AN AXIOMATIC APPROACH TO
INFORMATION FLOW IN PARALLEL PROGRAMS

by

Gregory R. Andrews

and

Richard P. Reitman

TR 78-361

Department of Computer Science
Cornell University
Ithaca, New York 14853

School of Computer and
Information Sciences
313 Link Hall
Syracuse University
Syracuse, New York 13210

This work was supported in part by NSF grant MCS 77-07554.

AN AXIOMATIC APPROACH TO INFORMATION FLOW IN PARALLEL PROGRAMS

Gregory R. Andrews

Cornell University

Richard P. Reitman

Syracuse University

ABSTRACT

This paper presents a new, axiomatic approach to information flow in sequential and parallel programs. Flow axioms that capture the information flow semantics of a variety of statements are given and used to construct program flow proofs. The method is illustrated by a variety of examples. The applications of flow proofs to certifying information flow policies and solving the confinement problem are considered. It is also shown that flow axioms and correctness axioms can be combined to form an even more powerful proof system.

__Keywords and Phrases__: information flow, information security, security certification, parallel programs, axiomatic logic, proof rules.

__CR Categories__: 4.32, 4.35, 5.21, 5.24.

This work was supported in part by NSF grant MCS 77-07554.
Author's Addresses: Professor G.R. Andrews, Department of Computer Science, Cornell University, Ithaca, NY, 14853. Professor R.P. Reitman, School of Computer and Information Sciences, 313 Link Hall, Syracuse University, Syracuse, NY, 13210.

AN AXIOMATIC APPROACH TO INFORMATION FLOW IN PARALLEL PROGRAMS

Gregory R. Andrews

Cornell University

Richard P. Reitman

Syracuse University

1. Introduction

A computer system is secure if the objects and information it contains can only be accessed in ways authorized by a security policy. A security policy has two components: an access policy and an information policy. An access policy specifies the actions that subjects, such as users, can take on objects, such as files. An information policy specifies the acceptable classifications of information in objects. For example, an access policy might specify that User 1 can read from File 1 and write to File 2; an information policy might specify that the information in File 1 be at most confidential and that the information in File 2 be at most secret. In order to validate the security of a system, one must verify that system programs accurately implement the given policy and verify that the underlying hardware and software protection mechanisms are correct. This paper focuses on the problem of ascertaining the flows of information within parallel programs. The interested reader is referred to [13,15] for discussions of validating access policies and to [14,17] for discussions of validating protection kernels.

We view a system as a large, parallel program in which variables are the objects that contain information, statements define actions, and processes are the subjects that take actions. Given a program that defines a specific system, an information policy is validated by solving the information flow problem, namely by determining how information flows between variables as a result of program execution.

The information flow problem has been examined previously by several researchers (see [3] for a survey). Some have advocated monitoring program execution at run-time [6,7,14] while others have defined the flow semantics of program statements in order to validate programs prior to execution [2,4,9,12]. Three papers in particular have made important contributions: Denning and Denning [4] defined flow semantics for sequential language statements and developed a certification procedure for a specific flow policy; Jones and Lipton [9] analyzed flows in a flowchart language and showed how to add checks to the program to validate a given policy; and Cohen [2] examined the information flow problem, again in sequential problems, from an information theoretic point of view. The above research has increased both our understanding of information flow and our tools for addressing the problem. It has, however, only dealt with information flow in sequential programs. Since information flow is of greatest concern in parallel programs, such as operating or data base systems, further work is required.

This paper presents a new approach to information flow that utilizes a program _flow proof_ constructed by applying flow axioms and rules of inference. There are four distinct advantages to this approach. First, it is powerful enough to capture the flows in parallel as well as sequential programs; in fact, flow axioms for all of Concurrent Pascal have been developed [18]. Second, once a flow proof for a program has been constructed, the same proof can be used to validate a variety of different information policies; the flow proof technique is not tied to any one type of policy (in contrast to [4,9,12]). Third, a flow proof is valid even if a program may not terminate; this is important given that many processes in operating and data base systems are not intended to terminate. Fourth, flow proofs combine in a straightforward way with correctness proofs; this leads to a proof system more powerful than one that only considers the classifications and not the values of variables. Subsequently, each of these points will be explained in detail.

The paper is organized as follows. Section 2 discusses basic concepts about information flow in programs and presents the concepts and notation used in flow proofs. Section 3 presents proof rules for sequential program statements, in particular assignment, alternation (<u>if</u>), iteration (<u>while</u> and <u>for</u>), and procedures. Section 4 presents proof rules for parallel execution (<u>cobegin</u>) and synchronization (semaphores and monitors). Section 5 discusses the use of flow proofs for certifying information policies and solving the confinement problem and describes how flow proofs and correctness proofs can be combined. Finally, Section 6 summarizes the approach.

2. Basic Concepts

With respect to information flow, a program has three components:

(1) <u>variables</u>, which contain information;

(2) an <u>information state</u> consisting of the current classification (i.e. degree of sensitivity) of each variable; and

(3) <u>statements</u>, which modify variables (and hence alter the state) or control the order of execution.

This section discusses information classification schemes, describes the ways in which statements alter variable classifications, and presents the concept of a flow proof.

2.1. Classification of Information

To characterize the sensitivity of information, each variable has a classification. Following the model in [3,4], we assume that the set of security classes is finite, is partially ordered ($\leq$), and has a least upper bound operator ($\oplus$). The notation $\underline{x}$ denotes the class of variable x. The ordering $\leq$ indicates the relationship between classes. For example, $\underline{x} \leq \underline{y}$ means that the information contained in variable y is at least as great as that in variable x. The least upper bound operator $\oplus$ is the means by which classes are combined. For example, if E is the expression x * y then the classification of the information in E, denoted $\underline{E}$, is by definition $\underline{x} \oplus \underline{y}$ since E contains information from both x and y. We will use low to represent the lowest classification and high to represent the highest.

Two specific classifications schemes are used in practice. The first assumes that classes are linearly ordered from low to high; it corresponds to the military classification scheme where low = unclassified and high = top secret. The second assumes that a distinct class is associated with each variable, that constants have class low, and that the other classes consist of the powerset of the variable classes and low. In this scheme, $\oplus$ is set union, $\leq$ is set inclusion, and high is the union of all the classes.

