******Submitted for Publication******

Technical Report CS 80011-R

# Language Extensions for Specifying Access Control Policies in Programming Languages

Billy G. Claybrook *

and

H. Rex Hartson

October 1980

Department of Computer Science

Virginia Polytechnic Institute and State University

Blacksburg, VA 24061

---

* Sperry Research Center, 100 North Road, Sudbury, Mass. 01776

# Language Extensions for Specifying Access Control Policies in Programming Languages

Billy G. Claybrook *
Sperry Research Center
100 North Road
Sudbury, Mass. 01776

H. Rex Hartson
Computer Science Department
Virginia Polytechnic and State University
Blacksburg, VA 24061

ABSTRACT

The scope rules in programming languages control the sharing of data among program units--e.g., blocks and procedures. Typically, scope rules provide an all-or-nothing kind of access control. A wide range of programming problems exist which require finer access control as well as considerable sophistication for the implementation of access control policies on high-level data objects such as files. This paper presents a number of language extensions that permit the programmer to specify the degree of access control for each abstract object that a program unit can manipulate. An attempt has been made to keep the number of extensions as small as possible and yet allow the user conveniently to specify the access control policies that he desires.

Some of the extensions permit access policies to be specified such that access correctness can be completely determined at compile time; other extensions permit policies to be specified that require some access checking to be done at runtime in order to ensure access correctness. The extensions have been developed such that subsets can be selected and implemented in programming languages to provide various access control policies.

---

* This research was sponsored by Army Research Office Contract DAAG29-80-C-0022 while the first author was a Visiting Associate Professor at the University of South Carolina, Columbia, SC.

# 1. INTRODUCTION

The scope rules in programming languages control the sharing of data among individual program units (blocks and procedures). Typically, languages (via their scope rules) provide an all-or-nothing kind of access control. In many applications finer access control is needed and the capability to implement a variety of access control policies, sometimes sophisticated, is desirable.

According to Denning and Denning [DENND79]: "Access controls regulate the reading, changing, and deletion of data and programs." There are two main kinds of access controls as classified by the agents they control: (1) those which control access by users (people), and (2) those which control access by programs. The motivation for this and related work is based on an observation made a decade ago [LAMPB71]: "...reasons for wanting protection are just as strong if the word 'user' is replaced by 'program'." However, there is still a distinction between the two cases.

Access controls that govern access by users can vary from user to user. As such, they are suitable for applications involving, for example, security and integrity in which different users must be given different access rights. Access controls of users are the type used by operating systems and database systems, and they involve an external authorization process--the dynamic granting and revocation of access rights to users.

On the other hand, access control policy controlling access by a program is placed (possibly by capabilities) into the program more or less permanently. No capabilities can be passed in from anywhere external to the execution environment. These controls are created by the programmer and are

typically independent of the user (the person on behalf of whom the program is running). That is, they behave the same way regardless of which person is using that program. As such, they are most useful for preserving integrity—for example, for preventing a malfunctioning program from damaging another program's objects. The contemporary approach to implementation of data structures, using data type abstraction, is a good example of integrity preserving controls. Another example is the way in which the scope rules of programming languages control access to variables in various program units (blocks). It is this latter kind, the integrity preserving controls which control access by programs, that are addressed in this paper.

The primary aim of this paper is to present a number of language extensions that permit the programmer to specify the degree of access control for each abstract object that a program unit can manipulate. An attempt has been made to keep the number of extensions as small as possible and yet allow the user conveniently to specify the access control policies that he/she desires. The extensions are suitable for specifying access control policies in both applications and systems programs.

Access control policies specified such that access correctness can be determined entirely at compile time do not require validity checks at runtime and, hence, do not incur any runtime overhead. On the other hand, the greatest flexibility in specifying access control policy is achieved by providing the validation of access requests at runtime. While the user can specify access policies such that access correctness can be completely determined at compile time, no attempt is made to restrict the discussion to only such extensions (e.g., Jones and Liskov [JONEA78]). Instead, extensions are described that also provide the expressive power required to spe-

cify policies for dynamic systems, such as file systems, in which objects and access paths are created and deleted and rights to objects can be changed dynamically.

McGraw and Andrews [MCGRJ79] state that access control mechanisms should adhere to two principles:

1) expressive power - they should allow a wide variety of access policies to be expressed clearly and exactly, and
2) access validation - they should allow the implementation of an access policy to be validated.

Extensions introduced here support both of these principles.

An important ingredient of any access control facility is the binding rule. Binding causes a variable (a capability variable in this paper) to refer to an object by storing a reference to the object in the variable. In particular, the binding operation x <-- y causes the capability variable x to reference the object already referenced by capability variable y, creating a new access path to the object. Most access control facilities provide a single binding rule. For example, Jones and Liskov [JONEA78] provide a binding rule in which binding is legal provided access rights to an object are not increased. Others, e.g. McGraw and Andrews [MCGRJ79], Ekanadham and Bernstein [EKANK79], Wulf, et. al. [WULFW74], use a binding rule in which the caller determines the access rights of the called procedure during procedure invocations. In addition, most access control mechanisms provide some form of amplification [JONEA73], in which, in certain situations, the access rights to an object can be increased.

In general, a binding rule determines to some extent how much control is afforded the programmer for passing access rights to data objects among

program units. Because a wide range of access control policies are needed for manipulating high level data objects, a single binding rule is not sufficient. Three different binding rules are permitted for use in specifying access control policies (these are discussed in detail in Section ). Thus, in addition to specifying access rights to data objects, specification of access control policy involves specifying which binding rules should be used during binding operations.

In addition to the authors of the main references in this paper, a few others have investigated protection in programming languages. An early integration of protection into programming languages was done by Morris [MORRJ73]. In this work, objects can be sealed and passed from module to module. The module which seals an object is the only one that can unseal it. Other modules can access the object only by calling that module which did the sealing. Malfunctions in other modules, then, cannot damage the object by accessing it directly.

Ambler and Hoch [AMBLA77] also recognize this same approach which, in its more recent--and more general--form, is the foundation of the abstract data type approach to object type implementation: "...never pass direct access to a protected object, but rather pass access to operations on that object." Ambler and Hoch provide a comparative examination of Pascal, Euclid, Clu, and Gypsy and their application to the solution of an interesting exercise in security--the Prison Mail Problem.

The remainder of this paper describes the syntax and semantics of the language extensions for specifying access control policy. These extensions provide the capability to:

1) specify access rights to data objects (either at compile time or runtime),

2) confine or limit the visibility of data objects in program units by restricting the scope rules,

3) specify the binding rule to be used in each binding operation, and,

4) specify whether or not a capability can be exercised by a program unit.

