ACCESS CONTROL IN PARALLEL PROGRAMS

James R. McGraw
and
Gregory R. Andrews

TR78-328

Department of Computer Science
Cornell University
Ithaca, NY 14853

ACCESS CONTROL IN PARALLEL PROGRAMS

James R. McGraw*

Gregory R. Andrews

Department of Computer Science

Cornell University

Ithaca, N.Y. 14853

An important component of a programming language for writing operating systems, or other large parallel systems, is the set of access control facilities. Two principles for access control, expressive power and access validation, are discussed. Then two new language mechanisms are presented: one for expressing the static structure and access rights of parallel systems, the other for controlling dynamic access to shared objects (monitors). The use of the proposed mechanisms is illustrated by message passing and file systems. Finally, the relationship between the mechanisms and access validation is discussed and a solution to the safety problem for the facilities is given.

Key Words and Phrases: access control, programming language, protection, security, processes, monitors, access safety

CR Categories: 4.20, 4.32, 4.35

This work was supported in part by NSF grants MCS 74-41115 and MCS 77-07554.

* Current address: Computing Science Group, Department of Applied Science, University of California at Davis, Livermore, CA 94550

1.0 Introduction

In an effort to both increase the reliability and correctness of concurrent systems and decrease the cost of their design and implementation, many people have been working on the development of parallel programming languages [1,2,4,8]. The initial focus has justifiably been upon the central problem of all concurrent systems, namely processes and their interaction. A second important problem is access control. If we are to develop reliable and secure software, it is important that each module only have access to exactly those objects it requires. This leads to the need for language facilities to specify the static (permanent) and dynamic (temporary) access rights of processes and procedures. This paper proposes two flexible yet simple language features: a grant declaration for expressing the static structure and permanent access rights of a system of parallel processes, and capability variables for controlling dynamic access to shared objects (monitors).

In the remainder of the paper, we present our proposals in detail, illustrate their use, compare them to other approaches [2,5,6,8], and discuss their relation to access security. Before proceeding, however, we discuss the two principles, expressive power and access validation, that guided their development and against which they or any access control facilities should be evaluated. For the purpose of illustration, we use Pascal [7] augmented by processes and monitors as the language to which our access control facilities are added. However, our principles and proposals apply to many structured languages for parallel programming (e.g. Concurrent Pascal and Modula).

2.0 Principles

A programming language provides a set of mechanisms for implementing systems. Using the mechanisms, one implements algorithms and organizations which enforce policies. In our opinion, 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;

(2) access validation - they should allow the implementation of an access policy to be validated.

Two types of access control mechanisms, static and dynamic, occur in parallel systems. The purpose of static mechanisms is to increase reliability and reduce unwanted interference by controlling the objects that each subject sees. In terms of processes and monitors, this means that each process should only have access to those monitors which it needs to perform its intended task. In a dynamic system, such as a file system, objects come and go; consequently, access paths come and go. The purpose of dynamic access control mechanisms is to allow these changes to be represented and efficiently implemented.

An expressive set of access control mechanisms should make it possible to describe clearly a variety of access policies. In particular, it should be possible to:

(1) limit each subject to those objects and operations which it needs;

(2) insure that only meaningful and authorized operations are applied to objects; and

(3) control the order and timing of access to controlled objects.