2.2. Flows of Information

The classifications of variables change as a result of executing statements. There are three types of information flow that can occur in parallel programs[1]:

(1) direct flows resulting from assignment and IO statements [3,4];

(2) local conditional flows within statements resulting from Boolean expressions controlling execution [3,4]; and

(3) global conditional flows between statements resulting from process synchronization and loops that may not terminate.

The last two types of flows are indirect in the sense that information flows from variables to other variables as a result of conditional execution.

To illustrate these flows, consider the following statements:

(1)	assignment	x:= E
(2)	alternation	if B then S_1 else S_2
(3)	iteration	while B do S
(4)	synchronization	wait(sem) signal(sem)

The assignment statement causes a direct flow of information from E to x, hence it causes the class of x to change. The new class of x is the least upper bound of the class of E

1 Covert flows [10] are being disregarded for now because they cannot be represented within a program; they are discussed and related to this approach in Section 5.2.

and the class of any conditional flow that affects execution of the assignment.

Alternation and iteration both cause a local flow of information from the guard (Boolean expression) B to the body of the statement. For example, in the statement

<u>if</u> x = 0 <u>then</u> y:= 0 <u>else</u> y:= 1

y is set to zero if and only if x is zero, hence y depends on x. In general, statements S_1 and S_2 both obtain information about B. The same type of flow occurs in iterative statements; the body S of a <u>while</u> loop receives information about the guard B. When alternative or iterative statements are nested, every nested statement obtains information from every guard controlling the nesting.

An iterative statement also causes a global flow from the statement to all subsequent statements. For example, the statement sequence

```
x:= 1;
while y = 0 do "skip";
x:= 0
```

causes x to be set to zero if and only if y is zero. This is because x:= 0 is executed conditionally, depending on whether the loop terminates.[2] In general, the information in the expression y = 0 flows to all statements that can be executed after the loop. Unless one assumes that all loops

2 In [4] this flow is ignored because all programs are implicitly assumed to terminate.

terminate (in parallel programs, many loops are not supposed to terminate), the possibility of this type of flow must be accepted.

The other source of global flow is process synchronization. If one process executes

```
if x = 0 then signal(sem)
```

and another executes

```
y:= 1;
wait(sem);
y:= 0
```

where sem is a semaphore with initial value zero, the final value of y indicates whether or not x is zero. This results from the fact that when a process blocks, it depends on another process to be awakened. All statements executed after the <u>wait</u> in the blocked process potentially receive information from the signaller. Although the above program fragment permanently blocks the second process if x is not zero, as will be shown, a global flow due to process synchronization can occur even if all processes are known to terminate.

2.3. <u>Flow Proofs</u>

The <u>information state</u> of a program consists of the <u>class</u> (classification) of the information in each variable. To represent restrictions on the set of possible states, assertions are used to indicate relationships between variables and classes. For example, the assertion $\{\underline{x} \leq \underline{y}\}$

represents the set of states where the information in x is no more sensitive than that in y. This is analogous to assertions that indicate relationships between variables and values in proofs of correctness.

To represent the flows arising from conditional execution and synchronization, two auxiliary (history) variables _local_ and _global_ are used. _Local_ contains the classification of the information available as a result of indirect flow within a statement; _global_ contains the classification of the information that may flow between statements as a result of loops and synchronization. In proofs of correctness, one often needs auxiliary variables to capture the "history" of statement execution; _local_ and _global_ serve an analogous role here since they capture the "history" of nesting, termination, and synchronization.

The information state of a program changes as a result of executing statements. Let P and Q be assertions about the information state and let S be a statement. The notation

{P} S {Q}

means that if P is true before execution of S then Q is true after execution of S, provided that S terminates. This is the same notation used in correctness proofs; the difference is that P and Q refer to classes rather than values. To develop a flow proof of {P} S {Q}, a deductive flow logic that describes the information flow semantics of statements is needed. The remainder of the paper presents and analyzes such a logic.

3. Proof Rules for Sequential Programs

This section first presents proof rules for sequential programs containing null, assignment, alternation, iteration, composition and procedure call statements. It then presents examples to illustrate the use of the rules. Throughout P and Q are assertions about the information state, and $P[x \leftarrow y]$ is the assertion P with every free occurrence of x syntactically replaced by y. The notation $P \vdash Q$ means that Q can be derived from P; the notation

$$\frac{A_1, \ldots, A_n}{B}$$

means that if logical statements $A_1, \ldots, A_n$ are true, then so is B. Occasionally assertions of the form {V,L,G} are used where the comma means conjunction (and), V is an assertion about the classes of program variables, L is an assertion about the class of auxiliary variable *local*, G is an assertion about the class of auxiliary variable *global*, and *local* and *global* only appear in L and G, respectively.

3.1. Null Statement

The null statement, *skip*, has no affect on the information state. If P is true before *skip*, then P is still true after *skip*. Accordingly we have the rule:

{P} **skip** {P}

3.2. Assignment

The assignment statement is the only statement being considered that can change the classification of a variable (the effect of IO statements is similar and is discussed in [18]). In the simple assignment statement x:= E, x receives information from three sources, as discussed in Section 2.2. First, information in E flows directly into x. Second, the assignment may be within an alternative or iterative statement in which case x receives information from auxiliary variable **local**. Finally, the assignment may follow a loop or synchronizing statement in which case x receives information from auxiliary variable **global**. (The ways in which **local** and **global** change will be discussed shortly.) Let P be an assertion that is to be true after executing x:= E. The proof rule for assignment then is

$$\{P[\underline{x} \leftarrow \underline{E} \oplus \underline{\textbf{local}} \oplus \underline{\textbf{global}}]\}\ \ x{:=}\ E\ \ \{P\}$$

Array element assignment of the form A[i]:= E is similar to simple assignment but has two additional complications. First, since the flow proof system only deals with classes of variables, not their values, one cannot reason about the value of i. Therefore, the class of an array must be at least as great as that of any of its element; assigning to A[i] cannot lower the classification of A. The second complication is that information from i also flows into A as a result of assigning to A[i]; for example, if A is initially all zeros and A[i]:= 1 is executed, the value of i can be determined by searching A. Given these considerations, the proof rule for array assignment is:

$$\{P[\underline{A} \leftarrow \underline{A} \oplus \underline{i} \oplus \underline{E} \oplus \underline{\mathrm{local}} \oplus \underline{\mathrm{global}}]\}\ A[i]:=E\ \{P\}$$