Item (1) above is discussed in Sections , , and The confine declaration for limiting the visibility of data objects in discussed in Section . The semantics of the binding rules are defined in Section and conditional capabilities (item (4) above) are briefly discussed in Section .

## 2. MODIFYING SCOPE RULES

The first step in providing finer access control to data objects involves modifying the scope rules. This involves controlling the objects that each program unit sees. In other words it is desirable for the capability to limit the visibility of data objects. McGraw and Andrews [MCGRJ79] use the grant declaration to fully replace the scope rules. A program unit must be granted access to an object that is defined in another program unit, otherwise, access to the object is prohibited. Note, however, that grant must be applied at successive levels to pass an object to an inner program unit.

This approach is analogous to a "closed system" of access control [DALER65] in which all allowed accesses must be explicitly granted. This has been shown to be the most secure approach in a highly dynamic authorization system, because an accidentally omitted grant will not allow unauthorized accesses. However, in the more stable programming language environment where long term integrity is the issue, the explicit grant declaration is too cumbersome for many applications. Instead, therefore, normal Algol-like scope rules are permitted to apply here, unless they are altered by the confine declaration. The confine declaration has two forms:

1) confine <object list> to <program unit list>; and

2) confine <object> rights {rights list} to <program unit list>;

The rights specification in the second form of the confine specifies the rights that the listed program unit(s) have to <object>. In this paper a right is represented by the name of the operation that it allows to be performed on an object (e.g., read, write, copy). Any object specified in a confine declaration automatically has its scope confined to the program unit(s) specified in the declaration.

A confine declaration can be applied only to those objects that the program unit (in which the confine declaration resides) defines or inherits via another confine declaration. The program unit(s) in a confine declaration must be defined in the program unit containing the declaration. The visibility of an object referenced by variable t can be limited to the program unit, say XPROC, in which the object is defined by using a confine of the form

confine t to XPROC;

The confine declaration does not prevent an object from being passed as a parameter to another procedure and confine can also be applied to formal parameters to limit access to objects passed from a calling procedure. The confine declaration is appropriate for objects referenced by what is referred to as "normal" variables (see Section ) but is intended primarily for capability variables.

The example in Figure 1 illustrates the use of the confine declaration. The object referenced by variable x, defined in procedure T, has its visibility confined to procedure R (x is also visible in T since it is defined in T). Variable x would have been visible in all of the procedures in Figure 1, if the confine on x has not been used. Even though variable x is not visible in procedure V, the object referenced by x can be manipulated in procedure V, because x is an actual parameter to the call to procedure S. The corresponding formal parameter, a, in S is restricted to V. Variable a in S is not visible in procedure U as it would normally be.

The program units that can be used in the <program unit list> in a confine declaration in procedure T are limited to procedures Q and R (as well as T itself) since they are the only program units directly defined by T. The "confine y to T;" declaration limits the visibility of variable y to T; thus eliminating y as a global variable to the other procedures in Figure 1. On the other hand, z is global to all procedures in T.

As illustrated in Figure 1, the confine declaration can be used to alter the scope rules of a language when desired. The scheme is relatively simple yet expressive enough to specify a wide range of access policies not handled by scope rules or other existing mechanisms. Inclusion of the confine declaration in a programming language requires slight modification of the symbol table routines within the compiler.

```
T: procedure;
     x,y: integer;
     z: real;              /* z is global to whole figure */
     confine x to R; /* x also visible in T, where it
                                                 is defined */
     confine y to T; /*prevents y from being global to
                       whole figure*/
          .
          .
          .
     R: procedure;
          .
          .
          .
          S: procedure (a);
               a: integer;
               confine a to V;
                    .
                    .
                    .
               V: procedure; /* x not visible in V, but
                    .          object referenced by x
                    .          can be manipulated here
                    .          because x is actual par-
                    end V;       ameter of call to S */
               U: procedure;
                    .
                    .
                    .
                    end U;
             .
             .
             .
             end S;
          call S(x);
             .
             .
             .
          end R;
     Q: procedure;
          .
          .
          .
          end Q;
     end T;
```

Figure 1. Using the confine declaration

## 3. CAPABILITY VARIABLES

In conventional Algol-like languages, variables hold values rather than pointers (or references) to values. These variables are referred to as "normal" variables and the variables that hold references to values are referred to as capability variables. A capability is essentially a ticket to access a data object, and if a program unit has the ticket then it has access to the data object referenced by the ticket. A conventional capability contains a type field holding the type of data object referenced, a list of access rights to the object referenced and a reference to the data object. In addition to the three items listed immediately above, a capability, in this paper, may contain one or more binding rules that can be applied to the capability variable during binding and a list of locks that can control the use of the capability by program units. This generalized capability variable is illustrated in Figure 2.

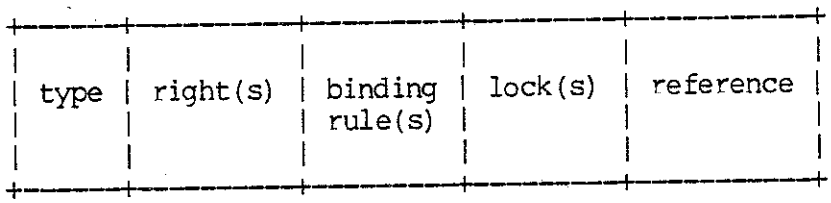


Figure 2. Capability variable

In Jones and Liskov's [JONEA78] system, all data objects are assumed to be referenced by capability-like variables. In this paper, access to objects of built-in types is assumed to be via normal variables, whereas access to objects of programmer defined abstract data types (that is, types

specified and implemented via an encapsulation mechanism such as Claybrook's module [CLAYB79], [CLAYB80]) is through capability variables. Each abstract data type* specified and implemented by the module has a rights specification in it which gives the maximum set of rights available for the programmer to use. The built-in types have a predefined set of primitive operations.

There is no confusion of normal variables with capability variables, since a compiler can easily make the distinction. The use of normal variables to access objects of built-in types does not limit the utility of this access control facility, since normal variables can be replaced with capability variables to reference built-in types as well as programmer defined types. This would mean of course that primitive objects would also be referenced with capability variables. This paper will concentrate on access to programmer defined types.

If "xlist" is a programmer defined type then the declaration

x: xlist capability;

declares x to be a capability variable to type xlist; that is, x is an access path to objects of type xlist. The capability attribute can be omitted from declarations such as the above because it is easy for a compiler to differentiate between built-in types and programmer defined types; therefore, it will be omitted from some of the declarations in the remainder of this paper.

