LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

An Access Control Facility for Programming Languages

Computation Structures Group Memo 137
April 1976

Anita K. Jones
(Computer Science Department, Carnegie-Mellon University)

Barbara H. Liskov

This research was supported by the National Science Foundation under grants DCR74-21892 and DCR74-04187.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

AN ACCESS CONTROL FACILITY FOR PROGRAMMING LANGUAGES

Anita K. Jones
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Barbara H. Liskov
Department of Electrical Engineering
and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

Controlled sharing of information is needed and desirable for many applications. Access control mechanisms exist in operating systems to provide such controlled sharing. However, programming languages currently do not support such a facility. This paper argues that to enhance software reliability programming languages should support controlled sharing of information; the paper illustrates how such an access control facility could be incorporated in a programming language. The mechanism described is suitable for incorporation in object-oriented languages which permit the definition of abstract data types; it is defined in such a way as to enable compile time checking of access control.

Keywords and Phrases: access control, data types, type checking, capabilities.

CR Categories: 4.20, 4.35

This research was supported by the National Science Foundation under grants DCR74-21892 and DCR74-04187.

1. INTRODUCTION

One of the most important attributes of a programming language is the way the scope rules of the language define how data is to be shared among the individual program units (procedures, blocks, modules) out of which a program is constructed. Ordinarily, access to data is provided on an all-or-nothing basis: if a module can access some data base at all, then every component of the data base may be accessed, and every possible type of access (usually just reading and writing) may be performed . Experience in building large applications, or applications involving sensitive data, has indicated that sharing of data is greatly enhanced if finer control than all-or-nothing access is provided. For example, manipulation of the information in a data base is much more controlled if not every program which reads the data base is also permitted to write it. In addition, if some of the information in a data base is sensitive, then control over which programs can read which information is also desired.

Current programming languages are deficient in providing mechanisms for controlling the sharing of information among program units. For example, passing a data base "by value" ensures that the called procedure may not modify the data base. However, this mechanism does not provide control over what parts of a data base may be read; in addition, it is so expensive for large data bases that other parameter passing mechanisms are used instead. Proposals for avoiding the overhead of call by value while retaining the benefit that the data base cannot be modified (for example, call by reference, but permitting only read access to the formal parameter) solve the efficiency problem, but still do not provide for selective reading of the data base. In addition, such proposals do not provide for the control of selective alteration of the data base.

The thesis of this paper is that programming languages should provide mechanisms for controlled sharing of data. We define a syntax and semantics for such a mechanism. The mechanism we will show borrows heavily from work in operating systems, where access control mechanisms have long been one of the tools useful for realizing controlled sharing of data. In particular, our mechanism is modeled after the capability protection mechanisms provided by some operating systems [Sturgis74, Wulf74].

To incorporate an access control mechanism in a programming language, we will choose an approach that permits programmers to express access control restrictions in terms that are meaningful to their application domains. We assume that all data are contained in objects for which there exists a set of accesses. Objects are those entities, such as data bases, libraries, stacks or files, which are of interest to programmers. Accesses are limited to those that are meaningful manipulations of the objects; accesses are the only means for altering an object or extracting information from it. In some cases, meaningful accesses are the familiar read, write, and, possibly, execute access. In other cases, the accesses themselves are user-defined, tailored to the abstract notion the user intends to capture. For example, a file system may distinguish between write access and append access. In contrast to a write access, an append access is assumed to modify the file, but not to alter existing content. This permits a user to share a file with others, allowing them to augment the file by appending to it, but not allowing them the ability to rewrite any portion of it.

What is to be gained by incorporating access control into a programming language? We believe several benefits will accrue. The crucial benefit is enhanced software reliability in that programs can be written to be well-behaved with respect to the constraints governing sharing of data. We will call such programs access-correct. An access-correct program obeys the following constraints:

1. It may access only those objects which it has a legitimate right to access.

2. It may perform only meaningful accesses to these objects.

3. If it is restricted to performing some proper subset of the meaningful accesses to an object, this restriction cannot be circumvented.

Access control restrictions are stated in a declarative fashion analogous to type declarations for variables. They may be viewed similarly as a statement of user intent, which may be relied on by someone reading the program to obtain a better understanding of the purpose of the program.

In addition access control can be introduced into languages in such a way that the access-correctness of a program can be checked at compile-time. This will lead to benefits similar to those derived from compile-time type checking (indeed, the mechanism we provide is a logical

extension of type checking): the assurance that a compiled program is access-correct, and (possibly) enhanced efficiency over the dynamic mechanisms currently provided by operating systems. Note that compile time checking of access control is beneficial even if dynamic mechanisms are retained: access control errors are caught early, and a programmer may be confident that his program will not fail due to an access-control violation.

The last benefit we wish to point out is that a programmer will be able to express fully in the language how he intends to make use of the protection facilities of an operating system. At present, the access control information is expressed separately from the program in some sort of job-control language; such a separation increases the difficulty of writing programs for such systems. In addition, the language permits more precise specification of access requirements on a program by program basis, not on a user job or job step basis.

In the next section, we describe the kind of programming language we have chosen as a basis for incorporation of an access control mechanism, and define how access control is achieved for simple, unstructured data objects. Section 3 extends the access control mechanism to data structures. In Section 4 we compare our mechanism with the dynamic mechanism in the Hydra operating system [Wulf74, Jones75]; this is especially interesting since our mechanism is modeled after capability protection mechanisms like that provided in Hydra. We conclude in Section 5 with a discussion of what we have accomplished.

2. THE BASIC MODEL

In this section, we describe the kind of programming language that we have selected as a basis for incorporation of access control, and then define a notation and set of rules sufficient for controlling access to simple, unstructured objects. Since our purpose is to illustrate how access control might be incorporated into a programming language, rather than to define a complete programming language, we introduce only a minimum of syntax and semantics to express the access control rules. Our semantic model is chosen to have the following characteristics:

1. It is consistent with defining access control in terms meaningful to user applications.

2. Sharing of data objects is natural and straightforward.

3. It is possible to determine at compile time whether a program obeys the access control rules, and is thus access-correct. In addition, the decision about whether a program is access-correct can be made based on only local information, similar to the way that type-checking in a strongly typed language is performed.[1]

We can develop some intuition about the character of a language into which access control can be integrated by considering how operating systems provide access control: The data-containing objects to be controlled are uniquely distinguishable (each object has a unique identity). All direct manipulation of an object is via accesses to it. The accesses to an object must be distinguishable so that unauthorized accesses can be prevented. For each object there is a set of potentially allowed accesses; no other accesses can be performed on that object. The potentially allowed accesses depend on what kind of object is being accessed, and users require the ability to define new kinds of objects suitable to their particular application domains.