3.3. Alternation

The effect of the statement

if B **then** S_1 **else** S_2

is to make the information in B available to S_1 and S_2. In correctness logics, one has a rule of inference of the form:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

which clearly shows that B is a part of the pre-condition of the proofs of both S_1 and S_2. Analogously, a flow logic must ensure that the information in B is included as part of the pre-condition of flow proofs for S_1 and S_2. The variable local represents this type of flow; consequently, within the alternative statement, $\underline{B}$ must be "added" (using $\oplus$) to local. Let V and V' be assertions about the classes of program variables only, let L and L' be assertions about local, and let G and G' be assertions about global. The flow rule for alternation is:

$$\frac{\{V,L',G\}\ S_1\ \{V',L',G'\},\ \{V,L',G\}\ S_2\ \{V',L',G'\} \qquad V,L,G \vdash L'[\underline{local} \leftarrow \underline{local} \oplus B]}{\{V,L,G\}\ \underline{if}\ B\ \underline{then}\ S_1\ \underline{else}\ S_2\ \{V',L,G'\}}$$

If the local flow initially satisfies L, and L', after substituting local $\oplus$ B for local, can be derived from the initial state, then L' can be used in flow proofs of S_1 and S_2. After the alternation statement, the local flow once again satisfies L although V may have changed to V' and G may have changed to G'.

3.4. Iteration

Iterative statements can lead to three types of flow. First, the loop body can change the information state directly. Second, there is a local flow from the variables controlling the iteration to all repeated statements. Third, if the iterative statement may not terminate, there is a global flow from the variables controlling the iteration to all subsequently executed statements.

Consider the statement:

<u>while</u> B <u>do</u> S

To motivate the flow rule for <u>while</u>, consider its correctness rule:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{P \wedge \neg B\}}$$

This rule states that P must be invariant over S, that B can be used in the proof {P B} S {P}, and that $\neg$ B is true after the loop terminates (if it does).

The flow rule for *while* loops has three analogous parts. First, an assertion must be invariant over the execution of S. Second, the flow proof of the body, S, must take account of a potential increase in the local flow due to the guard B. Third, execution of statements after the loop is contingent on termination of the loop; hence the class of B flows into *global*. Let V be an assertion about the classes of program variables, let L and L' be assertions about *local*, and let G and G' be assertions about *global*. The flow rule for *while* then is:

$$\frac{\begin{array}{l}\{V,L',G\}\ S\ \{V,L',G\},\\ V,L,G \vdash L'[\underline{local} \leftarrow \underline{local} \oplus \underline{B}],\\ V,L,G, \vdash G'[\underline{global} \leftarrow \underline{global} \oplus \underline{local} \oplus \underline{B}]\end{array}}{\{V,L,G\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{V,L,G'\}}$$

Any flow caused by statement S is captured by the loop invariant {V,L',G}. Within S, the local flow is L' (derived from the initial state in the same way as for alternation) and the global flow is G (which must be at least as great as the global flow resulting from S). After the *while* statement, the local flow is again L but the global flow is changed to G'. Note that the global flow after the loop is at least as great as that before the loop and that it is affected by both B and local; local affects global because the while loop may itself have been nested within a conditional statement.

For an iterative statement that always terminates, the flow rule is slightly different because the global flow does not change. For example, the statement

for index:= start **to** finish **do** S

has the following flow rule (assuming that index is not modified by S):

$$\frac{\begin{array}{l}\{V',L',G\}\ S\ \{V',L',G\} \\ V,L,G \vdash V'[\underline{\text{index}} \leftarrow \underline{\text{start}} \oplus \underline{\text{finish}} \\ \qquad\qquad\qquad \oplus\ \underline{\text{local}} \oplus \underline{\text{global}}] \\ V,L,G \vdash L'[\underline{\text{local}} \leftarrow \underline{\text{local}} \oplus \underline{\text{start}} \oplus \underline{\text{finish}}]\end{array}}{\{V,L,G\}\ \textbf{for}\ \text{index:= start}\ \textbf{to}\ \text{finish}\ \textbf{do}\ S\ \{V',L,G\}}$$

In this rule, V' is the invariant about the variable state; it is derived from V, L, and G using a substitution comparable to that in the assignment axiom because index is implicitly assigned values between start and finish.

3.5. Composition and Consequence

To build programs, compound statements of the form

begin $S_1;\ldots;S_N$ **end**

are also needed. The flow rule for composition is identical to the corresponding correctness rule in which the post-condition of one statement is the pre-condition of the next. The rule is:

$$\frac{\{P_i\}\ S_i\ \{P_{i+1}\} \quad 1 \le i \le N}{\{P_1\}\ \textbf{begin}\ S_1;\ldots;S_N\ \textbf{end}\ \{P_{N+1}\}}$$

To develop flow proofs, especially for alternative and iterative statements, a rule of consequence is also needed to allow new proofs to be derived from existing ones. The rule of consequence for flow proofs is also identical to that for correctness proofs:

$$\frac{\{P'\}\ S\ \{Q'\},\ P \vdash P',\ Q' \vdash Q}{\{P\}\ S\ \{Q\}}$$

3.6. Procedures

As the final sequential programming construct, consider procedures of the form:

```
procedure pname (x; var y);
        <variable declarations>;
        S
```

where x is a sequence of value parameters, y is a sequence of value/result parameters, S is a statement, and no variable appears in both x and y[3]. The information flow rule for a call of the form

<u>call</u> name(a;b)

is again similar to the procedure call rule for correctness: there is a substitution of actual parameters for formal parameters, and, given a flow proof {P} S {Q} of the body of the procedure, Q with appropriate substitutions is true after the call given that P with appropriate substitutions is true before the call. In addition, there is a potential increase in the global flow in the calling program resulting from the fact that the procedure may not terminate.

A flow proof rule for procedures must capture its use of parameters and local variables and its contribution, if any, to global flow. Initially, each local procedure variable has classification <u>low</u> since it contains no information. Let g and c be new variables which will be used as placeholders for the global flow in the procedure. Then a flow proof of a procedure with body S has the form:

{P,<u>global</u> $\leq$ <u>g</u>} S {Q,<u>global</u> $\leq$ <u>g</u> + <u>c</u>}

where P and Q are assertions about the parameters, procedure variables, <u>local</u>, and <u>global</u>. Initially <u>global</u> is assumed to be no greater than <u>g</u>; <u>c</u> is therefore the procedure's con-

---

3 For simplicity, assume that there is no aliasing or recursion, that there are no own variables, and that all variable names are unique.

tribution to the global flow. This means that a valid call will increase the global flow in the calling procedure by <u>c</u>.