Instances of type xlist are created by executing a "createlist" operation. In this paper it is assumed that each abstract data type has a create-like operation specified for it in the module defining the type, and

---

* The terms "data type" and "object type" are used interchangeably in this paper.

that this create operation creates an instance of the type with full access rights. The statement

x.createlist(); *

causes an instance of data type xlist to be created (with full rights) and a reference to it placed in capability x. Initially, a capability variable is empty in the sense that it does not reference anything; however, it still can initially contain access rights when they are explicitly specified in a declaration as in:

y: xlist capability rights {insert,delete};

The rights specified in the declaration of y must be a subset of the full rights specified in module (the type implementation module of) xlist.

As mentioned earlier, a capability variable can have the following attributes: rights, binding rules, and locks. In their most general form, capability variable declarations would specify whether each of the attributes is static or dynamic. For convenience, a convention is adopted that retains this generality while not unduly encumbering the programmer. The convention is that attributes are static if they have declared initial values. Attributes without declared values are dynamic. None, any, or all of the capability variable attributes can be initialized at declaration time. Static attributes can never be altered during execution, while dynamic attributes can be assigned values during program execution. To illustrate the difference between static and dynamic attributes, consider the create statement

y.createlist();

---

* In this paper, an operation on an object referenced by a capability variable will be denoted either by a qualified notation of the form "variable.operation" or by the notation "operation(variable);" both forms are used in Alphard [WULFW76].

This statement creates an xlist instance with full access rights and places a reference to it in y, whose declaration is given above. But capability variable y does not receive full rights (as variable x did above), because y has static access rights, "insert" and "delete," while the binding rules and locks attributes are dynamic.

Jones and Liskov [JONEA78] require that access rights be declared at compile time and thus provide only for static access rights. When only static attributes are used to specify policy, then access corrrectness can be determined entirely at compile time. A disadvantage, however, of static attributes is that sometimes there is a need for different sets of attributes for a single object. In Jones and Liskov's system this means at least one new capability variable must be declared for each set of access rights (the only attribute in their facility is rights), whereas, if dynamic attributes are available, a single capability can be used. In McGraw and Andrews' [MCGRJ79] system, access rights are associated with a capability variable, not at declaration time, but when an instance of an object is created or a binding operation is executed. Access policies specified using their language extensions are quite flexible, but they lose the capacity to determine access correctness entirely at compile time. By means of both static and dynamic attributes in the language extensions described in this paper, runtime determination of access correctness is allowed, but used only if the access policy requires it. In other cases access correctness can be decided completely at compile time.

## 4. QUALIFIED TYPES

In this paper the concept of "qualified type" is used, as it is by Jones and Liskov [JONEA78], to facilitate the definition of binding rules and to specify when a binding is legal. The concept is used here for the same purposes. A simple qualified type x is specified as

T{r1,...,rn}

and has two parts: a type part* and a rights part, i.e.,

type(x) = T, and

rights(x) = {r1,...,rn}

Definition 1. If x and y are two simple qualified types then x = y iff

1) type(x) = type(y), and

2) rights(x) = rights(y).

In addition, $x \leq y$ iff

1) type(x) = type(y), and

2) rights(x) $\subseteq$ rights(y).

Further, x and y are incomparable iff type(x) $\neq$ type(y).

In general, a qualified type x is defined recursively as:

1) A simple qualified type, or

2) T[Q1,Q2,...,Qm] {r1,...,rn}

where each Qi, i = 1,...,m, is a qualified type.

---

*Jones and Liskov [JONEA78] refer to this as a "base type" part.

In this paper the objects described by nonsimple qualified types are synonomous with structured objects. A qualified type is essentially a hierarchical structure. The Qi are components of the structured object, each with its own set of rights called "component rights." The set of rights {r1,...,rn} in the qualified type x is the rights to the structure (as a whole) and are called "structure rights."

An example of a qualified type is a file composed of records, whose qualified type declaration might be as follows:

file [record rights {readsalary}] rights {insertrec, deleterec}

where {readsalary} comprises the component rights (the salary field may be read within any record). The structure (file level) rights, {insertrec, deleterec}, indicate operations allowed on the overall structure.

The functions type(x) and rights(x) still apply to non-simple qualified types as follows:

$$TYPE(x) = T$$

$$RIGHTS(x) = \{r1,...,rn\}$$

The upper case function names denote functions that yield the structure type and the structure rights, respectively. The three items in Definition 2 below are each a pairwise comparison of the nodes in the hierarchies x and y. This requires the elements of sets A and B, defined as follows, to be generated in the same order.

Let $A = \{t \mid t = x \text{ or } t = \text{components } (s) \text{ such that } s \in A\}$

and $B = \{t \mid t = y \text{ or } t = \text{components } (s) \text{ such that } s \in B\}$.

Definition 2. a) $TYPE(x) = TYPE(y)$ iff:

$type(a) = type(b)$ for all a, b such that $a \in A$ and $b \in B$

b) RIGHTS(x) = RIGHTS(y) iff:

rights(a) = rights(b) for all a,b such that a∈A and b∈B

c) RIGHTS(x) $\leq$ RIGHTS(y) iff:

rights(a) $\subseteq$ rights(b) for all a,b such that a∈A and b∈B.

Similarly,

TYPE(x) = {type(a) | a∈A}

RIGHTS(x) = {rights(a) | a∈A}

The lower case function names denote functions that yield attributes of individual components.

Definition 3: Let x and y be two qualified types

x: T[Q1,Q2,...,Qm]{r1,...,rn}

y: T'[Q1',Q2',...,Qm']{r1',...,rk'}

then x $\leq$ y iff:

1) TYPE(x) = TYPE(y),

2) RIGHTS(x) $\leq$ RIGHTS(y)

The next section discusses the use of qualified types to describe the three binding rules proposed for the language extensions.

## 5. BINDING RULES

As stated in section 1, a single binding rule is not sufficient for providing the convenience and flexibility required to express a wide range of access control policies. Thus, we permit the programmer to specify the rule to be used during binding. This is a departure from other systems where there is usually a single binding rule in effect and the programmer has no control over it. Permitting the programmmer to specify a binding rule is analogous to the process of giving the user the ability to specify how parameters are to be passed during procedure invocation in conventional programming languages. For example, Algol permits parameters to be passed by name or by value, and the programmer can specify which.