Thus, to discuss access control we require a language that permits the writing of programs in terms of data objects and the accesses that are meaningful for them. In particular, languages in which a datum is viewed as an aggregate of memory cells are not suitable, because of the difficulty of expressing access control on anything but a cell basis. One class of languages, including the

---

1. The question of how static checking interacts with programming power is addressed in Section 4.

languages Simula67 [Dahl72], CLU [Liskov76] and Alphard [Wulf76], provides a natural environment in which to embed an access control facility. We will call such languages object-oriented languages.

The suitability of object-oriented languages for embedding access control arises primarily from the view of data types taken in these languages. A data type is considered to be more than simply a set of objects or values. A type also specifies a set of operations which provides the means for manipulating the objects: The operations provide for creating objects of the type, for obtaining information about objects of the type, and for altering objects of the type. The operations of a type correspond very closely (though not identically, as we shall show) to our notion of access, and access control corresponds to the ability to control the use of the operations.

The user of objects of some type is constrained to view those objects abstractly in terms of the type's operations rather than in terms of the objects' representation. In order to define a new type, a storage representation must be specified for the type's objects; however, this representation can be manipulated only by the type's operations. Limiting knowledge of the storage representation to just the operations ensures that those operations completely determine the behavior of the type's objects [Liskov76].

In order to accommodate access control, we will add one more component to a type: In addition to objects and operations, a type also specifies a set of rights. A right is a name that represents a meaningful manipulation of objects of the type; often a right corresponds to the use of one of the type's operations. The basic idea behind rights is: to legally apply one of the type's operations, a user must hold appropriate rights to the objects passed to that operation as parameters.

An example is given in Figure 1 for the type, associative-memory. Operations for this type include an operation to create an empty associative-memory of a particular size (makemem), an operation to add a name-value pair to an associative-memory (insert), an operation to change the value associated with a given name (change), an operation to fetch the value associated with a

given name (getval), and an operation to remove a name-value pair (delete) In order for insert, change, getval, or delete to be invoked, the invoker must present a right to apply the operation to the associative-memory parameter; in this particular example, the name of the required right is the same as the name of the operation. The makemem operation returns all these rights for the associative-memory object it creates. The associative-memory operations also use objects of type integer. For simplicity we have chosen to let a single right ("use") control the use of all integer operations. In general, we can expect some rights to correspond to the use of a single operation, some to a group of operations (type integer provides a degenerate example of this case), and some to a single parameter of an operation taking more than one object of the type.

```
type:     associative-memory
rights:   "insert", "change", "getval", "delete"
operations:
  makemem
    input:      integer; "use" right     comment desired associative-memory size
    returns:    associative-memory; "insert","change","getval", "delete" rights are given
  insert
    input:      associative-memory; "insert" right
                integer;   "use" right     comment the name
                integer;   "use" right     comment the value
    effect:     (insert modifies its associative-memory parameter)
  change
    input:      associative-memory; "change" right
                integer;   "use" right     comment the name
                integer;   "use" right     comment the new value
    effect:     (change modifies its associative memory parameter)
  getval
    input:      associative-memory; "getval" right
                integer; "use" right     comment the name
    returns:    integer; "use" right     comment the value
  delete
    input:      associative-memory; "delete" right
                integer;   "use" right     comment the name
    effect:     (delete modifies its associative-memory parameter)
```

Figure 1. The Associative-memory Type.

Types such as associative-memory can be implemented by means of a special kind of program that defines what the rights and operations are and provides implementations for all of the type's

operations. We will refer to such a program as a type-module.[2]

Notation and Rules for Access Control

Our notation for access control involves a declaration for each variable of the type of object that variable may reference, and the rights that are available for that object when it is accessed via the variable. These two pieces of information are captured in the notion of a qualified type. A qualified type is written

T{r1,...,rn}

where T is the name of some type, and {r1,...,rn} is a non-empty subset of the rights of T. We refer to the two parts of a qualified type as the base type and the rights; if Q is a qualified type, then base(Q) is the base type and rights(Q) is the rights. For example, the following are some of the qualified types derived from associative-memory

associative-memory {getval}
associative-memory {insert, change}
associative-memory {insert, change, getval, delete}

The final example specifies all the associative-memory rights; a special notation

T{all}

may be used instead of listing all the rights.

Qualified types are used in variable declarations and in formal parameter specifications in procedure headings. An example of a variable declaration is:

v: associative-memory {insert, change}

The meaning of this declaration is: v is a variable which can be used to reference associative-memory objects, but only the "insert" and "change" rights may be exercised in conjunction with v.

We view a variable as a pair

---

2. Type-modules will be discussed later in this section. Type-modules are similar to classes in Simula, clusters in CLU and forms in Alphard.

{object id, qualified type}

The object id is a unique name which is interpreted by the underlying addressing mechanism to select an object. The type of this object is guaranteed (by the access control rules) to be the base type of the qualified type of the variable. When a variable is created, its qualified type is defined once and for all and can never be altered. However, the object named by a variable (via the object id) can change by application of the binding operation discussed below. Note that it is possible for sharing of objects to take place, because two variables may contain the same object id. In this case, the qualified type in the two variables may differ, but the base type is necessarily the same.

A variable is a capability in the operating system sense [Dennis66, Lampson71, Jones73]. The capability provides the basis for restricting the kinds of manipulation that can be performed on the object specified by the object id. Intuitively, the restrictions on how an object can be used are expressed along the path to the object (the path through the object id in the variable). Thus, using one path rather than another to name an object changes the way the object can be manipulated. For example, suppose

a: associative-memory{getval, insert}
b: associative-memory{getval}

both name the same object. Using b it is impossible to modify this object, since only the getval operation can be used; using a, the object may be modified by application of the insert operation.

The notions of variables, objects and binding are different from the related notions of value and assignment which underlie block-structured languages. This difference is illustrated in Figure 2. Figure 2a shows the traditional view of variables and values, in which the value resides in the variable and a new value can be copied into a variable by means of assignment. Figure 2b illustrates our semantics: a variable is bound to an object, and a value is contained in an object. This value may only be accessed or modified by means of one of the operations of the object's type. Our rule of binding differs from assignment in that it causes sharing of the object involved, rather than the copying of the value in the object. Furthermore, this sharing is significant since for some types of objects, operations exist to change the value inside of the object. For example,

the associative-memory operations insert, change and delete modify the value inside of an associative-memory object.

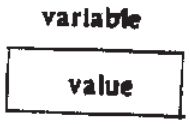

Figure 2a. Traditional view of variables and values.

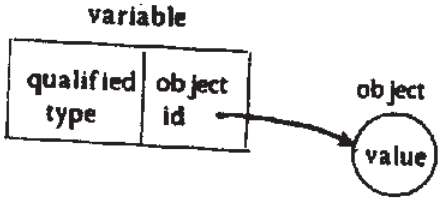

Figure 2b. Model used in this paper.

Figure 2. Comparison of Semantic Models.