Given a flow proof of procedure pname(x;y), the <u>call</u> statement then has the following proof rule:

$$\frac{\{P, \underline{global} \le \underline{g}\}\ S\ \{Q, \underline{global} \le \underline{g} \oplus \underline{c}\}}{\begin{array}{l}\{P[\underline{x} \leftarrow a,\ \underline{y} \leftarrow \underline{b}],\ R[\underline{global} \leftarrow \underline{global} \oplus \underline{c}]\} \\ \quad \underline{call}\ \text{pname}(a;b) \\ \{Q[\underline{y} \leftarrow \underline{b}], R\}\end{array}}$$

where R contains no program variables that can be changed by S. This rule says that if Q and R are to be true after the call, P and R with appropriate substitutions must be true before the call. Notice that the rule makes no mention of <u>local</u>; assumptions about <u>local</u> used in the proof of S (e.g. <u>local</u> ≤ <u>low</u>) are stated in P and must therefore be proved to be valid at the points where the procedure is called.

3.7. <u>Examples</u>

The use of the above proof rules will now be illustrated by a series of examples of flow proofs. Each example pertains to the program shown in Figure 1, which sorts an array A[1:n] using the successive maxima sorting algorithm. The program successively finds the largest element of A (A[maxindex]) and swaps it with the first of the remaining, unsorted elements (A[first]). Variable i is used as a counter and temp is used for swapping.

Figure 1

Successive Maxima Sorting Program

```
begin
   first:= 1;
   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
```

Example 3.1

If local $\leq$ low and global $\leq$ low and constants are of class low, the following flow proof can be derived using the assignment rule:

{low $\oplus$ local $\oplus$ global $\leq$ low,

local $\leq$ low, global $\leq$ low}

first:= 1

{first $\leq$ low, local $\leq$ low, global $\leq$ low}

Since low $\oplus$ local $\oplus$ global $\leq$ low is a consequence of local $\leq$ low and global $\leq$ low, the more intuitive proof outline:

{*local* $\le$ *low*, *global* $\le$ *low*}

first:= 1

{*first* $\le$ *low*, *local* $\le$ *low*, *global* $\le$ *low*}

can be derived using the rule of consequence. Proof outlines for other assignments are derived in the same way.

Example 3.2

Consider the statement

if A[i] > A[maxindex] *then* maxindex:= i

Within the body of the *if* statement, the local flow increases by the class of the expression A[i] > A[maxindex], namely *A* $\oplus$ *i* $\oplus$ *maxindex*. Therefore the assignment of i to maxindex causes [illegible] to be potentially increased by *A* $\oplus$ *i* $\oplus$ *maxindex*, which means that *maxindex* cannot decrease. If *c* is an upper bound on *A*, it can be proved that

{*A* $\le$ *c*, *i* $\le$ *low*, *maxindex* $\le$ *low*, *local* $\le$ *low*, *global* $\le$ *low*}

if A[i] > A[maxindex]

{*A* $\le$ *c*, *i* $\le$ *low*, *maxindex* $\le$ *low*, *local* $\le$ *c*, *global* $\le$ *low*}

then maxindex:= i

{*A* $\le$ *c*, *i* $\le$ *low*, *maxindex* $\le$ *c*, *local* $\le$ *c*, *global* $\le$ *low*}

{*A* $\le$ *c*, *i* $\le$ *low*, *maxindex* $\le$ *c*, *local* $\le$ *low*, *global* $\le$ *low*}

using the rules for assignment, alternation, and consequence.

Example 3.3

Now consider the portion of the sort program that finds the largest element of sub-array A[1:n]:

```
while i ≤ n do
    begin if A[i] > A[maxindex] then maxindex:= i;
          i:= i + 1
    end
```

Let $V = \underline{A} \le \underline{c} \oplus \underline{n}$, $\underline{i} \le \underline{n}$, $\underline{maxindex} \le \underline{c} \oplus \underline{n}$. The following can then be derived using the previous example and the rules for assignment, iteration, composition, and consequence:

```
{V,local ≤ low, global ≤ low}
 while i ≤ n do
    {V,local ≤ n,global ≤ low}
    begin if A[i] > A[maxindex] then maxindex:= i;
          i:= i + 1 end
    {V,local ≤ n,global ≤ low}
{V, local ≤ low, global ≤ n}
```

Within the loop, $\underline{local}$ increases by $\underline{i} \oplus \underline{n}$, which implies $\underline{local} \le \underline{n}$. V is invariant over both statements in the loop. It is invariant over the if statement as can be seen by appropriate changes to Example 3.2; it is also invariant over the assignment since $\underline{i} \oplus \underline{local} \oplus \underline{global} \le \underline{n}$. In the assertion after the loop, $\underline{local}$ returns to $\underline{low}$ but $\underline{global}$ is increased by $\underline{local} \oplus \underline{i} \oplus \underline{n}$, which implies $\underline{global} \le \underline{n}$.

In the sort program, the above loop is embedded within another loop. Consequently, the local flow present at the start of the inner loop is actually *first* $\oplus$ *n* since first $\leq$ n is the guard of the outer loop. In addition, the global flow present at the start of the loop is actually *n* since the global flow produced by the inner loop must be invariant throughout the body of the outer loop. Therefore, in a proof of the sort program, the proof for the inner loop is:

{V, *first* ≤ *n*, *local* ≤ *n*, *global* ≤ *n*}
 while i ≤ n **do**
  begin if A[i] > A[maxindex] **then** maxindex:= i;
   i:= i + 1 **end**
{V, *first* ≤ *n*, *local* ≤ *n*, *global* ≤ *n*}

Example 3.4

By applying the techniques and results of the previous examples, the following proof for the sort program can be developed:

{*A* ≤ *c*, *local* ≤ *low*, *global* ≤ *low*}
"Sort Program of Figure 1"
{*temp* ≤ *n*, *maxindex* ≤ *n*, *i* ≤ *n*, *first* ≤ *n*,
A ≤ *c* $\oplus$ *n*, *local* ≤ *low*, *global* ≤ *n*}

In the sort program, there is a direct flow of information from A to temp, maxindex, and i within the bodies of the *while* loops. However, this flow is negated by unconditionally assigning 0 to these values after the array is sorted.