The three binding rules are referred to as

1) subset,

2) amplify, and

3) domtrans

The subset rule is essentially the binding rule that Jones and Liskov propose in [JONEA78] . It does not permit the access rights to an object to be increased during binding. The amplify rule is used (primarily in procedure invocation) where the called procedure must have greater rights to an object than those of the caller. The domtrans rule transfers all rights of a capability already referencing an object to another capability which is a new reference to the object (regardless of what the new capability's rights were prior to binding). Detailed definitions of these rules are given in the next section.

## 5.1. Binding rule evaluation

As previously stated, the binding operation x <-- y causes the capability variable x to reference the object already referenced by capability variable y, creating a new access path to the object.

Two things must be determined during binding operations:

1) the one binding rule that is applicable, and
2) whether the binding operation is legal with respect to the rule determined in (1).

The second of these is the simplest to discuss, and it is covered first in this section. In systems such as Jones and Liskov's, McGraw and Andrews', etc., to determine whether or not a binding operation can be completed (or is legal), it is sufficient to determine if the capability variables involved in the binding operation satisfy the binding rule. For example, in Jones and Liskov's mechanism, the binding x <-- y is legal if $x \leq y$ (see Definitions 1 and 3 in Section 4 for the meaning of $x \leq y$).

Since the manifestation of qualified types is in the corresponding capability variables, the attributes of qualified types will be referred to as though they were attributes of capability variables as well. In particular, we wish to apply the type() and rights() functions to capability variables. Doing so merely simplifies the notation. The specification of a qualified type, then, and the declaration of a capability variable are identical.

For generality purposes in discussing the binding rules below, the capability variables x and y are assumed to represent the qualified types

x: T[Q1[Q2 [...[Qn]...]]] rights {r1,...,rk}

y: T[Q1'[Q2'[...[Qn']...]]] rights{r1',...,rm'}, respectively.

The validity of a binding operation for each binding rule is, then, summarized as follows:

1) subset: x <-- y is legal iff x $\leq$ y (by Definition 3)

   [Result: The rights to y's object cannot be increased by accessing via x.]

2) amplify: x <-- y is legal iff

   a) TYPE(x) = TYPE(y)

   b) checkrights(x) $\subset$ rights(y)

      [Result: The right to y's object can be temporarily increased.]

3) domtrans: x <-- y is legal iff TYPE(x) = TYPE(y)

   [Result: The variable x takes on the rights of y.]

Note: For the binding x <-- y to be legal with the domtrans rule the capability variable x must be declared with no rights specified; for example:

x: file [record]

so that the rights of x is a dynamic attribute by the conventions of Section 3.

As an aside, it is interesting to consider a special case of the subset rule, which can be called the equal rule. The requirements of this rule are:

1) TYPE(x) = TYPE(y)

2) RIGHTS(x) = RIGHTS(y)

The rule would seem to have its application limited to situations in which it is desired to retain information about rights over a sequence of related bindings. Thus, while the equal binding rule does not allow an increase of rights, it also does not allow a decrease.

5.1.1. The subset binding rule

The subset rule does not permit the rights to data objects to be increased during a binding operation. For binding operations involving structured data objects using the subset rule, it may appear that the rule need only be applied to the object as a whole and to the component objects that are explicitly manipulated (say within a procedure).

However, even if component objects are not explicitly manipulated, they may need to be specified to check the validity of either or both of the conditions during a binding operation under the subset rule:

1) TYPE(x) = TYPE(y)

2) RIGHTS(x) $\leq$ RIGHTS(y)

where x and y are qualified types (see Definition 3). Component objects may have to be declared even though the components are not explicitly manipulated, say, in a called procedure. This is because programmer defined types can be parameterized and the component objects of a data type may vary from one instantiation of the type to another. For example, a parameterized stack data type may be instantiated with integer elements one time and real elements the next. In other cases in which components are not explicitly manipulated, their rights must be stated to permit condition (2) above to be satisfied.

In comparison, Jones and Liskov do not allow for separate application of the rule for structure and components of a structure.

For the binding operation

```
x <-- y;
```

to be valid they require that all the component parts of x and y be pairwise identical. For $x \leq y$, Jones and Liskov require that:

1) TYPE(x) = TYPE(y),

2) rights(x) $\subseteq$ rights(y), and

3) RIGHTS(Qi) = RIGHTS(Qi'), for i = 1,n.

The subset binding rule proposed here is similar but not exactly identical to the binding rule in [JONEA78]. The difference between the present subset rule and their rule is item (3) above. (See Definitions 1 and 3 of $x \leq y$, which allow for pair-wise checking of rights(Qi) $\subseteq$ rights(Qi').)

In order for their binding rule to apply to structured data objects and to permit a procedure to declare precisely what limited rights that it requires to component objects during procedure invocation, they introduce the "?types" concept. The symbol "?" indicates that the exact rights of that type are not known, but must satisfy a subset constraint given in the formal parameter declarations. The exact rights are then known at invocation time, when matched with the actual parameters. The use of ?types in a formal parameter position permits the rights to component object types to be decreased and yet permits the binding rule to be legal.

5.1.2. The amplify binding rule

Sometimes, when a procedure is called to perform an access operation on an object, the procedure needs greater rights to the object than those of the caller. This occurs, for example, when the object is an occurrence of an encapsulated data type and the called procedure that implements operations in the type definition must manipulate the details of the object's

representation in storage. Such representation details are not accessible to the caller, but are necessary to implement the access operations. The process of access amplification [JONEA73] is used to provide the necessary additional access rights to the called procedure. (Amplification is not, however, universally accepted as necessary [MINSN78].) A good example of this concept is given for the type implementation module of an associative memory object in [JONEA76, JONEA78].

The caller needs some prerequisite rights to invoke a procedure that performs amplification. At levels closest to the object representation, the prerequisite right requirement is fulfilled by having the name of the called procedure as a right stated in the caller's capability to that object. The prerequisite rights needed to invoke a higher level system procedure that performs amplification can be made more general. One way that these prerequisite rights can be checked is by the checkrights mechanism of HYDRA [WULFW74]. The calling procedure must satisfy the requirements specified in the checkrights list of the called procedure. The same concept was called "reqacc" earlier by Jones [JONEA73].