Our notion of binding corresponds to assignment involving variables holding (typed) references to objects. Some programming languages are based on a semantic model like ours. The most widely known of these languages is LISP [McCarthy62]; LISP lists are objects (with operations car, cdr and cons) and LISP setq is the same as our binding. Our model is also used in SIMULA 67, CLU and Alphard.

We claim that our semantics models very well what is going on in systems where controlled sharing is of interest. Note that sharing of objects is a fundamental fact in these systems; the sharing of actual objects (rather than just copies of the values of objects) leads both to interesting behavior (e.g., many programs working with the same data base), and the need to exercise some control over exactly how the object should be shared. Protection schemes exist to provide this control.

Binding Rule

A single rule, governing the legality of binding of objects to variables, is sufficient to provide the required access control and is the basis for determining whether a program is access-correct (satisfies the sharing constraints discussed in Section 1). Binding is the operation that causes a variable to reference an object (by changing the object id). The effect of binding is creation of a new access path for the object. Therefore, in order to ensure that a program is access-correct, we must guarantee that no new rights to access the object are obtained from this new access path. For example, suppose that x and y are variables, and that x is to be bound to the object currently bound to y. This new binding should be allowed only if the qualified types of x and y both arise from the same base type, and if the rights obtainable by accessing the object via variable x do not exceed the rights obtainable by accessing the object via y.

We can formalize this rule as follows. First, we define what it means for one qualified type to be greater than or equal to another. If Q1 and Q2 are qualified types, then Q1 is greater than or equal to Q2, written

$$Q1 \geq Q2$$

if base(Q1) = base(Q2) and rights(Q1) $\geq$ rights(Q2). Now the rule of binding can be defined:

$$v \leftarrow e$$

where v is a variable and e is an expression and

Tv = qualified type of variable v
Te = qualified type of expression e

is legal provided that

$$Te \geq Tv$$

Thus a binding is legal only if the new access path provides at most a subset of the rights obtainable via the original access path. Note that this rule ensures that a variable will always reference an object whose type is the base type of the qualified type of the variable.

An expression is either a variable, in which case its qualified type is the same as the qualified type of the variable, or it is a procedure invocation. In the former case, we have now defined the

rule of binding (since Te is the qualified type of this variable). For example, suppose

a: associative-memory[getval, insert]
b: associative-memory[getval]

Then b ← a is legal, but a ← b is not. This is illustrated in Figure 3. In Figure 3a, an initial configuration is shown in which a references an associative-memory object α, and b references an associative-memory object β. Figure 3b shows the result of b ← a. Both b and a now reference α. A new access path (from b to α) has been created as a result of this binding, but no new rights to α are obtained by it; in fact, the new access path via b has fewer rights to α than the old access path. Figure 3b illustrates what would be the result of a ← b. If this binding were allowed, the new access path from a to β would allow more rights than the old one, and therefore the binding must not be permitted.

In order to understand binding when the righthand side is a procedure invocation, we must examine the semantics of parameter passing. Our notion of parameter passing is defined in terms of binding. A procedure definition has the form

```
procedure <procname> (<formals specification>)
        returns <result specification> =
        <body>
        end <procname>
```

where <formals specification> specifies the name and qualified type for each formal parameter, and <result specification> specifies the qualified type returned by the procedure. Each formal parameter is considered to be a local variable of the procedure; this variable is created at invocation, and the actual parameter is bound to it. The <body> is then executed, and finally an object, whose type is the base type of the qualified type in the <result specification>, is returned.

For example, suppose a procedure P has type requirements

```
procedure P (x: T1[f1,f2]) returns T2[g1]
```

and declarations

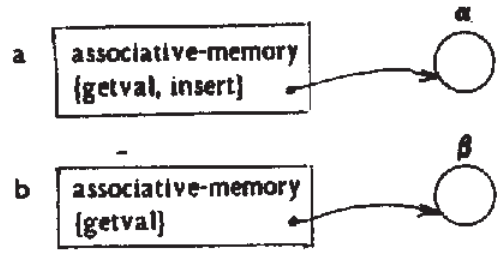

Figure 3a. The initial state.

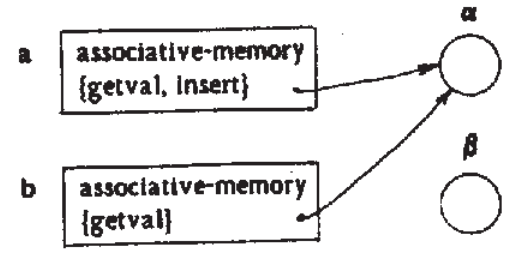

Figure 3b. Result of b ← a.

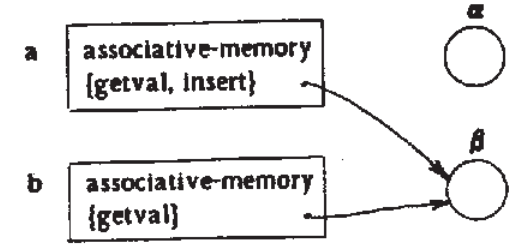

Figure 3c. Result of a ← b (disallowed).

Figure 3. Binding.

a: T1{f1,f2,f4}
b: T2{g1}

occur in the invoker of P. Then the statement b ← P(a) is legal. The passing of parameters and the return value is effectively simulated as follows: As part of the procedure invocation and before execution of the procedure body, two locals, x and retval, are declared

x: T1{f1,f2}
retval: T2{g1}

and the object referenced by the actual parameter is bound to x (x ← a). Execution of the body terminates with execution of a return statement of the form

return e

This can be simulated with bindings

retval ← e
b ← retval

The procedure P is access-correct only if all of its bindings are legal; this includes the binding retval ← e but not x ← a nor b ← retval. For retval ← e to be legal, the qualified type of expression e must be ≥ that of retval, i.e., that defined in the return specification. Thus if (in the body of P)

y: T2{g1,g3}
z: T2{g2,g3}

then return y is legal but return z is not. Note that the access-correctness of P can be determined by local examination of its definition.

The invocation of P

b ← P(a)

is legal because the bindings of x ← a and b ← retval are legal. However, c ← P(a), where c: T2{g1,g3}, is not legal.

Procedure invocation is the mechanism whereby objects are created in the first place. There exist a number of primitive data types (for example integer, boolean, array). The create operations of these types provide objects of the type whenever they are invoked, and these objects are returned with full rights. For the non-primitive, user-defined types the situation is analogous. This has already been illustrated in the associative-memory example shown in Figure I; whenever the makemem operation for associative-memory is invoked, it returns a new associative-memory object with full rights. Thus the creator of an object obtains all rights to it. As the object is passed from one access-correct procedure to another, certain rights may be removed, but rights are

never gained. This is true because binding is the only method provided for transmitting access paths to objects (references to objects) between procedures.