These variables may contain information about n, though, since termination of each loop depends on the value of n. Stated differently, this program "remembers" information about n but <u>not</u> about A. Had <u>for</u> loops been used instead of <u>while</u> loops, the program could have been proved memoryless [10] meaning that not even information about n could be retained in the local variables.

<u>Example 3.5</u>

As listed in Figure 1, the sort program is a distinct program. Suppose instead that it were a procedure of the form:

```
procedure sort (n:integer; var A:array 1..n of T);
        <declaration>
        <body as in Figure 2>
```

where T is some type. The proof outlined in the previous example is then a proof of the body of sort where

$P = \{\underline{A} \le c, \underline{local} \le \underline{low}\}$

$Q = \{\underline{temp} \le \underline{n}, \underline{maxindex} \le \underline{n}, \underline{i} \le \underline{n}, \underline{first} \le \underline{n}, \underline{A} \le \underline{c} \oplus \underline{n},$
$\underline{local} \le \underline{low}\}$

and the global flow increases by $\underline{n}$ in the proof rule for procedures.

4. Proof Rules for [illegible] Programs

In sequential programs, information is transmitted either directly via assignment or indirectly via conditional execution. The same types of flow arise in parallel programs. Processes can transmit information either directly by shared variables and message passing or indirectly by synchronization. Flow proof rules for a representative set of parallel programming constructs will now be presented.

4.1. Concurrent Execution

Consider the <u>cobegin</u> statement [5], which has the form:

> <u>cobegin</u>
>
> $S_1 \| \ldots \| S_N$
>
> <u>coend</u>

and means that each of the statements S_1 thru S_N are executed in parallel. Owicki and Gries [16] have shown that executing processes concurrently has the same effect as executing them sequentially provided their (correctness) proofs are <u>interference-free</u>. The same notion applies to information flow; as long as the flow proofs of two or more processes are interference-free, executing the processes concurrently results in the same flows as executing them sequentially. However, the definition of interference-free must be changed slightly because the auxiliary variables *local* and *global* exist separately for each process (the fact that one process is executing an *if* statement or *while* loop has no effect on flows in another process).

Definition: Given a flow proof $\{V,L,G\}$ S $\{V',L,G'\}$ and a statement T with pre-condition pre(T), T *does not interfere* with the proof provided that:

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

(2) For any statement S' in S with pre-condition $\{U,L,G\}$, $\{U, pre(T)\}$ T $\{U\}$

This says that executing T does not invalidate either the result of executing S or any pre-condition used in the proofs of statements within S.

Definition: Let T be an assignment statement within process S_i. The flow proofs $\{P_1\}$ S_1 $\{Q_1\}, \ldots, \{P_N\}$ S_N $\{Q_N\}$ are *interference-free* if for all j, $j \neq i$, T does not interfere with $\{P_j\}$ S_j $\{Q_j\}$.

The proof rule for the *cobegin* statement then is:

$$\frac{\{V_i,L,G\}\ S_i\ \{V_i',L,G'\}\quad 1 \le i \le N \text{ are interference-free}}{\{V_1,\ldots,V_N,L,G\}\ \textbf{cobegin}\ S_1 \| \ldots \| S_N\ \textbf{coend}\ \{V_1',\ldots,V_N',L,G'\}}$$

Since the local flow only changes within a statement, the final local flow is the same as the initial one. However, the global flow can change; after the ___cobegin___ statement it is at least as great as the global flow produced by any process.

4.2. Semaphores

For process synchronization, assume that semaphores [5] are available. A semaphore, sem, is a non-negative integer that can be manipulated by two indivisible operations, ___wait___ and ___signal___, defined as follows[4]:

___wait___(sem): ___while___ sem = 0 ___do___ ___skip___; sem:= sem - 1

___signal___(sem): sem:= sem + 1

Both wait and signal alter the value of sem, therefore they both cause a flow like that due to assignment statements. Although the ___signal___ operation completes immediately, ___wait___ can cause a process to delay, as shown by the presence of the ___while___ loop in its definition. Because completion of ___wait___ is dependent on another process executing a ___signal___, information can be transmitted by means of the semaphore from the signalling to the waiting process. All subsequently altered variables in the waiting process are affected, so the effect of ___wait___ is to increase the ___global___ flow in the process. Given these considerations, the proof rules for wait

---

4 Other definitions for ___wait___ and ___signal___ have the same effect upon information flow.

and signal are:

$\{P[\underline{\text{sem}} \leftarrow \underline{\text{sem}} \oplus \underline{\text{local}} \oplus \underline{\text{global}},$

$\quad \underline{\text{global}} \leftarrow \underline{\text{sem}} \oplus \underline{\text{local}} \oplus \underline{\text{global}}]\}$

<u>wait</u>(sem)

$\{P\}$

$\{P[\underline{\text{sem}} \leftarrow \underline{\text{sem}} \oplus \underline{\text{local}} \oplus \underline{\text{global}}]\}$

<u>signal</u>(sem)

$\{P\}$

In flow proofs, the class of a semaphore must be at least as great as the least upper bound of all the <u>local</u> and <u>global</u> flows that affect its value. Otherwise, the processes that use it will not have interference-free proofs.

4.3. Monitors

A monitor [1,8] is a collection of shared variables and the operations (procedures) that manipulate them. It has the form:

mname: <u>monitor</u>;
permanent variables;
<u>procedure</u> op_1( );

S_1 <u>end</u>; ...

<u>procedure</u> op_N( );

S_N <u>end</u>;

S_0

<u>end</u> mname

Operations $op_1,\ldots,op_N$ are mutually exclusive procedures (only one at a time may execute) with bodies $S_1,\ldots,S_N$. S_0 is the statement that initializes the monitor's permanent variables; it is executed before any of the operations can be called. Since processes sharing a monitor typically need to synchronize with each other, a monitor may contain queue variables, which are manipulated by <u>delay</u> and <u>continue</u> operations.[5] <u>Delay</u>(q) causes the invoking process to enter a queue associated with q and exit the monitor; <u>Continue</u>(q) allows one processing waiting on q, if any, to resume execution and causes the invoking process to exit the monitor.