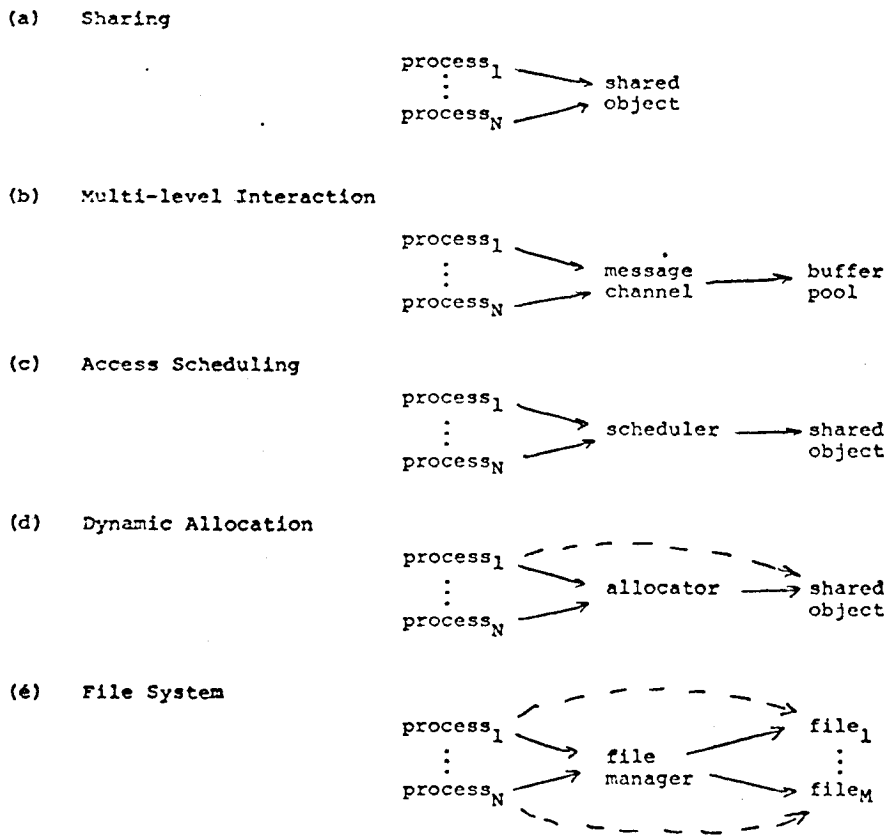
The first two points have been called "access correctness" by Jones and Liskov [5]. To this we have added a third criteria to capture the fact that an access policy for dynamic objects is often concerned with the timing of access. Examples of such policies are: access must be scheduled before it occurs; allocation must precede access; and access cannot occur after an object has been released.

As an illustration of these points, Figure 1 presents the abstrac structure of five common situations in parallel systems. Part (a) shows the structure when several processes share an object such as a device or message channel. Possible access policies in this case are restricting use of the shared object to a subset of system processes and, possibly, restricting each process to a subset of the operations on the object. For example, some processes may be able to send but not receive messages over a specific channel. Part (b) shows a message channel which may be implemented by a queue of buffers acquired from a common pool. A policy here may be that processes cannot directly access the buffer pool, only message channels can. This is one example of a multi-level interaction. Another example, device scheduling, is shown in Figure 1(c). Here the policy might be that all processes must go through the scheduler in order to access the shared object.

The first three structures of Figure 1 are static in nature; the next two are dynamic. In part (d), an allocator controls access to a shared object. Processes request access from the allocator; when their request is granted, they receive a dynamic access right that enables them to directly (hence efficiently) access the object. Examples of access policies in this case are that an object must be

Figure 1

System Structure and Access Control



requested before it can be accessed, that once requested it can be accessed directly, and that once released it can no longer be accessed. In part (e), a file system is modelled. Here many objects (files) may be controlled by one manager. A process may potentially have access to many files at once. The access policy may be that a process can be restricted to a subset of the file operations, that the creator of a file controls all access to it (via the file manager perhaps), and that shared access can be revoked at any time by the owner of the file.

An expressive set of access control mechanisms should allow any or all of the above access policies to be implemented by a system programmer. This is our first guiding principle. A systems' users, however, are concerned with adequate enforcement of a specific set of stated policies. In particular, it is desirable to validate the correctness of an implementation of the stated policies. This is our second guiding principle.

In order to validate an implementation, it is first necessary to solve the access safety problem. As defined in [3], the safety problem is concerned with deciding whether a given protection system can lead to a "leak". Here we will take a more global view and define access safety to be the problem of determining the potential access rights of every subject for every object. A program in a language defines, via declarations, a static set of subjects and objects. It may also define, via type declarations, the patterns for dynamically created objects that result in dynamic access paths. Referring back to Figure 1, the static access rights, represented by solid arrows, specify the structure of the different interactions; the dynamic access

rights, represented by dashed arrows, specify the use of objects and can vary dynamically.

In order to solve the safety problem, we need to be able to determine the access rights that each subject could potentially acquire. For static rights this can be readily determined at compile time. Dynamic rights present a difficulty, however, since they change as a program executes. Our approach is to model the execution of a program in order to determine the maximum potential access of each subject, namely the access which could occur if each execution path were taken. This information can then be used to answer access policy questions such as "Can process P access file F?" If the answer is yes, our access model also determines the access path that must be followed by the process. So even if the process is not supposed to be able to access the file, we can pinpoint the spots where leakage could occur. It is then necessary to validate the correctness of the potential offender by using program verification techniques.

