International Journal of Computer and Information Sciences, Vol. 7, No. 2, 1978

An Operation-Control Scheme for Authorization in Computer Systems[1]

Naftaly Minsky[2]

Received October 1976

The access-control authorization scheme, which is being used for the protection of operating systems, is found to be inadequate in other areas, such as in databases and information systems. A new authorization scheme, which is a natural extension of access control, is proposed. The new scheme, which is called "operation control," is shown to be superior to the access-control scheme in a number of ways. In particular, it facilitates more natural and efficient representations of policies, particularly the type of complex policies that appear in information systems, it facilitates enforcement by compile-time validation due to a greater stability of authority states, and it reduces the need for revocation.

KEY WORDS: Protection; access control; operation control; authorization; operating systems; information systems.

1. INTRODUCTION

Authorization in computer systems is a discipline under which an action on the system can be carried out by a user or by one of the modules of the system only if the actor is authorized to perform this action. Such a discipline is necessary for the protection of the security and integrity of systems.

Most current protection techniques are based on the so-called access-control approach to authorization. This approach has been developed by Lampson,(7) Graham and Denning,(3) Wulf and Jones,(18) and others, mostly in the context of operating systems, and it enjoys a considerable degree of success in this area. Unfortunately, however, this success has not been

[1] This work was partially supported by Grant DAHCIS-73-G6 of the Advanced Research Project Agency of the US government. This paper is a modified version of the paper "An Activator-based protection scheme," July 1976 (SOSAP-TR-25).

[2] Rutgers University, Department of Computer Science, New Brunswick, New Jersey.

0091-7036/78/0600-0157$05.00/0

matched in other areas, such as databases and information systems. It is our contention that this failure is a result of some fundamental limitations of access control as a scheme for representation of authority structures. These limitations are discussed in Sec. 2. A generalization of the access-control authorization scheme is suggested in Sec. 3, and its merits are discussed in Sec. 4.

2. THE ACCESS-CONTROL (AC) APPROACH TO AUTHORIZATION

The access-control approach to protection and authorization is well documented in the literature. In particular, the reader is referred to the excellent review articles by Saltzer and Schroeder[16] and by Linden.[9] Here we outline only the essential features of this approach and discuss some of its limitations.

The system to be protected is formally viewed as a fourtuple (B, O, J, U), where B is a set of *objects*; O is a set of *operators*; J is a set of *subjects*, which are the *actors* that actually apply operators to objects, and are thus responsible for the dynamic behavior of the system; and U is the *authority state* of the system. The authority state is formally defined as a set $\{(S, b, o)\}$, where a triple (S, b, o) is the permission for subject S, which belongs to J, to apply operator o to object b. In other words, (S, b, o) is a permission for S to have *access o to object b*.[3] Of course, the system must be supported by an *enforcement mechanism* that guarantees that the only operations which are carried out are those permitted by the authority state U.

There are a number of ways to represent the set of permissions $\{(S, b, o)\}$. A method particularly relevant to this paper is called the *capability-based protection.*[8,18] Under this version of the AC-scheme the authority state of a system is represented by a distribution of special *control objects* which we call *tickets*. A ticket is a pair $(b; r)$, where b is an identifier of an object, and r is a subset of a finite set of symbols, called *rights* (or access rights), which identify, in some way, the operators that can be applied to object b. That is to say, the subject S that possesses a ticket $(b; r)$ is allowed to apply to b the operators identified by r. The generation and transport of tickets are tightly controlled so that the mere fact that a subject S has a ticket $(b; r)$ is taken as uncontestable proof that S is authorized to have the specified access rights to object b.

Thus, under the AC-scheme (or, rather, under the capability-based[3]

[3] The phrase *capability based* used for this version of the access-control protection is appropriate, even though we are using the term "ticket" for what is usually called *capability*, because the set of tickets owned by a given subject determines its capabilities with respect to the system. (In this paper the term capability is used in its colloquial sense).

version of it) the tickets are used as the *elementary building blocks of authority structures*, a kind of *elementary particle* of authority. Unfortunately, for a number of reasons, tickets are not suitable to serve as the *only* elementary building blocks of authority.

