the function call attribute(runtime) is the amount of time used by the calling process. This value gets updated as a side-effect of every statement. Thus after every program statement there is an implicit execution of the statement runtime := runtime + exectime, where exectime is the amount of time needed to execute the program statement.

In Chapter 3, 4, and 5 we viewed input and output as assignments from files to variables and vice versa. However, in Concurrent Pascal i/o is done at the device level; we associate a security classification with each device. All data transferred from a particular device is considered to be of one classification. Essentially we view devices as if they were arrays, except that there is no statement which can decrease the classification of the device. The result is that <u>all</u> files on a single disk are of the same classification.

The effect of exporting permanent class variables has also been disregarded. This language feature allows the

evaluation of these variables, but not their modification. Since each class is only accessible to one process or monitor, exported variables are equivalent to simple functions.

Chapter 7

Conclusion

7.1 Summary

In this thesis we have examined the problem of controlling the flow of information in computer systems. Our approach has been to certify programs with respect to a security policy, where the certification technique is based on the axiomatic method for program correctness.

The thesis developed an axiomatic logic for reasoning about information flow. The assertions of this logic describe a set of information states; the axioms and rules of inference describe how program statements transform these states. By using proofs in this flow logic, the flows generated by a program are determined without regard to any specific policy.

Certification is the process of reconciling a flow proof with an information security policy. A policy specifies, for each point in the program, the set of acceptable information states. A program is certified with respect to a particular policy by using a flow proof to show that the policy is maintained. The flow proof itself is completely independent of the security policy. In fact, the

certification of several different policies can be achieved using only one flow proof.

The logic was then extended to parallel programs. We found that flows arising from process synchronization are similar to those produced by statement non-termination. We developed axioms and rules of inference that allowed us to give flow proofs for programs with these constructs.

We also noticed the similarity between our flow logic and logics for functional correctness. Although we found that correctness logics alone are inappropriate for reasoning about information flow, flow and correctness logics together form a more powerful proof system. In the combined system, flow proofs can deal with individual array elements and disregard impossible execution paths. We feel that these aspects are important, since they enable proofs of real programs to be more exact. By reasoning about individual array elements, our flow techniques can be of use in practice.

As an indication of the feasibility of our approach, we developed a flow logic for the language Concurrent Pascal. Although there are more complicated features in Concurrent Pascal than we had previously considered, we found that the application of our ideas to the language was straightforward. A proof of a large program that contains subtle synchronization flows was also given.

7.2 Areas for Future Research

Although our flow logic enables a precise specification of policy, flow and certification, there remain some unsolved problems in information security. First, covert flows are disregarded in the logic we presented. We could handle them in our system by introducing a third auxiliary variable, covert, to capture these flows. Rules of inference for language constructs that allow conditional execution would be modified to indicate that the covert flow is increased by the class of the expression controlling the execution. Covert would then indicate the class of information available due to timing, such as through knowledge of program execution time. This is in fact the approach taken by Jones and Lipton [40]. However, we feel that this is a very crude approximation to covert flow, and would invalidate most programs not exhibiting the *-property. Additionally, not all covert flows arise from conditional execution. In many computer systems, power consumption by main memory depends upon the values it contains. Thus, reading a file into memory can cause a covert flow of the information contained in the file. Current techniques for attacking the problem of covert flow have not been successful.

Another unresolved problem concerns reasoning about files and other abstractions of memory. In our logic for

Concurrent Pascal, we associated one classification with an entire device. Clearly this is not acceptable when dealing with practical applications where many logical files reside on the same device. We need techniques to reason about the interface between computers and their external environment. Programming languages in which aspects of this environment can be represented abstractly are needed. This is a difficult problem, but one that must be solved if we are to have secure systems.

A final problem concerns all aspects of security. In the past, the techniques used for access control, information control, and functional correctness have been very different. This thesis has attempted to unify the latter two problems by using axiomatic techniques. Recent work in access control has also been moving in this direction. The unification of these three areas would greatly aid our understanding of security.