3.0 Static Control System

The most common scheme for representing access rights in a program is through static access controls (i.e., scope rules). Unfortunately, in most languages the default access rights of a block are far too powerful and cannot be easily restricted. As a result, it is difficult to enforce access policies that prevent harmful or unexpected interplays between modules. Our system for providing static access control is based on modifying the common Algol-like approach to scope rules. Each program object's initial visibility is severely limited. It can, however, be expanded through a new grant feature

which allows one to explicitly state static policies to be enforced by a compiler.

3.1 Basic Description

Our default scope rule is that each program block (process, monitor, or procedure) may only access those objects (e.g., variables, types, monitors, or procedures) which it defines. This approach is nearly the opposite of the one used in Algol, PL/I, Pascal, etc., where a block may, by default, access any object declared in a surrounding block. There, the default access rights are by definition the maximum allowable rights. In contrast, our default provides the minimum reasonable set of access rights so that any more general policy must be implemented explicitly.

The grant declaration permits an object's visibility to be selectively expanded. Its syntax is:

grant <object_list> to <block_list>

Each block named in the block list* is given permanent access to all objects specified in the object list. Grant can be used to give a block access to any program object such as a variable, type, procedure or monitor. With monitors, which in general define more than one procedural operation, each operation must be explicitly granted. The syntax used is:

grant monitor name {operation_list} to <block_list>

---

* Each list is a sequence of one or more identifier names, separated by commas.

Grant is not an executable statement and it must appear with the declarations at the beginning of each block. In addition, a block may only grant access rights to blocks which it defines. However, grant may be applied at successive levels to pass an object to an inner block. In this paper, we will not impose any restrictions on which objects can be given to the various block types. In practice, however, it is generally desirable to prevent processes as well as monitors from directly accessing global variables (for example, see [1,2, or 8]).

3.2 Applications

A small example will illustrate the general usefulness of this scheme. Assume we have a Message monitor and two system processes, Job_scheduler and Spooler, that use Message to exchange information. One grant with two parts provides the appropriate access rights as shown in Figure 2. Each process is able to use directly the Message operation it needs without having to take any actions or even acknowledge receipt of the rights. If this is the only grant for Message, other program units (like user processes) cannot interfere because the default scope rule prevents them from accessing the monitor.

More complex policies can also be expressed with this approach. Consider the problem of using a type declaration to define a set of monitors for controlling remote terminal I/O. Users should be allowed to access the instances defined by the operating system; however they should be prevented from defining any new instances. In many languages this is impossible to enforce because users need to specify the type of any object they acquire; hence they could use the type to declare

Figure 2

Simple use of grant

```
System : begin
        monitor Message;
                operations send,receive;
                   :
                end Message;
        grant   Message (receive) to Job_scheduler,
                Message (send) to Spooler;
        process Job_scheduler;
                   :
                Message.receive ( );
                   :
                end Job_scheduler;
        process Spooler;
                   :
                Message.send ( );
                   :
                end Spooler;
        process User;
                /*no access to Message*/
                   :
                end User;
        initialization statements
        end System
```

more instances. The grant feature solves this problem by letting us give each user direct access to a remote terminal monitor. Since the user need not explicitly acknowledge receipt, he need not have the type name available to him. Figure 3 illustrates this point. Since no grants are applied to type Terminal, users cannot access the type and therefore cannot declare new terminals which are unknown to System. On the other hand, the grants applied to Term1 and Term2 permit users to access the terminals defined by System.

We can illustrate the usefulness of nesting by extending this example further. If User1 contains a number of blocks, he may not want all the blocks to be able to do I/O directly. Grant permits User1 (or any user) to control the propagation of access rights within his scope. Consequently, User1 could grant Term1 to only those sub-blocks which are supposed to access the terminal. Erroneous attempts by other blocks to use the terminal will then be detected at compile-time.

3.3 Comparison With Other Approaches

Two approaches for handling similar problems in parallel programs have already been discussed in the literature. Concurrent Pascal [2] allows each subject to access directly only those objects it declares. Parameter passing is used at initialization to specify the interconnections of subjects (this allows any kind of object, specifically monitors and processes, to be used in a type declaration). Since the receiver must have formal parameters (identified by type) to acquire access to external objects, he must have access to the object's type name. Hence, this approach makes it difficult (if not impossible) to