First, every ticket represents privileges with respect to a *specific object*, the one addressed by it. These privileges are independent of the value (or state) of the object. The problem is that authority structures are frequently based on the value of the objects involved, and are independent of their identity. To demonstrate the difficulty here, consider the following example. When a higway patrolman is sent to his duty, he has to be given the authority to cite traffic violators. This authority is not given to him in the form of tickets, one for each violator. Indeed, the patrolman's authority cannot be defined in this form because at the time that the patrolman is sent to his duty, the traffic violators do not exist, and the identity of the future violators is not known, so that it is impossible to construct individual tickets for the violators at that time. The point is that the patrolman's authority has to do with the behavior of motorists, not their identity. Tickets are too specific for this purpose, and at the same time they are not sensitive enough, being independent of the properties (values) of the objects addressed by them.

Another problem with tickets results from the *unit of activity* that they are designed to authorize. Every ticket represents a permission to apply certain operators to the object addressed by it. However, the activity of a subject may not be expressible purely in terms of operations on individual objects. One may have to use *interactions* between objects, where by "interaction" we mean an operation that involves several objects, and that cannot be decomposed into a sequence of legal unary operations on the individual objects. The problem is that privileges with respect to an interaction cannot be expressed purely by means of tickets, which represent permissions to perform operations on individual objects. (This difficulty is demonstrated by an example in Sec. 4.2.2).

Our conclusion from these observations is that there is a need for a new type of control object which can directly authorize interactions between objects, and which would not be based exclusively on the identity of the objects involved in operations. Such a control object, which we call an *activator*, is the basis for the protection scheme proposed in this paper.

3. THE OPERATION-CONTROL (OC) SCHEME

The protection scheme to be introduced in this section is *capability based* in the sense that it associates with every subject a set of control objects

that determine its capabilities.[4] However, our scheme differs from the access control scheme by the nature of these control objects. In addition to the tickets, which, as under the AC-scheme, represent privileges with respect to objects, we have control objects called *activators*, which represent *privileges with respect to operators*, in the following sense: Every activator A identifies an n-ary operator o (for $n \geqslant 0$), specifying the conditions under which o can be invoked by the subject that possesses A. There may be several such activators for the same operator o, which may impose *different* preconditions on the activation of o, representing different privileges with respect to o. We are using the name *operation control* (OC) for this scheme because the activators possessed by a subject determine directly the type of operations it can perform, quite independently of the access rights that it has to various objects.

3.1. Terminology and Conventions

To set the stage for our discussion we now introduce our interpretation for some well-known terms such as *object*, *subject*, *domain*, and *type*.

3.1.1. Objects and Their Types

We base our approach toward types on the type concept used in Hydra,[5] which can be described briefly as follows:

The set of all objects in a system is described in terms of the three-level tree shown in Fig. 1. The root of this tree is a primitive and unique object called *template*. The objects at the second level are *instances* of this template, and are called *template objects*, or *type objects*. Each of these type objects, such as the object t, serves as a template for a set of objects in the third level, which are said to be *objects of type t*, or *t-objects*. A template t is supposed to contain the structural definition of all its instances.

In order to impose some *behavioral discipline* on objects, we introduce the concept of *protected type*. This is a type t for which there is a fixed set of operators (procedures) that have the exclusive ability to manipulate and observe t-objects directly. We will say that such an operator is *privileged with respect to t*. The set of all these operators for a given t is denoted by *privileged*(t). Thus, a t-object for a protected type t can be manipulated either directly, from within one of the privileged(t) operators, or indirectly by invoking these operators. An important special case is an operator that is privileged with respect to only one type t: such an operator is called a

[4] Note, again, that the term capability is being used here in its colloquial sense.