Amplification and Type-Modules

We have presented a rule for binding that regulates how users of an object can create new access paths to objects in order to judiciously share them between procedures. Now we focus on the type-module, the mechanism that is used to implement objects and accesses to objects.

Sometimes, in order for a useful function to be accomplished, it is necessary for the called procedure to obtain more rights to the object than the caller had. When this occurs it is called amplification [Jones73]. In our model, we permit amplification to occur at only one point: at entry to a procedure implementing an operation defined in the type-module. There are always two types associated with a type-module: the type being defined, which we will call the abstract type, and the representation type, which is used to represent objects of the abstract type. Amplification is the mechanism that controls conversion between these types, thus permitting the procedures implementing the operations of the type to obtain access to the objects' representations.

A type-module defines the following information:

1. A list of the rights defined for objects of the type.

2. A list of the operations defined in the module that may be invoked by programs external to the type-module. Note that the operations (defined by procedures) in such a module may require as parameters objects of types defined elsewhere.

3. A description of the storage representation for objects of the type.

4. Procedures, some of which define the type's operations.

In Figure 4, a portion of the type-module for the associative-memory type is shown. The storage representation for associative-memories is a record (similar to a PASCAL record [Wirth71]) containing an integer to tell how large the associative-memory is, an integer to tell how full the

associative memory is, and two arrays of integers to hold the names and the values.[3] This is declared by

```
rep= record [ size: integer{all}, full: integer{all},
              name: array[integer{all}]{all},
              value: array[integer{all}]{all} ] {all}
```

which defines the representation type for this type-module; the abstract type is associative-memory.[4] Note that all rights for every component of the representation are available for use within procedures defined in the type-module (when an object of type associative-memory is passed as an actual parameter).

It is the procedures in a type-module that determine the behavior of objects of that abstract type. To do so these procedures need to manipulate the representation of abstract type objects received as parameters. Thus procedures in the type-module require the right to convert an object between its abstract and its representation type. For example, the associative-memory insert procedure takes an associative-memory object as its first parameter s. Inside the body of insert, s is treated as type record (the associative-memory representation type) and the record components are accessed.

The makemem procedure creates a variable r of the representation type, and constructs a new record object which is bound to r through the statement

```
r: rep ← recordcreate(size: n, full: 0, name: arraycreate(l,n), value: arraycreate(l,n))
```

The component labelled full of this record object is initialized to 0, the component labelled size is initialized to the size desired by the caller, and the components labelled name and value are initialized to new array objects having l and n as lower and upper bounds.[5] Makemem is defined

---

3. Data structures are discussed in Section 3.

4. If this mechanism were embedded in an actual programming language, specification of full rights would probably be elided.

5. We have chosen to make array bounds information part of array object creation to simplify the semantics of data structures (see Section 3).

```
type-module associative-memory =

    rights insert, change, getval, delete;

    operations makemem, insert, change, getval, delete;

    rep = record [ size: integer{all}, full: integer{all},
                   name: array[integer{all}]{all},
                   value: array[integer{all}]{all} ] {all};

    comment we have full rights to integer, array and record objects;

procedure makemem (n: integer{all}) returns associative-memory{all} =

    comment parameter n determines size of associative memory;
      r: rep ← recordcreate(size: n, full: 0, name: arraycreate(1,n), value: arraycreate(1,n));
      return r;

    end makemem;

procedure insert (s: associative-memory{insert}, n: integer{all}, v: integer{all}) =

        if s.full = s.size or in(s, n) ≤ s.full then signal inserterror;
        s.full ← s.full + 1;
        s.name[s.full] ← n;
        s.value[s.full] ← v;

    end insert;

procedure in (s: rep{all}, n: integer{all}) returns integer{all} =

    comment in is an internal procedure;
    comment returns index of s.name entry containing n, else returns s.full + 1;
        for i: integer{all} ← 1 step 1 to s.full do
            if s.name[i] = n then return i;
        return (s.full + 1);

    end in;
    .
    .
    .
end associative-memory
```

Figure 4. Part of the associative-memory type-module.

to return, not a record, but an associative-memory object; the record object is automatically converted to its abstract type associative-memory as part of the return from the makemem procedure.

The insert procedure can add a name-value pair to the associative memory only if there is room, and if there is no previous entry for this name. If these conditions are not satisfied, it reports an error, using whatever error reporting mechanism exists in the language. To determine whether an entry already exists for this name it calls procedure in; this is an internal procedure of the type-module, which is not accessible outside because it is not listed among the operations.

We have chosen to make conversion between abstract type and representation type be automatic within a type-module;[6] type module procedures can reference abstract objects as if they were of representation type and vice versa. In either case, "full" rights are available. Full rights to the abstract type are those defined by the type-module; full rights to the representation type are those specified in the rep type definition. Note that these conversions are purely changes in the point of view of the compiler; no code need be executed to accomplish them. Note also that the conversions apply only to the abstract type being defined by the type-module.

Type conversion is limited to the type-module. If an abstract type object could be converted to its representation outside the type module, any operation of the type could be performed (via manipulation of the object representation), even if the right to perform that operation were not present. By limiting this conversion to just the type-module, we guarantee that the access control restrictions cannot be violated. Conversion from representation to abstract type is also limited to the type-module, so that counterfeit objects, whose representation might not even agree with the representation type of the type-module, cannot be formed.

It is worth noting the difference between the rights qualifications appearing in the heading of

---

6. A discussion of the semantics of these conversions may be found in [Liskov76].

a procedure defining a type's operations and an ordinary procedure. In the case of the ordinary procedure, the rights qualification describes constraints on the procedure itself and also on its caller. For example, in

procedure P (s: associative-memory{insert})

P can use only the insert operation on the associative-memory object named by s (either directly or through some procedure P calls), and the caller of P must have been able to insert values in this object. In the case of an operation-defining procedure, the caller is constrained to provide appropriate rights for an object of the type, but because of amplification the operation itself has no rights constraints.

Remarks

We have now described an access mechanism sufficient to control the sharing of many of the kinds of objects of interest in programming. For example, suppose we define a type employee-record, with operations (and rights) to read-job-category, write-job-category, read-salary, and write-salary, among others. Using the rules defined so far, we can define a procedure

procedure P (x: employee-record{read-job-category, write-salary})

which computes a new salary based on the employee's job-category, but is unable to change the job-category, or to read the old salary.

We claimed earlier in this section that if all the bindings in a procedure were legal, then the procedure was access-correct. We offer the following informal justification for this claim. The binding rule is defined so that no new access rights can be obtained through a legal binding. Therefore, the only way to obtain extra rights is by passing an object to some other procedure, which somehow gives out the extra rights. There are two cases to consider here: a call on an ordinary procedure, and a call on a procedure implementing an operation on the object in question. No extra rights can be obtained in the former case if the called procedure is access-correct. In the latter case, extra rights can be obtained if the type-module is defined to permit this. So ultimately the access-correctness of a system of programs rests on the type-modules in use: if the type-modules are trustworthy, no extra rights can be obtained, but if not, access control can be violated.