Figure 3

Controlling the use of type declarations

```
System : begin;
         type    Terminal = monitor;
                            operations read,write;...
                            end;
         var     Term1, Term2 : Terminal;
         grant   Term1  {read,write} to User1;
         grant   Term2  {read,write} to User2;
           :
         process User1;
                   :
                   Term1.read (  );
                   :
                 end User1;
         process User2
                   :
                   Term2.write (  );
                   :
                 end User2;
         end System
```

permit a user to access objects without allowing him to create new instances as we did with terminals in Figure 3. Also, Concurrent Pascal does not allow either selective or nested control over individual monitor operations. Although it could be modified to do so, care would need to be taken in order to avoid the necessity of run-time checks. Concurrent Pascal is intended for programming fairly small systems; our proposals apply to all sizes of systems and therefore generalize some of the Concurrent Pascal concepts. In particular, in the next section we present a more general approach to passing access rights by parameter.

Another method for specifying static access control is Modula's use list [8]. In Modula, each block identifies all objects that are imported for use within the block's range. Although this feature was not really intended for enforcing security policies, it can be used by introducing extra dummy blocks into a program. The dummy blocks employ use to specify the objects an inner block can then further import. Conceptually, the problem with use for enforcing access policies is that an inner block takes what it needs, rather than being told what it can use. For specifying security policies, grant is more appropriate.

In summary, our approach to static access control is based on a desire to incorporate security policies directly into programs. The default scope rule forces a designer to specify his access policy explicitly through the grant feature. Any errors of omission in detailing that policy cannot lead to security violations because no accesses are granted by default. Furthermore, since policy specification is accomplished with scope rules, enforcement and implementation

can be handled entirely within the compiler where only the symbol table routines require significant modification. But the most important advantage is that this scheme is conceptually simple yet expressiv enough to implement a wide range of policies not easily handled with existing mechanisms.

4.0 Dynamic Control System

In systems applications there is often a need for dynamically allocating rights for objects such as files or buffers. The problem is to allow movement of rights among different processes, in contrast to movement within one process environment (which can be handled by standard procedure calls and parameter passing). For example, if a file is defined as a monitor, how can we "give" it to some process which will use it and later return it to us? The solution we propose is based on the concept of a capability [9]. All dynamically-controlle objects (in our case monitors) are referenced and managed solely through capability variables, which name objects and access rights. Access rights are transmitted between processes by passing capabilities as parameters to and from other monitors (such as allocators). This system permits a wide range of dynamic access policies to be expressed, and it also combines well with the static system.

4.1 Basic Description

In order to control a monitor dynamically, it must first be described in a type definition, using the attribute dynamic. For example:

```
type File = dynamic monitor;
            operations read,write;
            < normal internal structure >
            end
```

Dynamic indicates that all instances of File will be accessed solely through capabilities. Except for the manner of access, dynamic monitors are identical to the standard (static) ones.

Capabilities for referencing monitors are declared in the same way as other variables:

```
var My_File : File capability
```

A capability variable contains two items: (1) a reference to a monitor and (2) a list of access rights (i.e., monitor operations) currently authorized for that monitor. Initially each capability is empty. Dynamic monitor instances are created by executing a new type of statement:

```
My_File := File.create
```

This create command generates a new File monitor and puts a reference and full access rights for it in My_File. If a dynamic monitor type is parameterized (e.g. to specify the size of a file), actual parameters are passed to the new instance by create.

The notation for accessing dynamic monitors is similar to that for standard monitors; the underlying mechanism is more complex, however. We can access the file just created by the call:

```
My_File.read ( )
```

Each time a call is made using a capability, the rights list of the capability is checked for the particular operation (in this case read). If the right is present, the monitor currently referenced by the capability is invoked; otherwise the call is undefined (in practice it should result in a trap similar to an arithmetic exceptic

Access rights for monitors can be manipulated by performing operations on capabilities. The most important operations are:

(1) assignment - Your_File := My_File {read} and

(2) parameter passing - Output (My_File)

The assignment operation permits several capabilities to share access to a monitor, with the possibility that the actual access rights diff between them. The left side capability variable receives a copy of t contents of the right side capability variable except it receives onl those access rights listed in the brackets. The constraints are that the granting capability must have all rights it is transmitting and further, it must have the language defined copy right. This righ is the one special right stored in a capability variable used in a create operation. Its purpose is to allow a subject, for example an allocator, to control the number of outstanding capabilities for a dynamic monitor. Copy itself may be assigned with other rights, as the programmer chooses.