The information flow resulting from monitors is a combination of the effects of procedures, synchronization, and the use of the permanent monitor variables. In order to develop a flow proof for a monitor, three steps are taken. First an invariant assertion, INV, that represents the "reasonable" states of the permament variables is stated.

5 Queue variables are the Concurrent Pascal [1] synchronization mechanism; the condition variables in [8] affect flow in the same ways.

Second, one proves that {INIT} S_0 {INV} is true where INIT states that the classification of each permanent variable is *low* (it initially contains no information) and INV is the invariant assertion. Third, a flow proof for each operation is developed. These take the form of procedure proofs with the added constraint that the invariant must be preserved. In particular, for each operation one needs a proof

$$\{P_i,\ INV,\ \underline{global} \le g\}\ S_i\ \{Q_i,\ INV,\ \underline{global} \le g \oplus c\}$$

where $1 \le i \le N$, S_i is the body of operation op_i, and P and Q state truths about the parameters and local variables.

In order to develop the proofs of monitor operations, proof rules for queue variables are needed. Information flows through queue variables in the same ways that information flows through semaphores. However, the proof rules for queues differ slightly from those for semaphores because one also needs to ensure that the monitor invariant is true before and after delay and continue. This leads to the following rules:

{INV,P[$\underline{q} \leftarrow \underline{q} \oplus \underline{local} \oplus \underline{global}$,
$\quad \underline{global} \leftarrow \underline{q} \oplus \underline{local} \oplus \underline{global}$]}
<u>delay</u>(q)
{INV,P}

{INV,P[$\underline{q} \leftarrow \underline{q} \oplus \underline{local} \oplus \underline{global}$]}
<u>continue</u>(q)
{INV,P}

Given a flow proof of a monitor, the effect of invoking a monitor operation is identical to the effect of invoking a procedure: there is a substitution of actual parameters for the formal parameters used in the proof of the operation and there is a potential increase in the global flow. The operation <u>call</u> rule is therefore the same as the procedure <u>call</u> rule shown in Section 3.6.

4.4. Example

As an illustration of information flow in parallel programs, consider the program shown in Figure 2. The program contains four processes, which interact by means of three semaphores and a message passing monitor. Even though the processes do not directly share variables, the sender process transmits information derived from its local variable x to the receiver process. It does so by using the semaphores as timing signals. If x is positive, sender signals the pos semaphore then the nonpos semaphore; otherwise sender signals nonpos then pos. Helper1 waits for a pos to be signalled then sends a message with value 1; helper2 waits for nonpos to be signalled then sends a message with value 0. The sender process also synchronizes with the helper processes by using synch, which ensures that the helper processes send their messages in the order determined by sender. Receiver receives two messages, one from helper1 and the other from helper2. Depending on the value returned in the last message, receiver learns whether or not sender's local variable x was positive or non-positive and prints the appropriate message.

Figure 2
A Program with Flows Due to Synchronization

```
begin
   var poa, nonpos, synch: semaphore initial (0,0,0);

   message: monitor;
      var free: Boolean; mail: (0,1);
          oktosend, oktorec: queue;

      procedure send(msg: (0,1));
         begin if ¬ free then delay(oktosend);
               mail:= msg; free:= false;
               continue(oktorec)
               end;

      procedure receive(var msg: (0,1));
         begin if free then delay(oktorec);
               msg:= mail; free:= true;
               continue(oktosend)
               end;

      free:= true
      end message;

cobegin
   sender: begin var x: integer;
              read value into x;
              if x > 0
                 then begin
                    signal(pos); wait(synch);
                    signal(nonpos) end
                 else begin
                    signal(nonpos); wait(synch);
                    signal(pos) end;
              wait(synch)
              end
     ||
   helper1: begin wait(pos); message.send(1);
              signal(synch) end
     ||
   helper2: begin wait(nonpos); message.send(0);
              signal(synch) end
     ||
   receiver: begin var result: (0,1);
              message.receive(result);
              message.receive(result);
              if result = 1
                 then print("non-positive")
                 else print("positive")
              end
   coend

end
```

This program is one example of the subtle ways in which synchronization mechanisms can be used to transmit information; it can in fact be expanded to transmit the exact value of x from sender to receiver (for example, by encoding x as a stream of bits). Note also that the program terminates. Global flows due to synchronization <u>always</u> occur; they are not dependent on deadlock or infinite loops.

To formally prove that the program in Figure 2 transmits information from x to result, flow proofs for the monitor and for each process are developed and the process' proofs are shown to be interference-free. In sender, assume that $\underline{x} \le C$ for some class C.

A proof for the monitor must show that each call of send or receive leaves the monitor in an invariant state. Using the techniques described previously for sequential programs, as well as the proof rules for delay and continue, it can be proved that

$$INV = \underline{free} \le C,\ \underline{mail} \le C,\ \text{[illegible]} \le C,\ \underline{oktorec} \le C,\ \underline{local} \le C$$

is a valid monitor invariant where C is an upper bound on the class of information passed into the monitor via its parameters. Given this invariant, it can then be proved that

{INV, *global* ≤ C} body of send {INV, *global* ≤ C}

{INV, *global* ≤ C} body of receive {INV, *global* ≤ C}

where C is an initial upper bound on the global flow present when send or receive is called.

Flow proofs for each process are developed by using the techniques of Section 3 as well as the proof rules for semaphores and monitors. Because the sender process signals both the pos and nonpos semaphores in a conditional statement guarded by x > 0, information from x flows into both semaphores. Assuming that *x* ≤ C, the classes of pos and nonpos also become as great as C. Therefore, the global flow in each helper process increases by C once it waits for pos or nonpos. These considerations lead to the following flow proof outline for helper1 (helper2 is similar):

```
{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ low}
     wait(pos);
{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ C}
     message.send(1);
{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ C}
     signal(synch);
{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ C}
```

In the above proof outline, the initial assertion might appear to be weaker than necessary because it asserts that each semaphore is bounded by C instead of *low*. The reason is that if the assertion [*pos* ≤ *low*, *nonpos* ≤ *low*, *synch* ≤ *low*, *local* ≤ *low*, *global* ≤ *low*) were used instead, it would not be possible to prove that the process' proofs are interference free. The same observation applies to the proofs for the other three processes.