When the amplify binding rule is applicable for a binding operation, the capability variable x of x <-- y must have checkrights declared for it. The rights(y) are compared to checkrights(x) and, if y contains all checkrights(x) (y may also contain other rights), then the binding operation is legal and x is allowed access to the object according to rights(x). The set rights(x) may be a subset, superset, or disjoint set of rights(y). The variable x can be the left side of an explicit binding operation or it can be a formal parameter in a procedure (the distinction is discussed further in section ). For example, a program A having capability variable y referenc-

ing an object (as an actual parameter) calls program B to do operation "O" on the object, where the formal parameter in B for the object is x. Program B has the rights to perform O on the representation of the object and A does not. If the caller's rights are adequate (to invoke the procedure), then the called procedure has the rights listed in the rights list of the same formal parameter. Jones and Liskov [JONEA78] limit application to procedure invocation but amplification does not have to be tied to procedure entry since it also has other uses (see Wulf, et al. [WULFW74]).

A noncomputer example (adapted from [WULFW74]) of amplification occurs when the user of a telephone needs to have it repaired. The user (the calling procedure), who leases the phone (the object) but does not own it, has certain rights to the phone but they do not include the right to repair it (perform the operation directly on the object). They do however, include the right to call the repairman (i.e., they have the prerequisite rights to invoke the amplifying procedure). The repairman, once he has assured himself that it is indeed the proper caller who is requesting the repair (perhaps by a checkright mechanism), has the right to repair the phone. However, the repairman may not use the phone for personal calls unless the user grants him that right.

### 5.1.3. The domtrans binding

When the domtrans binding rule is applicable for a binding operation, x <-- y, the capability variable x takes on the rights of variable y (a variation on this is described in Section where x takes on only a subset of rights(y)). Any rights that x has at the time of binding are irrelevant, as

the dominant rights of y are transferred directly into x; thus, the name of the rule. As noted above, however, variable x in x <-- y cannot have rights set for it at declaration time, as it must have a dynamic rights attribute. Some examples of the use of domtrans are given later in Figure 4.

McGraw and Andrews [MCGRJ79] have taken a general view of binding similar to the domtrans rule. In their access control mechanism, a capability variable can have its access rights changed at runtime by a binding operation and there are no restrictions on whether or not the access rights to an object can be increased.

## 5.2. Specification of binding rules

Before getting to the topic of determining the single applicable binding rule, the application of binding rules is described. A capability variable can have, but is not required to have, a binding rules list. The binding rules list can be initialized at declaration time with one or more binding rules that are applicable for it (see procedures MAIN,A and B in Figure 3). When this occurs, the binding rule(s) declared are considered to be static (per the convention of Section 3) and can never be altered at runtime. On the other hand, capability variables that do not have binding rules specified at declaration time can have them assigned dynmically at runtime by using the

assignbinding <binding rule> to <capability variable list>;

statement and subsequently removed using the

removebinding <binding rule> from <capability variable list>;

statement. The assignbinding and removebinding operations permit a capability variable to be used with different binding rules, thus eliminating the need to declare an excessive number of capability variables for the same object -- one for each binding rule and situation in which it is used.

## 5.3. Explicit and implicit binding

The binding operation x <-- y is an explicit binding operation if it is actually written as "x <-- y" in the executed program code. Some authors call this operation an "assignment" operation. This term is not used here in an effort to avoid confusion with ordinary variable assignment (of value). There are implied (or implicit, usually temporary) bindings which bind each formal parameter (x) to the corresponding actual parameter (y) during procedure invocation. In either case the binding operation changes the reference field of the capability variable on the left side to point to the object referenced by the capability variable on the right side. In some cases, e.g., when the domtrans binding rule is used, the access rights of the left side variable can be changed. The next two sections describe the semantics of explicit and implicit binding and how the applicable binding rule is determined in each case.

## 5.4. Determining the applicable explicit binding rule

It should be understood that only one binding rule can be applicable during a given binding operation whether it be explicit or implicit. In the discussion to follow binding-rules(x) is a set function that yields the list of binding rule(s) in the capability variable x. The function card(A) yields the cardinality of set A. At binding time for the operation x <-- y the following statements must hold* or an applicable binding rule cannot be determined and an error results.

1) card(binding-rules(x)) $\geq$ 1,

2) card(binding-rules(y)) = 1, and

3) binding-rules(y) $\in$ binding-rules(x).

If the binding-rules(x) and/or binding-rules(y) are dynamic, binding rules can be added or removed using assignbinding and removebinding, respectively. For example, if y has two binding rules prior to binding, then one can be removed by the removebinding statement. Or if x or y does not have a binding rule associated with it, then the assignbinding can be used to set the proper one.

When the binding rule that is applicable for x <-- y is either subset or amplify, then both rights(x) and rights(y) must be non-empty; otherwise the binding is automatically illegal. That is, they both must have rights assigned either statically or dynamically (prior to binding), because the legality of binding in both cases is determined by comparing the rights of

---

* Exception: Rules 2 and 3 do not apply in the case 2 type of explicit binding (given below), where the binding rules(y) have nothing to do with determining the rule used for the binding. An example of this exception is shown in Figure 4 for the binding yourfile <-- COPYFILE(myfile) in procedure MAIN.

the two variables, possibly involving checkrights which must always be static. When domtrans is the applicable rule, x must have dynamic rights and y must have a non-empty set of rights, for binding to be legal. In addition, domtrans cannot appear in a binding rules list with either of subset or amplify. In summary, domtrans always appears alone in a binding rules list of a variable while subset and amplify can appear alone or together.

The following three cases exist for determining the applicable explicit binding rule:

1. x <-- y;
   if binding-rules(y) ∈ binding-rules(x) then
   binding(x <-- y) = binding-rules(y)
   else error.
2. x <-- y binding rules {bind-rule};
   if bind-rule ∈ binding-rules(x) then
   binding(x <-- y) = bind-rule
   else error.
3. x <-- y(Q) binding rules {domtrans},
   if domtrans = binding-rules(x) then
   binding(x <-- y) = domtrans
   else error.

In the third case given above, Q is a qualified type specification which contains a subset of the rights of y; i.e., $Q \leq y$. The semantics of the third case are such that, as a result of the binding, x's new rights become those of Q (for the attributes specified in Q); i.e., RIGHTS(x) = RIGHTS(Q).

Some simple examples illustrate the semantics of the explicit binding operation. Suppose

x: list rights {insert} binding rules {subset};

y: list rights {insert, delete} binding rules {subset};

are two capability variable declarations. Then the applicable binding rule for x <-- y clearly is subset. Furthermore, since x $\leq$ y, the explicit binding operation itself is legal. Notice that the explicit binding y <-- x is not valid because y $\not\leq$ x. The applicable binding rule in specific instances is associated with the right side variable while the left side variable gives the possible applicable rules.