Capabilities can be moved between subjects (e.g. processes and monitors) by passing them as parameters. The rules here are a variation on the value-result rules. On a call, the capability's reference and access rights are transferred from the actual to the formal parameter (i.e., the actual parameter is emptied); on return the reverse procedure is followed. (The copy right is not required

to pass a capability as a parameter.) The purpose of this approach is to permit processes to acquire and release access rights to monitors while simultaneously preventing parameter passing from being used to make more copies of some access right. Within a procedure a capability parameter can be assigned null; this erases the value in the actual parameter when the procedure returns. It is used by allocators to take access away from a process once the process releases a monitor.

4.2 Applications

Two different designs of a file system serve to illustrate the power and flexibility of these dynamic access controls. Figure 4 sketches a simple program organization that permits Users to create and manage their own instances of files. The operating system defines the type File, thus insuring that Users cannot directly access the Disk and gain illegal access to other data. However, Users can create their own files using the type name File, which is known to them. Further, each User is the sole manager of his files - not even the operating system can acquire a capability for a file without the User explicitly giving away the rights.

A more complex access policy for a file system is illustrated in Figure 5. Here, the operating system creates and manages access to all files through a Supervisor monitor. It retains the master capability for each file (in this case only one called Sysfile) in its permanent data area. When a User requests the file, he passes in an empty capability and receives a copy of the master giving him read access. Notice that the User does not receive the copy right. When a User calls release, he passes in a full capability which is then

Figure 4

User managed file system

```
System :  begin
          type  File = dynamic monitor ( ):
                       operations read,write;
                            :
                       procedure read ( );
                                 :
                            Disk.read ( );
                                 :
                            end
                       :
                       end;
          grant File to (User1,...,UserN);  (*grants type name, hence
                                             ability to declare File
                                             capabilities*)
          monitor Disk;
               operations read,write;
                    :
               end Disk;
          grant Disk {read,write} to File; (*grants Disk to all instan
                                             of File*)
          process User1;
               var My_File : File capability;
                    :
                   My_File := File.create ( );
                    :
               end User1;
             :
          process UserN (*same access rights as User1 - can do the
                          same things*)
                    :
                  end UserN;
               :
          end System
```

nullified so that on return his rights are gone.

The organization of this solution is critical for insuring that users cannot create files. The default scope rules prevent each User from accessing the File type name; since that name is needed for creating instances, users cannot create files. However, we need to allow users to access files and for that purpose they need to be able to define capabilities. The declaration and grant of Filecap fills this requirement.

One weakness in the simple file scheme outlined in Figure 5 is that the Supervisor cannot be sure whether a User is actually returning his file or is instead returning an empty capability. A useful extension to our capability system are two Boolean primitives for interrogating the contents of a capability:

object(cap1, cap2) - true if cap1 and cap2 reference the same object; false otherwise.

rights(cap1, rights list) - true if cap1 contains all of the rights in rights list (it may contain more); false otherwise.

With these extensions, Supervisor could check to see exactly what was being returned by a User before nullifying his rights. Such a check would be particularly important if Supervisor were managing several files instead of just one and were allowing users to share files.

4.3 Comparison With Other Approaches

Two other language facilities have been proposed to permit dynamic allocation of objects. Silberschatz et. al. [6] introduced the manager construct which is actually a new type of monitor with special

Figure 5

Supervisor managed file system

```
System : begin
        type File = dynamic monitor ( );
                    operations read,write;
                       :
                    end;
        type Filecap = File capability;
        grant File to Supervisor;
        grant Filecap to (Supervisor, User);

        monitor Supervisor;
              operations request, release,...;
              var Sysfile : Filecap;
              grant Sysfile to request, release;
              procedure request (var id : Filecap);
                        id := Sysfile {read };
                       end request;
              procedure release (var id : Filecap);
                        (*check validity - not shown*)
                        id := null /*empties id*/
                        end release;
              other operations;
              (*initialization - create file*)
              Sysfile := File.create ( )
              end Supervisor;
        grant Supervisor {request,release } to User;

        process User;
              var Sysfile : Filecap;
                  (*User cannot create a file because User was
                   not granted File*)
```

Figure 5 Continued

Supervisor managed file system

```
        begin
                :
            Supervisor.request (Sysfile);
                :
            Sysfile.read (  );
                :
            Sysfile.write (  );
                :
            Supervisor.release(Sysfile);
                :
        end User;
        :
end System
```

internal primitives. This feature is very high-level, and as a result imposes a significant amount of policy. For example, only the manager for some object type may alter any process' access rights for instances. Hence, managers could be used to solve our second example (control of system files - Figure 5) in a relatively straightforward manner by modifying Supervisor and making it a manager. However, managers cannot directly solve the first problem where User processes control access rights for their files. We feel that our approach is less complicated (while covering their applications), imposes fewer policy restrictions, and is easier to understand and use.