The proofs for sender and receiver are similar to those for the helpers. The reader is encouraged to develop these and them to show that the following is a valid proof outline for the entire parallel program:

{pos ≤ low, nonpos ≤ low, synch ≤ low, local ≤ low, global ≤ low} $\vdash$

{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ low}

```
cobegin
  sender || helper1 || helper2 || receiver
  coend
```

{pos ≤ C, nonpos ≤ C, synch ≤ C, local ≤ low, global ≤ C}

The key to developing a flow proof for a set of processes is to ensure that the process' proofs are interference-free. For any parallel program, a flow proof can be developed as follows. First, examine each process to determine how its local variables and conditional flows affect the global variables and synchronization channels, where synchronization channels are semaphores or monitors. Second, for each global variable and synchronization channel, compute the least upper bound of the information that flows into the variable or channel. Third, construct a flow proof for each process using the classifications determined in the second step as the initial classifications of the global variables and synchronization channels. Finally, construct a proof for the set of processes (i.e. the cobegin statement containing them) and show that the process' proofs are interference-free. By following this approach, showing that the proofs are interference-free is relatively easy since the initial assertion used in each proof captures the greatest possible flow from the processes into global variables and synchronization channels.

4.5. Other Programming Constructs

Information flow axioms for other language constructs can also be developed using the techniques described above. In particular, flow axioms for IO statements, classes, case statements, message passing and critical regions have been developed. These are described in [18], which also contains a flow proof of a fairly large (6 page) Concurrent Pascal [1] program. The main concepts needed to produce every rule are the information classification scheme and the auxiliary variables local and global. In fact, capturing the effect of indirect flows in local and global is a key aspect of the axiomatic approach presented in this paper.

5. Applications of Flow Proofs

The main use of flow proofs is to certify information policies; this section first shows how to describe flow policies and how to use a flow proof to certify a policy. It then discusses the relationship between flow proofs and the confinement problem. Finally, it describes how the flow logic can be combined with correctness logic to yield an even more powerful proof system for reasoning about information flow properties of programs.

5.1. Certification of Information Policies

An *information policy* is a specification of the acceptable classifications of the information in variables. It is defined by means of an assertion about the information state (e.g. $\{\underline{x} \le \underline{y}\}$). Two types of policies are of particular interest. A *final value* policy is defined by an assertion that must be true when a program terminates; it states the acceptable result of a computation. A *high-water mark* policy is defined by an assertion that must be true for each information state in a program; it is used when intermediate results must be secure, for example when a program may not terminate.

Given a flow proof and a policy assertion, POLICY, a program can be certified as follows. First, identify every assertion in the program's flow proof where the policy must hold. Second, show that P $\vdash$ POLICY for every assertion P that holds at the points identified in the first step. The only flow proof assertion of interest in a final-value policy is the last one (the post-condition); all intermediate assertions are of interest in a high-water mark policy. If the policy cannot be certified, either a stronger flow proof or an alternative program must be considered.

Two examples will illustrate these concepts.

Example 5.1

In the sort program of Figure 2, a desired policy might be

$$\{A \le \underline{c} \oplus \underline{n},\ \underline{temp} \le \underline{n},\ \underline{maxindex} \le \underline{n},\ \underline{i} \le \underline{n}\}$$

where A[1:n] is the array to be sorted and initially $\underline{A} \le \underline{c}$ for some classification $\underline{c}$. This policy states that after A is sorted, information in A depends on A and n alone and that no information about A is to be stored in the local variables temp, maxindex, and i. The sort program flow proof outlined in Example 3.4 clearly shows that this final value policy holds. The progam does not, however, satisfy this assertion if it is used as a high-water mark policy since at times the local variables contain information from A, in particular within the bodies of the *while* loops. Note also that it is not possible to satisfy the final value policy $\{\underline{temp}=\underline{maxindex}=\underline{i}=\underline{low}\}$ since the potential non-termination of the two *while* loops produces a global flow from n. If a *for* loop had been used or if the while loop is proved to terminate (see Section 5.3), this stronger policy could be certified.

Example 5.2

The parallel program in Figure 2 can be certified with respect to the high-water mark policy $\{\underline{x} \le C,\ \underline{result} \le C\}$ but not with respect to the policy $\{\underline{x} \le C,\ \underline{result} \le \underline{low}\}$.

If the program were changed to include the assignment result:= 0 in the receiver process after the two calls to message.receive, then the program could be certified with respect to the final value policy {$x \leq C$, result $\leq$ low} since the program would always print "positive".

The other certification mechanisms that have been proposed [4,9,12] are special cases of the flow proof technique since they only deal with sequential programs and specific policies, and they disregard the possibility of non-termination. A more detailed analysis of the certification problem and comparison of the possible approaches are given in [19].

5.2. The Confinement Problem

The confinement problem [10] concerns ensuring that a module that performs a service neither retains its input nor transmits it to another program. There are two ways in which a procedure could leak information: by storage channels (i.e. program variables) or by covert channels (e.g. its execution time). The flow logic presented here deals with storage channels. Given a module such as a procedure, a flow proof captures the classificaion of the information in variables used by the procedure. Consequently, one can directly use a flow proof to check for confinement with respect to storage channels. For example, the sort program discussed in Section 3.7 does not retain information about the input array A but it is not confined since information about the input value n is retained in the procedure's local variables. However, if _for_ loops had been used instead of _while_ loops, the procedure would be guaranteed to terminate and one could therefore prove that it is confined with respect to storage channels.

Covert flows have been disregarded in the flow axioms since they are not observable by examining program variables. However, an upper bound on the potential covert flow accessible via timing channels, such as execution time, can be computed. Jones and Lipton [9] have observed that execution time varies as a result of conditional branching. Consequently, it is exactly the tests in a program (e.g. in _if_ and _while_ statements) that potentially transmit information over timing channels. If a third auxiliary variable, _covert_, is added to the proof rules and its class is increased whenever a conditional flow occurs (namely _covert_ is set to _covert_ $\oplus$ _local_ $\oplus$ _global_ in any proof rule that changes _local_ or _global_), then _covert_ encodes the maximum potential covert leakage due to timing. Unfortunately, this tells very little about most programs. As Lipner has observed [11], it is probably futile to attempt to eliminate covert channels.