The fact that access control ultimately rests on the correctness of programs led to our restricting amplification to type-modules. This restriction ensures that the programmer knows exactly where to look to determine whether his programs will work as he desires. If, in addition, type modules provide as few operations as are necessary, then the amount of code to be examined is also minimized. A less stringent restriction on amplification, for example, to permit procedures outside the type-module to obtain additional abstract rights, would make the programmer's task much more difficult. Clearly, no power is lost by our restriction, since extra operations may be added to a type-module -- at the cost of the additional code to be inspected.

We have been careful never to state that rights are identified with operation names. In our examples so far, they have been used in precisely this way. We expect this to be usual. However, there are cases where this is not appropriate:

1. Some operations, most notably create operations like makemem for associative- memories, take no parameters of the type being defined; thus these operation names have no such corresponding rights.[7]

2. Some operations take more than one object of the type being defined, and require different rights for each object. For example, suppose file rights include "merge" and "mergeto"; and the file merge operation requires the following rights

   procedure merge (f: file {merge, mergeto}, g: file {merge})

   The procedure merges the contents of files f and g; f contains the result of the merge, but g is unchanged. Note that the procedure merge requires a special right to "mergeto" its first parameter where the results are to be placed.

It seems premature to make fixed rules about the relationships between operation names and rights. Practice will determine what is convenient. However, we expect a subset of the operation names will have corresponding rights. In unusual cases there will exist additional rights different from operation names. These will occur when operations treat parameters of the type being defined in different ways.

---

7. We assume that the right to use a type implies the ability to create objects of that type. This assumption is discussed in Section 4.

3. SHARING OF STRUCTURED OBJECTS

The access control rules described in the previous section provide control over the sharing of objects that are passed directly from one procedure to another. However, they are inadequate to control sharing of objects passed indirectly -- through the medium of another object. For example, suppose a number of procedures share a data base of employee records. Our rules can be used to control the sharing of the data base as a whole; it is a simple matter to grant read-only access to the data base. However, there is no way to also control access to the individual employee records stored in the data base.

In order to discuss this problem further, we must introduce a notation which permits us to talk about both the structure as a whole (the data base) and the elements of the data base (the employee records). The data type to be described is "data base of employee records" which is similar to data types already existing in programming languages such as "array of integers". The notation we will use is the following:

<data structure type name>[<element type names>]

Examples are

data-base[employee-record]
array[integer]

Both the data structure type name and the element type name(s) are the names of types, and so all can be qualified. To specify qualified structured types we will use the notation

T[Q1,...,Qn]{r1,...,rm}

where T is the type of the structure (for instance array, record or data-base) for which rights r1,...,rm are defined, and Q1,...,Qn are the qualified types of the n kinds of elements in the structure. In the following discussion we will limit ourselves to structures containing a single kind of element; this simplifies the discussion without loss of generality.[8]

---

8. Limiting what appears between the square brackets to just types is another simplification. It is easy to permit other compile-time-known quantities to appear between the brackets (for example, the selector names for components of records); an extension to quantities not known until execution time (for example, array bounds) can also be made, but at the expense of runtime checking.

Suppose that we wish to write a program, P, to scan a data base and calculate for each job category the average age of employees in that category. The program is not permitted to modify either the data base as a whole, or any of the employee records in the data base. In addition, there are a number of items in the employee records which may not be read, for example, salary information. Assume the rights to a data-base include 'read' and 'update', and the rights to employee-records include birth-date, read-job-category, write-job-category, read-salary, and write-salary. Further assume that all these rights permit the use of operations of the same name. Then the access control needs of procedure P to the data base can be expressed:

procedure P (d: data-base [employee-record{birth-date, read-job-category}]{read})

Another legitimate data-base type, one which might be used by a caller of P, is

e: data-base[employee-record{birth-date,read-job-category,read-salary}]{read,update}

We want the invocation P(e) to be legal. Intuitively, what we want is a binding rule that permits a structured object to be bound to a variable provided that the rights to the structure as a whole, and to the elements of the structure, do not increase. However, a straightforward extension of our binding rule, permitting the binding d ← e, is not possible for reasons explained below.

Just as with unstructured data, a data structure such as array or data-base may be characterized by a group of operations. For example, array operations of interest are arraycreate (which creates a new array of a given size), fetch (which fetches the ith element of the array) and update (which updates the ith element of the array).[9] However, a data structure is not a type; rather it is a set of types, containing a different type for each possible combination of element types of the structure. Thus, array is the set of types containing among other elements

array[integer]
array[string]

The types in this set of types differ from one another only in the kinds of elements the arrays

---

9. In the associative-memory example shown in the preceding section (Figure 4), we used the notation a[i] to stand for the invocation of fetch(a,i), and the notation a[i] ← x to stand for the invocation of update(a,i,x).

contain. Each type (in the set) is associated with a group of array operations that are specialized to work for the particular element type by an appropriate selection of types for their input and output parameters. For example, the parameter and return types of the operations for the type array[integer] are

```
procedure arraycreate (lb, ub: integer) returns array[integer]{all}
procedure fetch (a: array[integer]{fetch}, i: integer) returns integer
procedure update (a: array[integer]{update}, i: integer, s: integer)
```

Clearly it would be an error to attempt to perform an array[integer] operation on an array[string] object or vice versa. The operations for the two types differ in their input and output type requirements. In fact, the information about the element types of a particular data structure type is contained in the type requirements of the associated group of operations.

The above discussion has not taken into consideration any special requirements introduced by access control. In fact, all that access control introduces is a change in the possible element types of a data structure; in a language with access control, data structure element types may be qualified. Although this may seem like a very small difference, the consequences are profound. It means that

array[T{f1,f2}]

and

array[T{f1}]

are different types and it would be just as illegal to apply an operation of type array[T{f1}] to an object of type array[T{f1,f2}] (or vice versa) as it is to apply an operation of type array[integer] to an object of type array[string]. Therefore, it is not possible to make the desired binding d ← e (discussed above) because it would violate type-checking.[10]

---

10. The semantics of data structures and the motivation for this restriction will be discussed in a subsequent paper.

Extended Rule of Binding

Our extended rule of binding provides the desired access-control behavior, but avoids the type-checking violation described above. It permits a procedure to state precisely what limited rights it requires to all objects, including data structures and their component objects, and the procedure is restricted to exercising only those rights for which it explicitly stated a requirement. However, the extended binding rule does not require that the type of a variable be completely known. It allows the type to be partially specified. When this occurs, the precise type of a data structure object referenced by the variable is not known by the compiler. Nevertheless, the compiler can ensure that no erroneous assumptions are made about the type of a data structure object.