For another example, consider the declaration

z: list rights {insert, delete};

Then

x <-- z binding rules {subset};

produces subset as the applicable binding rule and the binding operation itself is legal since rights(x) $\subseteq$ rights(z). On the other hand, the binding rule domtrans is not applicable for

x <-- z binding rules {domtrans};

since domtrans $\notin$ binding-rules(x); furthermore, x has static access rights.

The domtrans binding rule is the only rule applicable for case 3 above. The subset and amplify rules do not change the rights of left side variables and so they are not applicable for explicit binding operations involving explicit rights lists. Case 3 is illustrated using the following declaration for w and the above declaration for z:

w: list binding rules {domtrans};

w <-- z rights {delete} binding rules {domtrans};

The domtrans rule is applicable here. In addition, the binding operation is legal because {delete} c rights(z). As a result of the binding operation, rights(w) = {delete}.

Following are some more examples, using the declarations

a: list [element] binding rules {domtrans};

b: list [element rights {readsalary, writename}]
rights {insert, delete};

For the case 2 binding

a <-- b binding rules {domtrans};

a takes rights of "element" and the rights of "list" from variable b. After this binding operation, "a" is the same as if it had been given the following declaration

a: list [element rights {readsalary, writename}] rights {insert, delete} binding rules {domtrans};

When using structured objects with case 3, above, the rights given by a right side variable must include not only the "structure rights," but must also include "components rights" for each component. For example, this binding

a <-- b (list [element rights {writename}]{insert}
binding rules {domtrans});

produces an "a" which is the same as if it had been given the declaration:

a: list [element rights {writename}]{insert}
binding rules {domtrans};

## 5.5. Determining the Applicable Implicit Binding Rule

The rules for determining the applicable binding rule for implicit binding (during procedure invocation) are essentially the same as for explicit binding, with two exceptions. For an implicit binding x <-- y,

1) variable x must have static binding rules, and
2) when binding-rules (x) $\subseteq$ {subset, amplify}, then rights(x) must be static.

The reason for these two restriction is explained as follows. Variable x must have at least one binding rule associated with it (whereas y must have exactly one) before the applicable binding rule can be determined. Note how this differs from the case 2 type explicit binding, where y does not have to have a binding rule, since the rule is supplied in the binding statement. If binding-rule(x) is not static, then the called procedure would have to execute a statement for x before the applicable binding rule could be determined.

Each of the formal parameters in a given procedure's parameter list may have different binding rules applicable at procedure invocation. This is analogous in Algol to having some parameters passed by name and some passed by value during a single procedure invocation. An actual parameter may have different binding rules associated with it on separate invocations of the same procedure or during invocations of different procedures in which the parameter is used. Thus, for all practical purposes, the actual parameter determines the binding rule for it and its corresponding formal parameter during procedure invocation.

An advantage of the applicable binding rule scheme is that it permits different invocations of a single procedure from the same procedure or from different procedures to have different binding rules applicable. A procedure B may wish the subset rule to apply during invocation of procedure T(x) from B, and A may want the amplify rule to be applicable when T(x) is called from A. This notion alleviates a shortcoming found in other access control mechanisms for high level programming languages. For example, the first invocation of procedure A below (Figure 3) results in subset as the applicable binding rule for parameters x and a, and the second invocation results in amplify for x and a. The binding rule associated with y and b is always subset.

Both of the invocations of procedure A result in applicable binding rules. In addition, the binding rules, when applied, are legal and procedure invocation is permitted. When procedure B calls A, the amplify rule is applicable for x and a and the binding itself is valid since rights(x) in B contains checkrights(a) in procedure A. When the applicable binding rule for x and a is subset, the checkrights are ignored.

The rules for use of domtrans in procedure invocation are the same as for explicit binding, so no examples are given.

The following section, however, provides two examples that illustrate the use of the three binding rules.

```
PROC:  procedure;
       x: list rights {insert, delete, copy};
       y: list rights {insert} binding rules {subset};
                .
                .
                .
       assignbinding subset to x;
       call A(x,y);
                .
                .
                .

       A:  procedure (a,b);
           a: list checkrights {delete} rights {insert, delete} binding
                    rules {subset, amplify};
           b: list rights {insert} binding rules {subset};
                    .
                    .
                    .
           end A;
       B:  procedure;
           x: list rights {delete} binding rules {amplify};
           y: list rights {insert} binding rules {subset};
                    .
                    .
                    .
           call A(x,y);
                    .
                    .
                    .
           end B;
                .
                .
                .
end PROC;
```

Figure 3. Example of Determining Applicable Binding Rule

## 5.6. Examples

Some of the characteristics of the language extensions proposed in this paper can best be illustrated via examples. The first example demonstrates the use of the subset and domtrans rules. The second example involves the amplify rule.

. Subset and Domtrans Example

This first example involves three procedures: MAIN, FILLFILE, and COPY-FILE in Figure 4. In many cases the subset rule can be used to develop a rather fine degree of access control. For example, FILLFILE and COPYFILE can be restricted to filling and copying only files that have certain sets of rights; otherwise, binding will not be valid. Thus, the subset rule is appropriate in cases where binding is required to be limited. In other cases, where it is desired to generalize the routines so that they can apply to various files with a wide variety of access rights, the domtrans rule is appropriate, because access rights can be passed by the caller.

In this example, FILLFILE can fill all files having the same record type as that of xf in FILLFILE, but possibly having different rights. The restriction on record type does not bear on the issue of binding, but does simplify the example with respect to type checking.