Bibliography

1. Anderson, J., "Computer Security Technology Planning Study," ESD-TR-73-51, Electronic Systems Division, AFSC, Bedford, Massachusetts, 1972.

2. Andrews, G.R., "Concepts and Conditions for Confinement," TR-75-241, Department of Computer Science, Cornell University, 1975.

3. Andrews, G.R., "COPS, A Mechanism for Computer Protection," TR-74-214, Department of Computer Science, Cornell University, 1974.

4. Andrews, G.R., "COPS, A Protection Mechanism for Computer Systems," TR-74-07-12, Computer Science Group, University of Washington, 1974.

5. Andrews, G.R., "Message Classes: An Approach to Process Synchronization," TR-76-275, Department of Computer Science, Cornell University, 1976.

6. Bell, D.E., "Secure Computer Systems: A Refinement of the Model," ESD-TR-73-278, volume III, Electronic Systems Division, AFSC, Bedford, Massachusetts 1974.

7. Bell, D.E., Burke, E.L., "A Software Validation Technique for Certification: The Methodology," ESD-TR-75-54, Electronic Systems Division, AFSC, Bedford, Massachusetts, 1975.

8. Bell, D.E., LaPadula, L.J., "Secure Computer Systems: A Mathematical Model," ESD-TR-73-278, volume II, Electronic Systems Division, Bedford, Massachusetts, 1973.

9. Bell, D.E., LaPadula, L.J., "Secure Computer Systems: Mathematical Foundations," ESD-TR-73-278, volume I, Electronic Systems Division, AFSC, Bedford, Massachusetts, 1973.

10. Boyer, R.S., Moore, J.S., "Proving Theorems about LISP Functions," JACM, 22, 1, 1975.

11. Brinch Hansen, P., "Structured Multiprogramming," CACM, 15, 7, 1972.

12. Brinch Hansen, P., "The Nucleus of a Multiprogramming system," CACM, 13, 4, 1970.

13. Brinch Hansen, P., "The Programming Language Concurrent Pascal," in IEEE Transaction on Software Engineering, Volume SE-1, Number 2, 1975.

14. Brinch Hansen, P., "The Solo Operating System: A Concurrent Pascal Program," Software Practice and Experience, 6, 1976.

15. Brinch Hansen, P., Operating System Principles, Prentice Hall, 1973.

16. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice Hall, 1977.

17. Cohen, E., "On the Synthesis and Analysis of Protection Systems," Operating Systems Review, 11, 5, 1977.

18. Cohen, E., "Strong Dependency: A Formalism for Describing Information Transmission in Computational Systems," Computer Science Technical Report, Carnegie-Mellon University, 1976.

19. Cohen, E., Jefferson, D., "Protection in the Hydra Operating System," Operating Systems Review, 9, 5, 1975.

20. Constable, R.L., O'Donnell, M.J., A Programming Logic, Winthrop, Cambridge, 1978.

21. Cook, S.A., "Axiomatic and Intrepretive Semantics For an Algol Fragment," Technical Report 79, Department of Computer Science, University of Toronto, 1975.

22. Denning, D.E., "A Lattice Model of Secure Information Flow," CACM, 19, 5, May, 1976.

23. Denning, D.E., "Secure Information Flow in Computer Systems," CSD-TR-145, Department of Computer Science, Purdue University, 1975.

24. Denning, D.E., Denning, P.J., "Certification of Programs for Secure Information Flow," CACM, 20, 7, 1977.

25. Denning, D.E., Denning, P.J., Graham, G.S., "Selectively Confined Subsystems," IRIA Workshop Notes, Rocquencourt, France, 1974.

26. Dennis, J.B., Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations," CACM, 9, 3, 1966.

27. Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages (F. Genuys, ed.), Academic Press, 1968.

28. Dijkstra, E.W., A Discipline of Programming, Prentice Hall, 1976.

29. Evans, A., Kantrowitz, W., Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," CACM, 17, 8, 1974.

30. Fenton, J.S., "Information Protection Systems," Ph.D. dissertation, University of Cambridge, 1973.

31. Fenton, J.S., "Memoryless Subsystems," The Computer Journal, 17, 2, May, 1974.

32. Good, D.I., London, R.L., Bledsoe, W.W., "An Interactive Program Verification System," IEEE Transactions on Software Engineering, SE-1, 1, 1975.