What is known about the type of a data structure object is that it contains at least those rights to elements required by the program. Consider the example of a procedure G which accepts as a parameter an array of elements of type T with f1 and f2 rights, with full rights to use the array. A call to G will be legal only if it is passed an object of type

array[R]{all}

where base(R) = T and rights(R) $\geq$ {f1,f2}. Thus R $\geq$ T{f1,f2}. We have just expressed exactly what is known inside G about the element type of the array. The notation actually used is

procedure G (a: array[?R $\geq$ T{f1,f2}]{all})

The '?' emphasizes that type R is not completely known when G is compiled; we will refer to types like R as ?types. The notation

$\geq$ T{f1,f2}

expresses exactly what is assumed about R inside of G.

For simplicity, we limit the introduction of ?types to formal parameter specifications in procedure headings. The key to understanding ?types is to understand what assumptions are made about such types inside a procedure which uses them. The assumptions made are very straightforward:

1. Every use of a particular ?type name is assumed to stand for the same real type.

2. Although this real type is not known inside the body of the procedure, a set of possible candidates for the real type is known; for example, assuming the rights for T are {f1,f2,f3}, then inside of G it is known that

?R ∈ {T{f1,f2}, T{f1,f2,f3}}

The binding rule is defined so that no extra rights can be obtained no matter which member of the set is associated with the ?type in the current invocation.

3. Two ?types with different names are not assumed to be comparable inside the procedure, even if they are drawn from the same set.

The association of actual type values with ?types is made at procedure invocation. The association must be done in such a way that the three assumptions discussed above are satisfied; only in such a case is the invocation considered legal. The most important is assumption 1, since it implies that even if a ?type name appears more than once in the procedure heading, it is still necessary to associate just one value with it. In fact, we require that a heading contain exactly one defining instance of a ?type name. Any other use of the ?type name must obey the constraints stated in the definition. An example is:

```
procedure H (a: array[ ?R ≥ T{f1,f2} ]{all}, b: array[ ?S ≥ T{f1,f2} ]{all}, t: ?S) returns ?R
```

An invocation of a procedure having ?types for some of its formal parameter types is checked for legality as follows. Each ?type is matched with the type of the actual parameter to be passed in the position where the ?type definition appears. This match can be done only if the type of the actual satisfies the constraints on the ?type, thus ensuring that assumption 2 holds. Next, the declarations for the formals, and for the return value, retval, are rewritten, replacing the ?types with the matched type values. Finally, the actuals are bound to the formals; if the bindings are legal (according to the rewritten declarations), and the use of the return value is legal, the invocation is legal.

For example, suppose the following declarations appear in the invoker of H:

x: array[T{f1,f2,f3}]{all}
y: array[T{f1,f2}]{all}
u: T{f1,f2,f3}
v: T{f1,f2}

The invocation u ← H(x, y, u) causes ?R to be associated with T{f1,f2,f3}, and ?S to be associated with T{f1,f2}. Then the formal declarations are rewritten:

a: array[T{f1,f2,f3}]{all}
b: array[T{f1,f2}]{all}
t: T{f1,f2}
retval: T{f1,f2,f3}

and the invocation is legal since all the bindings of actual to formals, and of retval to u, are legal (as will be shown below).

Note in the above invocation that the three assumptions are satisfied, and that ?R and ?S are associated with different type values. For this invocation of H,

?R ≥ ?S

However the legal invocation v ← H(y, x, u) would cause

?S ≥ ?R

and, therefore, assumption 3 is necessary.

Within a procedure body, ?types may be used to declare new variables in the usual way. For example, inside H,

c: array[?S]
v: ?R

are legal declarations.

Now we are prepared to extend our rule of binding to cover the additional cases introduced by data structures and ?types. We consider the binding

v ← e

where Tv and Te are the types of v and e, respectively; we wish this binding to be legal, as in Section 2, if we are certain that Te ≥ Tv. In the case of data structures, this is achieved by applying the rule of Section 2 directly: Te and Tv must be identical up to the rights on the

structure as a whole. For example, y ← x is legal if

x: array[T{f1,f2}]{all}
y: array[T{f1,f2}]{fetch}

or if

x: array[?S]{all}
y: array[?S]{fetch}

For bindings in which Te and Tv are unstructured but involve ?types, we make use of our intuitive understanding that a ?type can stand for any member of a set of types. Thus a binding involving ?types is legal only if it is legal no matter which member of the set is substituted for the ?type. For example, if

y: ?R ≥ T{f1,f2}

then x ← y is legal if

x: ?R

or if T{f1,f2} ≥ the type of x (e.g., x: T{f1}). On the other hand, y ← x is legal only if

x: ?R

or

x: T{all}

Discussion

The correctness of the extended binding rule rests on the correctness of the binding rule shown in Section 2. The ?type notation permits a name to stand for a set of types; the compiler can construct this set of types from the ?type definition. Whenever a ?type takes part in a binding, the compiler applies the rule of section 2 in the most stringent possible way, by requiring that the rule work for all types in the set.

The usefulness of the extended rule is demonstrated in Figure 6, which shows the implementation of a procedure, agesort, to sort an array of employee-records by employee age, using only birth-date access to employee-records. The procedure uses an array operation, size, to determine the current size of the array. A legal invocation of this procedure would be, for

example, agesort(b), where

b: array[employee-record{all}]{all}

A sorting example was chosen because sharing of the object being sorted is necessary, and because it must be possible to read an element from the object being sorted, and later to write that element back into the object. Observe how the use of ?R enables this activity (the interchange of the ith and jth elements of the array).

```
procedure agesort (a: array[ ?R ≥ employee-record{birth-date} ]{fetch,update, size}) =
  comment agesort sorts an array[employee-record] by employee age, using a bubble sort;
  index: integer{all} ← size(a);
  repeat
    bound: integer{all} ← index;
    index ← 1;
    for j: integer{all} ← 1 step 1 to bound-1 do
      if birth-date(fetch(a, j)) > birth-date(fetch(a, j+1))
        then begin
          temp: ?R ← fetch(a, j+1);
          update(a, j+1, fetch(a, j));
          update(a, j, temp);
          index ← j;
        end
  until index = 1;
  return;
end agesort
```

Figure 6. The agesort procedure.

4. COMPARISON WITH A DYNAMIC MECHANISM

We began the work reported here having observed that Algol-like scope rules were insufficient for controlling the use of shared data. We had also observed that the access control protection mechanisms available in operating systems did provide useful control over sharing of data. Could an analogous facility be of use in languages? The answer is affirmative -- even more so than we expected in that access control restrictions may be enforced at compile time. The next question is: how does enforcing access control using compile time checking impact the power of the language facility? To address this question, it seems appropriate to ask where and why language access control and operating system access control facilities are similar and different.