The other proposal, by Jones and Liskov [5], is similar in nature to ours. All access to dynamically controlled objects is through capability-like variables; but in their case each variable has its access rights (qualified type) set at declaration. Hence all use and movement of rights (via assignment or parameter passing) can be checke at compile-time to insure access correctness. One drawback of their approach is that if many different sets of access rights are needed for some object, at least one new capability must be defined for each set. In particular, they state in [5] that their static mechanism cannot be used to build a file system. Our system, on the other hand, requires run-time checking in the form of a bit-vector test. This test is limited to capability assignments and dynamic monitor calls, however, and costs very little. In our opinion both approaches are viable; the main difference is that Jones and Liskov focus on sequenti object oriented languages (e.g. CLU and ALPHARD) whereas we focus on parallelism and dynamic control.

Our dynamic access control scheme is intended to be a basic tool

for building complex policies into a system. As such, we make no attempt to embed policies (such as ownership) into the features. The capability concept has often been used to describe dynamic access control, but almost always from a hardware viewpoint. Here we have shown that it also is useful in software. Although we have assumed that capabilities are only used with dynamic monitors, they can also be used with dynamic processes in an analogous way - the rights would authorize whatever process operations, such as activate, that the language defines. Having both dynamic processes and monitors, we could easily describe a spooling system such as the one programmed in Concurrent Pascal in [2]. In parallel programs, we cannot immediately use capabilities for other types of objects, however, because of the potential for simultaneous access to shared variables. Only monitors and processes have built in exclusion. With suitable restrictions, capabilities could potentially be applied to other types of variables; one such proposal for safely sharing variables such as buffers is described in [1].

5.3 Access Security

As argued in Section 2, true protection requires the validation of a system's access policy. One key to accomplishing this goal is solving the safety problem which has been described by Harrison, Ruzzo and Ullman [3]. We now present a solution to the safety problem for our access facilities. Even though the solution is simple, it is important both because it shows that the problem can be decided for a reasonable class of systems (an open question in [3]) and because it enables us to validate some security policies. However, we also

show that being able to solve the safety problem does not guarantee that we can validate every security policy; hence, we have only parti reached our initial goal.

The safety problem is normally formulated in terms of leaking access rights to untrustworthy subjects. A protection system can be described in terms of subjects, objects, generic rights, protection commands, and protection states. An invocation of a protection comma can change some subject's access rights for an object, thus altering the current protection state. Such a change is defined to be safe only if the newly acquired rights cannot be used in conjunction with the protection commands to give away access rights in the future. If we can decide safety for every invocation of a protection command within a specific system, then we have solved the safety problem for that system. Clearly, the heart of this question is deciding what each subject could potentially do in the future.

The safety problem has a fairly natural mapping into a programmi environment. A language defines a class of protection systems and each program is a specific instance. The subjects are the program units that define the various execution environments - in our case each process, procedure, or monitor block constitutes a different subject. The objects of interest in a parallel system are the progra units that can be shared, namely monitors and procedures. (We assume here that processes cannot access shared variables directly.) Final the protection commands of a program are its uses of grant, capabilit and parameter passing since they are the only tools for changing the access rights of any subject.

Given this mapping, we need to compute the direct access rights

that each subject could acquire for every program object. By direct access rights, we mean those rights that a subject can use immediately without first changing environments. For example, if a process must call a monitor to access a disk, we need only know that the process can call the monitor and that the monitor can use the disk. If we have the information on direct access rights, it should be clear that we can compute full access rights (direct plus those resulting from procedure calls) and hence solve the safety problem.

Direct access rights in our programming environment can only be acquired in two ways - statically via grant or dynamically via capabilities. The static ones create no problems because the scope rules and explicit nature of grant make all such accesses trivial to compute. The difficulty is in computing the maximum possible dynamic rights of each subject. Our solution here is to model each subject's use of capability variables. The graph we construct will identify the maximum flow of rights between each pair of capabilities. So when a new object is created (and the reference stored in a capability) we can use the graph to determine where rights could propogate; we cannot distinguish between different objects created at different times with the same capability, however.