33. Graham, G.S., Denning, P.J., "Protection--Principles and Practice," Proceedings AFIPS. 1972 Spring Joint Computer Conference, Volume 40, 1972.

34. Harrison, M.A., Ruzzo, W.L., Ullman, J.D., "On Protection in Operating Systems," University of California, Berkeley, November, 1974.

35. Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," CACM, 12, 10, 1969.

36. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM, 17, 10, October, 1974.

37. Hoare, C.A.R., "Procedures and Parameters: An Axiomatic Approach," in Symposium on Semantics of Algorithmic Languages, Springer-Verlag, 1971.

38. Howard, J.H., "Proving Monitors," CACM, 19, 5, 1976.

39. Jones, A.K., "Protection in Programmed Systems," Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, 1973.

40. Jones, A.K., Lipton, R.J., "The Enforcement of Security Policies for Computation," in Operating Systems Review, 9, 5, 1975.

41. Lampson, B.W., "A Note on the Confinement Problem," CACM, 16, 10, October, 1973.

42. Lampson, B.W., "An Overview of the CAL Timesharing System," Computer Center, University of California, Berkeley, 1969.

43. Lampson, B.W., "Dynamic Protection Structures," AFIPS Conference Proceedings, 35, FJCC, 1969.

44. Lampson, B.W., "Protection," Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March, 1971, reprinted in Operating Systems Review, 8, 1, January, 1974.

45. Levin, R., Cohen, E., et al, "Policy/Mechanism Separation in Hydra," in Operating Systems Review, 9, 5, 1975.

46. Lipner, S.B., "A Comment on the Confinement Problem," in Operating Systems Review, 9, 5, 1975.

47. Lipton, R.J., Synder, L., "A Linear Time Algorithm for Deciding Subject Security," JACM, 24, 3, 1977.

48. London, T.B., "The Semantics of Information Flow," Ph.D. thesis, Department of Computer Science, Cornell University, 1977.

49. Millen, J.K., "Security Kernel Validation in Practice," CACM, 19, 5, 1976.

50. Milner, R., "Implementation and Applications of Scott's Logic for Computable Functions," SIGPLAN Notices, 7, 1, 1972.

51. Moore, C.G. III, "Potential Capabilities in Algol-like Programs," TR-74-211, Department of Computer Science, Cornell University, 1974.

52. Owicki, S., "Axiomatic Proof Techniques for Parallel Programs," Ph.D. thesis, Cornell University, 1975.

53. Owicki, S., Gries, D., "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica, Volume 6, 1976.

54. Owicki, S., Gries, D., "Verifying Properties of Parallel Programs: An Axiomatic Approach", CACM, 19, 5, 1976.

55. Purdy, G.B., "A High Security Log-in Procedure," CACM, 17, 8, 1974.

56. Ritchie, D.M., Thompson, K., "The Unix Time Sharing System," CACM, 17, 7, 1974.

57. Saltzer, J.H., "Protection and the Control of Information Sharing in MULTICS," CACM, 17, 7, July, 1974.

58. Shaw, A.C., et al., "A Multiprogramming nucleus with Dynamic Resource Facilities," Software Practice and Experience, 5, 1975.

59. Von Henke, F.W., Luckham, D.C., "Automatic Program Verification III: A Methodology for Verifying Programs," Stanford University Artificial Intelligence Lab Memorandum AIM-256, 1974.

60. Walter, K.G., Ogden, W.F., et al, "Primitive Models for Computer Security," ESD-TR-74-117, Electronic Systems Division, AFSC, Bedford, Massachusetts, 1974.

61. Wirth, N., "Modula: A Language for Modular Programming," Software Practice and Experience, 7, 1, 1977.

62. Wirth, N., "The Programming Language Pascal," Acta Informatica, 1, 1971.

63. Wulf, W., et al, "Hydra, the Kernel of a Multiprocessor Operating System," CACM, 17, 6, June, 1974.