In Figure 4 rights(myfile) are passed to xf in FILLFILE by the invocation of FILLFILE(myfile). As a result of the implicit binding using the domtrans rule when FILLFILE is invoked, xf references the same file as referenced by myfile. FILLFILE causes myfile to be filled (possibly from some external source of data). Then COPYFILE is invoked and xfile is bound to myfile. A copy of myfile is made in COPYFILE and returned to MAIN. Then yourfile is bound to this copy, using the subset rule and an application of the case 2 type explicit binding. A check to ensure that "createrec" and "insertrec" in FILLFILE and "copy" in COPYFILE are contained in rights(xf) and rights(xfile), respectively, is made at runtime, since the rights of xf and xfile are dynamic. On the other hand, the check to ensure that "create-

```
MAIN:  procedure;

       myfile: file [record (SSN: char(9); NAME: char(20); SAL: real;) rights
                {createrec, rssn,  wssn,  rname,  rsal}] rights {createfile,
                insertrec, deleterec, copy} binding rules {domtrans}

       yourfile: file [record (SSN:  char(9);  NAME:  char(20);  SAL:  real;)
                rights  {rssn,   rname}] rights  {insertrec}  binding  rules
                {subset};

       myfile.createfile();   /* creates a file referenced by myfile */

       FILLFILE(myfile);   /* fill myfile */

       yourfile <-- COPYFILE(myfile)  binding rules {subset}   /* make a copy
                                 of myfile and bind it to yourfile */
       end MAIN;


FILLFILE: procedure (xf);

       xf: file [r:record (SSN: char(9); NAME: char(20); SAL:  real;) binding
                rules {domtrans};

       while (more_records) do;

           r.createrec();   /* create a record */

           r = ...          /* fill record (with something) */

           xf.insertrec(r);   /*insert r into xf */

          endo;

       end FILLFILE;


COPYFILE: procedure (xfile);
       xfile: file [record (SSN: char(9); NAME: char(20); SAL:  real;)] bind-
                ing rules {domtrans};

       if compare (xfile, xfile.copy)  then return (xfile.copy);  else ERROR;
                /* copy  creates a  copy of  xfile with  xfile's attributes;
                "compare" compares the data content of the two files */
       return;

       end COPYFILE;
```

Figure 4. Example using the subset and domtrans binding rules

file" is in rights(myfile) is made at compile time, since the rights of myfile are static.

The declaration for xf in FILLFILE points out that component objects can have capability variables declared for them when it is necessary for them to be manipulated explicitly. Record components of file xf are declared with a capability variable "r"--for the sake of simplicity, a general notation for doing this is not developed here.

There is a, perhaps, subtle concept related to file xf (in FILLFILE). The rights one expects an object to have are the rights of the qualified type specification that defines the object's type (i.e., the rights of the capability variable that references the object). Those rights are either statically assigned (e.g., in the case of the record component objects in myfile in MAIN), or they are those rights received during a binding operation involving the domtrans rule. For example, the expected rights of the component records in file xf are {createrec, rssn, wssn, rname, rsal}. Since "file" is a programmer defined type, implemented by a programmer, one can describe an implementation such that the rights associated with each record occurrence possibly differ from the expected rights. Thus, there can be an inconsistency between expected rights (the rights of the defining qualified type) and the rights determined by an implementation strategy. For example, one way of implementing a file is as a list of capability variables, each referencing a record occurrence. This representation is general enough to allow each of the various record instances to be associated with some subset of the rights defined for the record component type in the main qualified type definition for the file. Such an implementation could allow the rights of record occurrences to be determined by the record capability variables, as opposed to the "expected" rights.

The above implementation of files implies that the condition "RIGHTS(x) $\leq$ RIGHTS(y)" must be checked at runtime. The implication is that RIGHTS is implementation dependent; but it is not. The RIGHTS function, as defined in Definition 2, was intended to use the full set of expected rights.

To resolve this inconsistency, the following interpretation is included as part of the language extensions: Component object occurrences (e.g., record occurrences in xf) are assumed to have the expected rights (i.e., the rights of the main defining qualified type specification), and, when these component objects are used in binding operations, these implementation independent rights should be taken as the rights of the occurrences. This interpretation is implemented by having a single capability variable associated with each component object type in a structure. All occurrences of each type are then related to one implicit capability variable and this capability variable does not contain an actual reference to any component. This means that there is one capability variable per component object type, holding the rights of all occurrences of that type. This concept can be generalized to multiple types of components in a straightforward manner.

### 5.6.2. Amplify Example

This example involves a file of records and a situation in which it is desired to delete a record. In order to delete a record, the delete operation procedure needs access to the representation of the file (something which the calling program does not have). The deletion is accomplished by repeated application of the "compactspace" operation, which moves the rest of the records up by one place, overwriting the record to be deleted.

Therefore, the calling program's rights to the file must be amplified to permit deletion at the representational level. Amplification typically occurs when a type module is entered to perform an operation on an instance of the type. The procedures in Figure 5 illustrate the part that checkrights plays in the amplify binding rule. The amplify binding is not valid unless rights(f) contains checkrights(xf). Despite the simplicity of this example, neither the subset nor the domtrans rule could have been used to specify access control policy here, because procedure DELETEREC requires amplification of rights to perform the delete operation. In particular, it needs to perform the "compactspace" operation, the use of which is not permitted outside of DELETEREC (or outside of the module in which DELETEREC is implemented).

## 6. CONDITIONAL CAPABILITIES

The contents of this section represent a further extension of the use of capabilities. A conditional capability (introduced by Ekanadham and Bernstein [EKANK79]), is a capability that can be exercised only when a particular condition or conditions are true. Conditions placed in a capability can control the use of capabilities. For example, they can be used to prevent the propagation of an argument capability after it has been passed as a parameter or to prevent a procedure from accessing the object the capability references except during specific time periods, etc.

```
MAIN: procedure;

      f: file [record (key: char(6);  info:  char(20)) rights {rkey, rinfo}]
               rights  {insertrec,   deleterec,    nextrec}  binding  rules
               {amplify};
                    .
                    .
                    .
      DELETEREC(f, keyval)    /* keyval contains the key of  the record to be
                                 deleted */
                    .
                    .
                    .
      end MAIN;



DELETEREC: procedure (xf, kval);

      kval: char(6);

      xf: file [r:  record  (key:  char(6);  info char(20))  rights {rkey}]
               checkrights {deleterec} rights {nextrec, compactspace} bind-
               ing rules {amplify};
                    .
                    .
                    .
      repeat r <-- xf.nextrec until r.rkey = kval;

      xf.compactspace(r);     /* compactspace  is a file operation  that col-
               lects garbage;  it picks up space left by deletion of record
               r */
                    .
                    .
                    .
      return;

      end DELETEREC;
```

Figure 5. Example using amplify binding rule

Conventional capabilities differ from conditional capabilities in the sense that, if a procedure possesses a conventional capability, it can be exercised to perform any of the operations permitted by the capability variable. No control can be exerted over a conventional capability as to when or where the capability can be used.

Conditional capabilities are utilized in the following manner. For each condition x in the capability there is a lock, L(x). To exercise a capability with a lock L(x), a matching key, K(x), has to be presented. As we stated earlier, an instruction attempting an operation on an object referenced by capability variable y is of the form

y.operation ( )

Such an operation is valid if:

1) the conditions associated with capability variable y are satisfied, and

2) the "operation" is in the rights list of y.

If variable y contains locks L(a) and L(b), then keys K(a) and K(b) must be presented along with the capability y if the capability is to be exercised. The keys are supplied as arguments of the operation in the following manner

y.operation ( ), K(a), K(b);

The above operation can be executed whereas

y.operation ( ), K(b);

will not be executed, because lock L(a) is not "unlocked."