The algorithm is actually very simple. Each node in the graph represents an instance of a capability in the program. The directed arcs represent possible access right transfers between nodes. In addition, the arcs are labeled to indicate the type of rights that could move. The algorithm has the following four steps:

1. For each capability variable C which is not a formal parameter, add a node C to the graph. If C is a formal parameter add

a node C_i for each differenct place in the program where C could be passed rights via a call.

2. For each call that passes a capability (between P and Q_i) add two arcs $P \xrightarrow{all} Q_i$ and $Q_i \xrightarrow{all} P$ where all stands for all of the possible rights for the object referenced by P and Q_i.
3. For each capability assignment Q := P {rights list}; add arc P → Q with the rights list as the label.
4. Compute the transitive closure of the graph constructed in the previous steps; namely, for each path from A to B, add an arc A → B, where the label is the intersection of the labels on the path (labels are regarded as sets of rights).

The first three steps set up the basic information on rights movement. Rights can flow from A to B only if a path exists in the proper direction. The final step consolidates this data so path searches are not necessary to decide flow between pairs of nodes.

We illustrate this algorithm on a very simple system which passes messages. Figure 6 outlines code for a program that has three communicating processes. Sender1 and Sender2 transmit messages (instances of the dynamic monitor Message) to Receiver through the static monitor Channel. In Figure 7, parts A and B show the access graph before and after closure. The final graph verifies that access rights only flow from the two senders to the receiver. In particular, the senders cannot exchange rights. (Notice that we could not verify this if we had only one node for each formal parameter, specifically in; we must distinguish between each call.)

The strength of this model of our language facilities is that

Figure 6

Outline of message passing system

```
System : begin
        type Message = dynamic monitor;
                       operations read,write;
                            :
                       end;
        grant Message to (Channel,Senderl, Sender2, Receiver);
        monitor Channel;
                operations send,receive;
                var store  :  Message capability;

                procedure send (in : Message capability)
                            :
                          store := in {read,copy};
                            :
                          end send;
                procedure receive (var out : Message capability);
                            :
                          out := store {read};
                            :
                          end receive;
                   :
                end Channel;
        grant Channel {send} to (Senderl,Sender2);
        grant Channel {receive} to Receiver;
        process Senderl;
                var ml : Message capability;
                begin ...
                          ml := Message.create;
                          ml.write ( )
                          Channel.send (ml);
                end Senderl;
```

Figure 6 Continued

Outline of message passing system

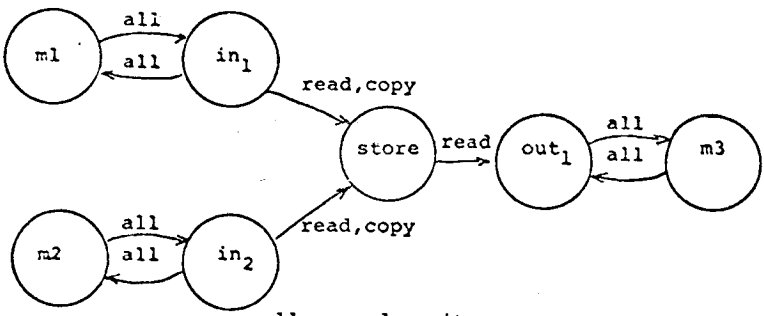
```
        process Sender2;
                var m2 : Message capability;
                begin ...
                        m2 := Message.create;
                        m2.write ( );
                        Channel.send (m2);
                end Sender2;

        process Receiver;
                var m3 : Message capability;
                begin ...
                        Channel.receive (m3);
                        m3.read ( );
                              ...
                end;
              ...
end System
```