The language mechanism we describe is based on an access control facility defined in terms of capabilities. In a capability-based operating system all data is recorded in objects. Each object has a type that determines the accesses applicable to that object. A process can reference an object only by exercising a capability for it. Each capability specifies a unique object in the system and the accesses permitted on that object. We found that in an object-oriented language it was useful to think of a variable as a capability. Both are essentially "access paths" to objects, useful for exercising just those rights named in the capability or the variable.

To go into greater detail we compare our language facility to the specific capability-based protection mechanism found in the Hydra operating system [Jones75, Wulf74]:

1) Both facilities are object-oriented. Users can create arbitrary numbers of abstract object types and specify the accesses appropriate to objects of the type. Operations on objects are implemented as procedures. Most though not all extant systems other than Hydra limit access control to a small number of types of objects, mainly segments or memory blocks. In Hydra new types can be created dynamically; in the language types are user defined and are known at compile time.

We have already noted that both the language and the operating system facilities control access to an object on the basis of the access path (through the variable or capability). In the language facility, variables are all known at compile-time and their use is controlled by the scope

rules. Though the object bound to a variable may change, the type of the object and the rights permitted by access through the variable are fixed and known. In contrast, in the system the capability associated with a name may change dynamically, permitting arbitrary types of objects to be accessed. No restrictions on the capabilities associated with a name are enforced by the system access control facility.

2) Only operations applicable to an object (based on its type) can be performed on the object. In the language facility, such operations are implemented as procedures defined in the type-module. Experience with Hydra has shown that its access control mechanism encourages programmers to construct "subsystems" which are analogous to type-modules in a number of ways: Each subsystem defines a new abstract type (occasionally several new types) along with the accesses applicable to objects of that type and the procedures that implement the operations of the type.

3) To embed our access control facility in a language requires determining a policy for controlling the scope of type modules. This policy specifies in which program segments variables of the type can be declared. If variables of the type can be declared, the policy specifies which operations defined by the type module can be invoked. Whether invocation of such an operation fails due to insufficient rights for actual parameters objects is a different question.

In Hydra, access to a type's operations is controlled using the access control mechanism. A procedure itself is an object and one access defined for procedures is "call" access. Thus a user must obtain "call" rights to a procedure in order to invoke it at all. Similarly types are also objects in Hydra. To create a new instance of the type a user exercises the "create" right to that type. So in Hydra use of types and operation defining procedures are controlled by careful disbursement of "create" rights to type objects and "call" rights to procedures.

So far we have not discussed the kind of programming language in which our type-modules and procedures would be embedded but we need to do so to consider the policy for type module usage. One possibility would be a language like Pascal or Algol 60 in which the compiler compiles an entire program text, including the texts of all type-modules and procedures, at one time. Here,

scope rules determine the blocks in which a type module is known. Another, more promising, possibility is a modular language in which type modules are separately compiled to be stored in a data base. At compile or load time external modules are found in the data base together with their specifications; the compiler can use the specifications to determine the types of the formal parameters so that access control consistency can be checked at compile or load time. Accessibility of a particular type module would be determined by the policy implemented by the language support system which maintains the data base of compiled modules. For this presentation we have tacitly assumed the policy that all or nothing of a type module was available for use.

4) Both the language and the operating system access control facilities employ amplification. Here there is a substantial difference between the two facilities. In the language facility, amplification of rights to an object occurs (automatically) only at entry to a procedure defined in the object's type-module. This restriction is motivated by the desire to localize the code that determines the behavior of objects of a type to the type-module.

Amplification in Hydra is not restricted to operation invocation; it can also be performed explicitly by a user (having the appropriate "amplify" right). However, subsystem builders voluntarily adopt the same sensible strategy discussed for the language facility: when a subsystem creator creates an abstract type, he is given the right to "amplify", that is to increase, the rights for any object of the type. Note that he is permitted to amplify access to any object of the the type, whether it exists yet or not. Because this is a very powerful ability, it is closely held. In general, only procedures implementing operations applicable to the type are endowed by the type (subsystem) creator with the ability to perform such amplification.

Another contrast between our language notion of amplification and that of Hydra is one of degree. In the language, amplification always yields full rights to the parameter object and to its representation. In Hydra, amplification can be tailored to the different requirements of the subsystem procedures. Each procedure can be defined to gain (via amplification) only those additional rights required within that subsystem procedure. Though the finer control introduced

by tailoring may be conducive to enhanced correctness and protection, we believe that it would introduce excessive complexity of expression. We felt that brevity of expression took precedence for the language facility.

5) The Hydra and language access control facilities diverge most strikingly in their treatment of structured objects. Using the language facility, once a structured object is created the types of its component objects are fixed. The semantics of the extended binding rule ensures that no violation of these types is possible. However, restricted access to component objects of a data structure can be accomplished through the use of ?types. In Hydra, objects contain data and capabilities. Object A is considered a component of object B if B contains a capability for A. Hydra's access control facility does not enforce restrictions on the type or number of objects that can be components of other objects. In fact, the system access control facility provides a variety of operations that can be explicitly invoked to alter objects. Capabilities can be transported from one object to another, replicated and destroyed. (Destruction of the last remaining capability for an object implies destruction of that object.) Of course, invocation of such operations is controlled via capabilities. Hydra provides more dynamic power than the language facility and its access control mechanism alone is not sufficient to accomplish the automatic restricted access to component objects provided by the language facility using ?types. However, as usual, the additional restrictions can be implemented as user programs to provide the control dynamically.

6) As noted above, Hydra objects contain both data and capabilities, and the access control operations can be explicitly invoked by the user to move capabilities out of one object into another. Thus Hydra is a suitable facility for long term storage and retention of capabilities. To illustrate we consider a file system. In most cases a file is defined as a sequence of pages or blocks holding data. In Hydra, a file could be a sequence of objects of arbitrary type, represented as a sequence of capabilities. Such a file system is used very much the way we use bank safety deposit boxes, as receptacles for entities whose existence we wish to assure and whose access we wish to control. The bank that provides safety deposit boxes has no need to know what is kept in safety deposit boxes, and does not enforce any control over the type of entities kept in them.

To ensure the continued existence of any object a user wishes to retain between executions of a program or between terminal sessions, the user places a capability for it in the Hydra file system. Later when a user retrieves something from the file system he does not have a string of bits he can manipulate arbitrarily; he has a structured object on which he can only perform accesses applicable to the object's type. Note that this is in contrast to the language's stricter access control rules which restrict a program to accessing only components of an object if the types and accesses are known at compile time. In order to build a file system in our language, we require dynamic checking of the qualified type of an object retrieved from a file system. Such a dynamic check ensures that a retrieved object is of the type expected by the compiler. Such dynamic checks seem very straightforward and could be generated by the compiler where appropriate, using a union discrimination mechanism such as the conformity relations in Algol 68 [Lindsey73] or the "typecase" construct in CLU [Schaffert75].