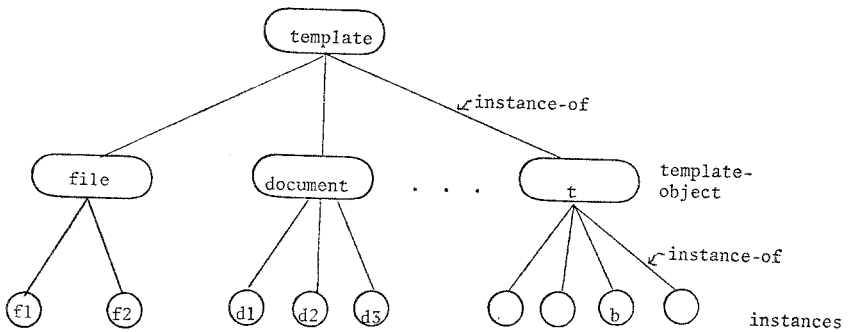

Fig. 1. The Hydra approach to types. The object b is an instance of the object t (or a t-object). The object t in turn, is an instance of the distinguished object *template*, and thus it is a template object (or type object).

t-operator. (Note that the existence of a fixed set of t-operators is the basis for the notion of "abstract data type" as it is defined in CLU[(10)5]).

3.1.2. Shareable Objects and Their Tickets

We distinguish between two broad classes of objects, to be called *shareable objects* and *concrete objects*. A *concrete object* is one that is physically contained in one's *domain*.[6] For example, the integer 7 and the symbol "seven" are concrete objects. A *shareable object*, on the other hand, is not contained in any private domain, but it can be shared, or accessed, by several subjects. Shareable objects are accessed by means of a concrete object called a *ticket* (which is essentially identical to the ticket of the access-control scheme).

With every type t of shareable objects we associate a (possibly empty) set of symbols:

$$rights(t) = \{r1,..., rn\}$$

Each of these symbols, ri, is called a "right" with respect to type t.

A *ticket* c for an object b of type t is defined to be a concrete object which is denoted by

$$c = (b: t; r)$$

[5] It should be pointed out that our scheme is not based on the concept of privileged operators. It is the other way around: we see later that privileged operators can be implemented under our scheme. This concept is mentioned at this point to accommodate some of the examples given in the following sections.

[6] The concept of domain is defined later; for the moment it is enough to see it as the work space of some subject.

where r, or $r(c)$, is a subset of *rights*(t). If a given right *ri* is in $r(c)$, we say that "the ticket c has the right *ri*". Since the content of a ticket depends on its t component, we use the phrase *t-ticket* to identify all tickets that address t-objects. A t-ticket that has all its possible rights is denoted by (b: t; ALL). The symbols *ri* have been called *rights* in anticipation of their role in our protection scheme, which is discussed later. (Note: The "t" part of c signifies that b is the identifier of an object of type t. Whenever the type of b can be understood from the context, we use the simplified notation (b; r) for a ticket.)

In general, there may be several tickets pointing to the same object b. We say that such tickets are *related.* Let $c1$, $c2$ be two related tickets. We say that $c1$ is *weaker*, or *less permissive*, than $c2$ if

$$r(c1) \subseteq r(c2)$$

Also, we say that $c1$ is *strictly weaker* than $c2$ if

$$r(c1) \subset r(c2)$$

As to the *generation and manipulation of tickets*, the following is assumed. Tickets cannot be changed, and they can be generated only in the following two ways:

1. When a new shareable object b of type t is created, a ticket for it is also created, with all of *rights*(t) in it. This ticket, (b: t; ALL), is called the *primary ticket* of object b.

2. Given a ticket c it may be possible to generate a new ticket c', which *cannot be stronger than* c. Such c', which addresses the same object as c, is called a *derivative* of c. (Later we see when it is actually possible to generate derivatives of a given ticket).