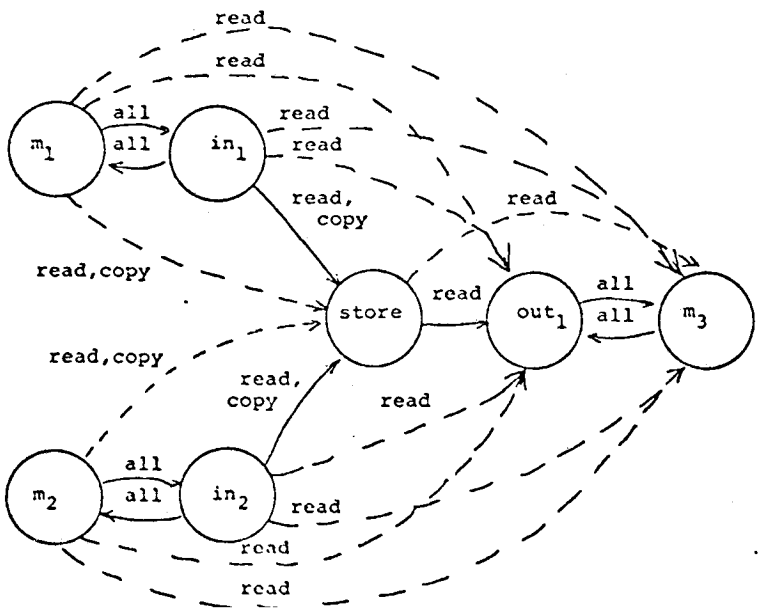
Figure 7

Capability flow graph for program in Figure 6

A. Direct flows caused by assignments and parameter passing.



B. Closure of flows on graph in Part A.



it is relatively inexpensive to compute the flow of rights. It can be done once, at compile-time, and then used for several types of analysis. Also, the information on maximum flow could be very close to the actual flow, depending on the system design and required protection. For example, all of the flows in Figure 7 are almost certain to occur. Hence it could be a practical tool for a system designer to validate his access policy.

The major weakness goes back to the definition of safety, which ignores program code. For an analysis such as we have developed, we must know a great deal about a subject's code - such as which capabilities are used in calls to the various monitors. This requires that we examine the entire program, and yet we do not use any information about the actual algorithms. Conceptually, our flow analysis assumes that a subject first acquires all rights and then gives away as much as possible. The program may actually do considerably less. Consider, for example, the message system of Figure 6 with one more receiver process. The flow graph would show full transmission through Channel from both senders to both receivers, even if Channel were coded in such a way that each sender is actually tied to only one receiver. The same problem would result if the file Supervisor of Figure 5 were extended to manage files for several users; the flow graph would show that each user could potentially access each file. This is not surprising, however, since a potential flow scheme cannot validate policies implemented in program code. The potential access graph for the modified message or file system would identify the source of the problem but it remains to prove formally that an unintended transfer of access does not occur. For the file system

this means that the correct file is returned; this obviously requires that each process (or user) is correctly identified.

6.0 Summary

At the beginning of this paper we identified two principles, expressive power and access validation, for access control mechanisms. An expressive mechanism should make it possible to limit the access of each subject to those objects it needs to know, to restrict the set of allowed operations, and to control the order and timing of authorized actions. Our static access control facility, grant, allows the use of static objects and capability variables to be controlled on a need-to-know basis. Individual monitor operations and nested access paths can also be controlled via grant and capability assignment. Finally, capability operations can be used to control the timing required for many dynamic access policies.

In order to validate access policies, in Section 5 we defined the access safety problem and showed how to determine the potential access of each subject. While many need to know and access restriction policies can be validated by examining the potential access graph, in order to verify access policies for dynamic objects it is usually necessary to examine and verify the code manipulating capabilities. The potential access graph pinpoints the problem; it remains to develop techniques for program verification which include the ability to treat protection problems.

Acknowledgements

The work reported here has profited greatly from numerous

discussions we have had with our colleagues David Gries, Carl Hauser, and Richard Reitman. Each of them also carefully reviewed an earlier draft of this paper.

Bibliography

1. Andrews, G.R. and J.R. McGraw. Language features for process interaction. Proc. of ACM Conference on Language Design for Reliable Software, Sigplan Notices 12, 3 (March 1977), 114-127.

2. Brinch, Hansen, P. The programming language Concurrent Pascal. IEEE Trans. on Software Engineering SE-1, 2 (June 1975), 199-207.

3. Harrison, M.A., Ruzzo, W.L., and J.D. Ullman. Protection in operating systems. Comm. ACM 19, 8 (August 1976), 461-471.

4. Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. ACM 17, 10 (October 1974), 549-557.

5. Jones, A.K., and B.H. Liskov. A language extension for controlling access to shared data. IEEE Trans. on Software Engineering SE-2, 4 (December 1976), 277-285.

6. Silberschatz, A., Kieburtz, R.B.,and A. Bernstein. Extending Concurrent Pascal to allow dynamic resource management. IEEE Trans. on Software Engineering SE-3, 3 (May 1977), 210-217.

7. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

8. Wirth, N. Modula: A language for modular multiprogramming. Software-Practice and Experience 7, 1 (January 1977), 3-35.

9. Wulf, W. et. al. Hydra: the kernel of a multiprocessor operating system. Comm. ACM 17, 6 (June 1974), 337-345.