5.3. Combining Flow Proofs and Correctness Proofs

A flow proof captures all (non-covert) flows *specified* by a program, but it also captures those arising from execution paths that cannot occur. For example, consider the program (also discussed in [2])

```
if b then y:= x;
if ¬b then z:= y
```

where b does not refer to x, y, or z. A flow proof for this program could show that $\{\underline{y} \leq \underline{x} \oplus \underline{b}, \underline{z} \leq \underline{y} \oplus \underline{b}\} \vdash \{\underline{y} \leq \underline{x} \oplus \underline{b}, \underline{z} \leq \underline{x} \oplus \underline{b}\}$ but could not show that z does not depend on x. This difficulty arises because flow proofs do not deal with variable values (i.e. b above), they only deal with variable classes. Another consequence of ignoring values in flow proofs is that the class of an array must be at least as great as the class of any element because one cannot differentiate between A[i] and A[j] without knowing the values of i and j.

These difficulties can be overcome by using a combined proof system that allows assertions about both classes and values. Each state in the combined system is the cross-product of an information state and a value state; the proof rules in the combined system describe how statements transform the composite state. For example, the assignment rule in the combined system is

$$\{P[x \leftarrow E,\ \underline{x} \leftarrow \underline{E} \oplus \underline{local} \oplus \underline{global}]\}\ x := E\ \{P\}$$

In the combined system, it can then be proved that the above program yields the result
$\{b \Rightarrow (\underline{y} \leq \underline{x} \oplus \underline{b} \wedge \underline{z} \leq \underline{low}),\ \neg b \Rightarrow (\underline{z} \leq \underline{y} \oplus \underline{b} \wedge \underline{y} \leq \underline{low})\}$
assuming that initially $\{\underline{y} \leq \underline{low},\ \underline{z} \leq \underline{low},\ \underline{local} \leq \underline{low},\ \underline{global} \leq \underline{low}\}$. This result shows that z depends on b, but not on x.

If the flow logic is combined with a logic for total correctness then global flows resulting from non-termination can also be avoided. The axioms for the *while* or other iteration statements would not affect *global* in the combined system; however, *global* would still be affected by synchronization statements, since it is merely the potential delay, not potential deadlock, that causes a global synchronization flow. The combined proof system, as well as an analysis of the relation between flow proofs and correctness proofs, is described in [19].

6. Summary

This paper has analyzed the semantics of information flow and presented a formal proof system for reasoning about information flow properties of programs. The resulting system unifies previous work on information flow and yields a number of additional benefits. First, a large number of both sequential *and* parallel programming constructs can be axiomatized. Second, since a flow proof is independent of any one policy, it can be used to certify a program with respect to a number of different policies. Third, the effects of non-terminating programs can be handled. Finally, the flow logic can be readily combined with correctness logic.

We have found that deriving a flow proof of a progam is much easier than deriving a correctness proof. This results from the fact that the algebra of the underlying classification domain is simple compared to the algebras underlying correctness proofs. Because flow proofs are relatively easy to construct and the flow proof system is powerful enough to handle a wide variety of constructs (including all of those of Concurrent Pascal), there is hope that information properties of large programs could be formally certified in the near future. To bring this hope to fruition, methods are needed to reason about files or other abstractions of memory and to ensure that the flow semantics of languages are in fact preserved during execution.

Acknowledgements

This paper has profited greatly from numerous discussions with David Gries, Carl Hauser, and Fred Schneider; they also provided very helpful comments on on an earlier draft.

BIBLIOGRAPHY

1. Brinch Hansen, P. The programming language Concurrent Pascal. <u>IEEE Trans. on Software Engr.</u> Vol SE-1, No. 2 (June 1975), 199-207.

2. Cohen, E. Information transmission in computational systems. <u>Proc. Sixth Symp. on Operating System Principles</u> (Nov. 1977), 133-139.

3. Denning, D.E. A lattice model of secure information flow. <u>Comm. ACM</u> 19, 5 (May 1976), 236-243.

4. Denning, D.E. and Denning, P.J. Certification of programs for secure information flow. <u>Comm. ACM</u> 20, 7 (July 1977), 504-513.

5. Dijkstra, E.W. Cooperating sequential processes. In <u>Programming Languages</u> (Ed. F. Genuys), Academic Press, New York, 1968.

6. Fenton, J.S. Memoryless subsystems. <u>Computer Journal</u> 17, 2 (May 1974), 143-147.

7. Gat, I. Security aspects of computer systems. Ph.D. Thesis, Technion-Israel Inst. of Tech., June 1976.

8. Hoare, C.A.R. Monitors: an operating system structuring concept. <u>Comm. ACM</u> 17, 10 (Oct. 1974), 549-557.

9. Jones, A.K. and Lipton, R.J. The enforcement of security policies for computation. _Proc. Fifth Symp. on Operating System Principles_ (Nov. 1975), 197-206.

10. Lampson, B.W. A note on the confinement problem. _Comm. ACM_ 16, 10 (Oct. 1973), 613-615.

11. Lipner, S.B. A comment on the confinement problem. _Proc. Fifth Symp. on Operating System Principles_ (Nov. 1975), 192-196.

12. London, T.B. The semantics of information flow. Ph.D. Thesis, Cornell University, January 1977.

13. McGraw, J.R. and Andrews, G.R. Access control in parallel programs. _IEEE Trans. on Software Engr._, Vol SE-5, No. 1 (Jan. 1979), 1-9.

14. Millen, J.K. Security kernel validation in practice. _Comm. ACM_ 19, 5 (May 1976), 243-250.

15. Neumann, P.G. _et. al_. A provably secure operating system: The system, its applications, and proofs. Final Report, SRI Proj. 4332, SRI International, Menlo Park, Calif., Feb. 1977.

16. Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs. _Acta Informatica_ 6 (1976), 319-340.

17. Popek, G.J. and Farber, D.A. A model for verification of data security in operating systems. Comm. ACM 21, 9 (Sept. 1978), 737-749.

18. Reitman, R.P. Information flow in parallel programs: an axiomatic approach. Ph.D. Thesis, Cornell University, August 1978.

19. Reitman, R.P. and Andrews, G.R. Certifying information flow properties of programs: an axiomatic approach. Proc. Sixth Symp. on Princ. of Prog. Lang. (POPL), January 1979.