A capability variable can be declared with locks as in the following declaration for capability variable y

y: list rights {insert, delete} locks {LOCK1, LOCK2};

The value (the key), set at declaration time, which opens a lock can never be altered. A capability variable can also be declared without locks as in the declaration

z: list capability;

Capability variable z can have locks assigned to it by using the

assignlock <lock> to <capability variable>;

statement. A lock can be removed from a capability variable by using the

removelock <lock> from <capability variable>;

statement. The assignlock and removelock operations are analogous to the assignbinding and removebinding operations.

The locks on capabilities are similar to the locks placed on certain operations in database management system schema definitions. For instance, a delete operation might not be permitted unless the user of the database supplies the correct lock key for the operation.

Ekanadham and Bernstein generate locks (and hence keys) at runtime by using unique random numbers [EKANK79]. They have a relatively elaborate scheme and it is particularly suitable for use in operating systems. The approach presented here differs in that it permit locks to be specified at compile time.

## 7. ACCESS CORRECTNESS

An access control facility should provide these two abilities, which are important to access correctness:

1) to limit each program unit to the data objects and operations that it needs, and
2) to insure that only authorized operations are applied to data objects.

Jones and Liskov's system requires each variable to have its access rights set at declaration time. Hence all use and movement of rights (via explicit binding and procedure invocation) can be checked at compile time to determine access correctness. One major drawback of their system is that it does not provide the flexibility required to build dynamic systems. In dynamic systems, such as file systems, rights must often be allocated dynamically for objects such as buffers, files, etc.

The language extensions described in this paper permit the user to develop access control policy such that access correctness can be determined at compile time. In this case each capability must have its access rights, binding rules, and locks, if any, set at declaration time. Access policies can also be developed for dynamic systems such that some accesses must be validated at runtime.

## 8. FUTURE WORK AND CONCLUSION

The work reported in this paper has also raised some additional questions which must be reserved for future work. Some of these questions relate to the adaptation of the concepts of this paper to the area of database systems. For example, it is interesting to consider the introduction of predicates into the object definitions which would make the acess rights dependent upon system state and data values. Some binding rules would then have to be evaluated at access time, since the contents of object instances would be required to make the corresponding access decisions.

Other extensions, too, might apply to languages designed for database applicationss. For example, consider the possibility of obtaining the data type definitions, especially the rights lists, at binding time from somewhere external to the program and its execution environment (for example, from a file in secondary storage). This would allow the access control policies to be determined dynamically within a database authorization system.

Another question for future work involves dynamic implicit binding of formal parameters during procedure invocation. As discussed in this paper, the binding rules of formal parameters are static. The reason is straightforward: dynamic binding would require the entry of the procedure to determine the applicable binding rule, prior to invocation. To allow this, the procedure would need some sort of "prologue" in addition to the "body" of the program. The prologue would serve to interact with the execution environment to make the binding decision, before proceeding (or not proceeding) with the execution of the body of the program.

Finally, a small modification of the syntax would generalize the attribute definitions within object type definitions by allowing attributes that are: static, dynamic but initialized, or dynamic and uninitialized. Static attributes are then indicated by the '=' operator, as in this example:

x:record (...) rights = {rssn,wname};

dynamic initialized attributes are indicated, as follows, by the ':=' operator:

x:file [record (...) rights {...}] rights := {insertrec,deleterec};

Dynamic, uninitialized attributes are still indicated by their absence. Dynamic initialized rights can be altered at runtime, and perhaps they could be reset to their declared values during program execution with a reset operator.

In this paper a number of language extensions have been described which permit a wide range of access control policies to be specified. By choosing a subset of the extensions to develop policy, the user can cause access correctness to be determined at compile time. On the other hand, the extensions permit policies to be specified that require access correctness to be determined at runtime. These extensions permit more flexibility in specifying language based access control policy than other access control facilities currently available.

REFERENCES


AMBLA77 Ambler, Allen L., and Charles G. Hoch, "A Study of Protection in Programming Languages," ACM SIGPLAN Notices 12, 3 (March 1977), 25-40.

CLAYB79 Claybrook, Billy G., et al., "Logical Structure Specification and Data Type Definition," Proc. of ACM Annual Conf. (October 1979), 203-211.

CLAYB80 Claybrook, Billy G., "module: An Encapsulation Mechanism for Specifying and Implementing Abstract Data Types," Proc. of the ACM Annual Conf. (October 1980), 225-235.

DALER65 Daley, R. C., and P. G. Neumann, "A General Purpose File System for Secondary Storage," AFIPS Proc. of the FJCC (1965), 213-229.

DENND79 Denning, Dorothy E., and Peter J. Denning, "Data Security," ACM Computing Surveys 11, 3 (September 1979), 227-249.

EKANK79 Ekanadham, Kattamuri, and Arthur J. Bernstein, "Conditional Capabilities," IEEE Trans. on Software Engineering, Vol. SE-5 (September 1979), 458-464.

JONEA73 Jones, Anita K., "Protection in Programmed Systems," Ph. D. dissertation, Department of Computer Science, Carnegie-Mellon University (June 1973).

JONEA78 Jones, Anita K., and Barbara H. Liskov, "A Language Extension for Expressing Constraints on Data Access," Comm. of the ACM 21, 5 (May 1978), 358-367.

LAMPB71 Lampson, Butler W., "Protection," Proc. Fifth Princeton Symp. on Information Sciences and Systems, Princeton University (March 1971), 437-443; reprinted in ACM SIGOPS Operating Systems Review 8, 1 (January 1974), 18-24.

MCGRJ79 McGraw, James R., and Gregory R. Andrews, "Access Control in Parallel Programs," IEEE Trans. on Software Engineering, Vol. SE-5 (January 1979), 1-9.

MINSN78 Minsky, Naftaly, "The Principle of Attenuation of Privileges and Its Ramifications," in Foundations of Secure Computation, ed. by R. A. DeMillo, et al., Academic Press, New York (1978), 255-277.

MORRJ73 Morris, James H., Jr., "Protection in Programming Languages," Comm. of the ACM 16, 1 (January 1973), 15-21.

WULFW74 Wulf, William, et al., "HYDRA: The Kernel of a Multiprocessing Operating System," Comm. of the ACM, 17, 6 (June 1974), 337-345.

WULFW76 Wulf, William A., Ralph L. London, and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Engineering, SE-2 (December 1976), 253-265.