In the above discussion we have shown that the language facility we propose and capability-based operating systems are quite similar in structure. Though the system which enforces access control dynamically seems more powerful than the language (using compile time checks), we found that many features are used in similar ways. Initially we expected that we could not do all protection checking at compile-time in the language and that we would have to embed some sort of escape mechanism to permit run-time checking. The requirement for such an escape is analogous to the need for type unions in a strongly typed language. We assume type unions to be available in the language to program applications like the file system discussed above. Whether anything further is required is an open question. We now suspect that type unions are a sufficient addition.

A system like Hydra and the language provide complementary facilities. The system run time checking could be used where dynamic checks are unavoidable. It would be possible to rely on the compiler to perform static checks, thus avoiding storage of unnecessary type information, and execution of unnecessary instructions, or the system could be used as a backup to the compiler with dynamic checking ensuring the correctness and reliability of the compiler and to some extent the

hardware. In addition the system would provide support for the storage of already compiled type modules and perhaps other user created objects. Finally, we believe the user will find it a pleasure to program and to execute in the consistent homogeneous environment which results when language and operating system are structurally similar.

5. DISCUSSION

A major premise of this paper is that it is important for programmers to express restrictions on the manipulation of abstract objects and to express these restrictions precisely, in abstract terms meaningful to the objects. We have proposed an access control facility suitable for extending object-oriented languages to achieve this goal. Access control is provided using variables, objects, type-modules and binding. At any point in the execution of a program the variables that can legally be named provide access paths to the objects that can be manipulated. Different programs (possibly written by different users) may share the same object; each has a different variable bound to the shared object. Thus, control of the manipulation of an object is based on the variable (the access path) used to identify the object.

The access control facility definition includes a binding rule which determines how variables can be bound to objects and, thus, how access to a new object can be obtained. This binding rule is based on several premises:

1. Object sharing is desirable.

2. The acquisition of access paths to an object should not permit a user to gain additional access to that object, i.e., programs should be "access-correct".

3. Only operations applicable to an object (as determined by its type) should be performed on it.

4. Access control should be checkable at compile time.

Control of access to both unstructured and structured objects is provided; for structured objects, access to the components can also be restricted.

We believe that a mechanism such as ours is a worthwhile addition to a language used for writing applications programs to run on modern operating systems; the systems themselves would also benefit from being programmed in such a language. Modern systems enable different users (often mutually suspicious users) to share data in a controlled way. Our mechanism permits the real-world activity of controlled sharing to be expressed in the programs which are doing the sharing. In addition, the static checking we provide not only eliminates run time access errors, but

raises the possibility that controlled sharing can be accomplished with lower cost than is currently possible. For the present, it is safer to retain dynamic checking, to back up the compiler (which is a complex and probably unverified program), to execute the programs which were not processed by the compiler, and, if appropriate, to enhance reliability with (possibly duplicate) run-time access control checks.

Of course, a static mechanism like ours can be considered as a replacement for a dynamic mechanism only if no useful programming power is lost in the process. We have considered this question in Section 4, where we compared our mechanism with the dynamic access control mechanism present in the Hydra system to determine whether the extra flexibility in the system mechanism permitted useful programs to be written that could not be written in the language. Our analysis indicates that sensible use of the system mechanism leads to program structures very similar to those in the language. In fact, our mechanism appears to be superior to the dynamic one, not only because static detection of access control errors is preferable to dynamic detection, but because certain types of restricted access control (restricted access to elements of a shared data structure) can be provided in the system only at the cost of the additional programming complexity. However, we recognize that there are cases where an escape to dynamic checking is necessary. More study is required to establish where such escapes are needed and how they should be provided in a programming language. An escape is needed to build the file system; here we assume the existence of run-time access control checks that are analogous to those needed in the presence of type unions.

The objective of an access control facility is to be able to restrict the access to objects according to some policy. One should then ask if an access control facility allows one to write access restrictions so as to implement policies of interest. This is a difficult question in language access control just as in systems. One accepted policy or guideline is the "need to know": it should be possible to implement programs so that an object is accessible in only the ways appropriate and necessary to the function currently being performed. Our facility permits tailored accessing environments to be designed and implemented. In fact, "need to know" is the policy which

underlies our access control mechanism; each program states what rights it needs, and the mechanism ensures that no additional rights may be obtained. We do not know whether the language mechanism could support other sensible access control policies. However, this question is an unresolved one for dynamic access control mechanisms as well.

Clearly, an access control mechanism such as ours is useful when programming in an environment in which data objects are shared among users. However, we believe our mechanism is worthwhile even when programs are not intended to run in such an environment. Our mechanism enables the programmer to declare restrictions on the accessing behavior of his programs, and these restrictions are enforced by the compiler. We believe this will lead to benefits similar to those provided by strongly-typed languages: the mechanism permits and encourages programmers to structure programs so as to reduce the space of possible errors. Enhanced ease of producing correct programs can be expected as a result.

ACKNOWLEDGEMENTS

We would like to express our appreciation to our colleagues, particularly those working on CLU and Alphard, whose criticism and observations have helped us prepare this paper.

REFERENCES

[Dahl72] Dahl, O.J. and C.A.R. Hoare, Hierarchical Program Structures. Structured Programming, (ed. Dahl, Dijkstra, and Hoare) Academic Press, 1972

[Dennis66] Dennis, J. and E.C. van Horn, Programming for Multiprogrammed Computations. CACM 9, 3, 143-155 (1966)

[Jones73] Jones, A., Protection in Programmed Systems. Ph.D. thesis, Carnegie-Mellon University, 1973

[Jones75] Jones, A. and W.A. Wulf, Toward the Design of a Secure System. Software Practice and Experience, V5, 321-336 (1975)

[Lampson71] Lampson, B.W., Protection. Proc. Fifth Annual Princeton Conference on Information Sciences and Systems, Princeton University, 437-443 (1971)

[Lindsey73] Lindsey, C.H. and S.G. van der Meulen. Informal Introduction to Algol 68. North Holland, Amsterdam-London, 1973

[Liskov76] Liskov, B., An Introduction to CLU. Computation Structures Memo No. 133, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., 1976

[McCarthy62] McCarthy, J. et al., LISP 1.5 Programmer's Manual. MIT Press, 1962

[Schaffert75] Schaffert, C., A. Snyder and R. Atkinson, Clu Reference Manual. Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., 1976

[Sturgis74] Sturgis, H.E., A postmortem for a time sharing system. Ph.d Thesis, University of California at Berkeley, 1974.

[Wirth71] Wirth, N., The Programming Language PASCAL. Acta Informatica, 1, 335-63 (1971)

[Wulf74] Wulf, W. A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and R. Pollack. HYDRA: The kernel of a multiprocessor operating system. CACM 17, 6, 337-345 (1974).

[Wulf76] Wulf, W. A., R. London and M. Shaw, Abstraction and Verification in Alphard, to be published.