3.1.3. The Facade of Objects

One of the objectives of our scheme is to enable value-dependent authorization. However, one cannot always expect to be able to determine the legality of an operation by using the value of *all* the attributes of an object, because:

1. It may happen that an attribute of an object is *not directly observable.* As an example consider the hidden components of abstract data types.

2. It may happen, in certain types of objects, that the mere act of observation of an attribute of an object introduces a change in the object itself. An example of this "quantum mechanical effect" is a record stored on a tape, which cannot be observed without repositioning the tape.

3. One may not *want* to allow the use of certain attributes for protection, because such a use may itself reveal confidential information about the object.

For these, and other reasons which include efficiency, we now introduce the concept of the *facade* of an object, which is *the part of an object that is usable for protection purposes.* More formally, with every type t we associate the set

$$\text{facade}(t)$$

which is the set of attributes of t-objects which are usable for authorization purposes. By convention, the facade of primitive scalar objects, such as *real* and *integer* numbers, is their value.

3.1.4. Subjects and Their Domains

A computing system changes in time in response to *instructions* submitted to a processor for execution,[7] where an *instruction* is a request to apply a specific operator to a specific sequence of operands. We distinguish between two types of *instruction sources*:

1. An *external source*, such as a human user sitting at the terminal.
2. An *internal source*, which is a *procedure* maintained as an object in the system.

One important difference between these two types of instruction sources is that an external source is totally unpredictable as far as the system is concerned, while the behavior of a procedure can be at least partially predicted ahead of time.

We define a *subject* to be a pair

$$(\text{INS}, D)$$

where INS is an instruction source, and D is the collection of objects that are directly addressable by INS. D is called the *domain* of S. We later see that the domain of a subject determines its capabilities and also serves as its workspace.

Corresponding to the two types of instruction sources we distinguish between two types of subjects. A subject (INS, D) whose INS is an external source is called a *user*, and a subject whose INS is an internal source is called an *operator*.

[7] For simplicity we assume that there is just one processor in the system.

3.1.5. Operators

Operators are the dynamic components of a system. Every sequential process interacting with the system can be described as a sequence of *operations*, each of which is the application of an operator to a sequence of zero or more operands. An operator may have various side effects on the system, but only one value that is called the *outcome* of the operator. The outcome is a concrete object that may, in particular, be a ticket of a shareable object. This outcome is stored in the domain of the subject that invokes the operator.

We distinguish between two types of operators. First, there is a fixed set of *primitive operators* whose internal activity would not be subject to the control of our protection mechanism. For example, the set of machine instructions may be considered the primitive operators of an operating system. Secondly, an operator may be a subject (INS, *D*) whose source of instructions INS is a procedure maintained by the system. Note the recursive nature of the operator concept: A procedure, which is the INS component of some operator, has been defined to be a source of instructions, while an instruction is a request to invoke an operator.

3.1.6. Authorization Scheme and Policies

Following Jones and Wulf,[18] we distinguish between the concept of *authorization scheme* and that of *policy*. A *policy* is a specific discipline that one would like to impose on a system. It is occasionally called *authority structure*. An authorization (or protection) *scheme* is a framework that should be general enough to accommodate a variety of policies as efficiently and conveniently as possible. Such a scheme consists of two main components: a *language*, which can be used for the specification of policies, and an enforcement mechanism, which guarantees that no illegal operations are carried out.

3.2. Activators and the Enforcement Mechanism

An *activator* is a concrete object, which we denote by

$$A = \langle o, p1,..., pk \mid G \rangle \rightarrow po$$

Here A is the name of the activator; o is an *operator identifier*; pi, for $i = 1,..., k$ is a *condition on the ith operand of o*, to be called *operand pattern*; G is a condition defined on all operands, and possibly on other objects in the system (it is called the *global condition* of A); and po is a condition on the outcome (result) of the operator, to be called the *outcome*

pattern.[8] (Whenever we do not wish to distingush between operand patterns and outcome patterns, we use the term *activation pattern* or just *pattern.*)

The existence of an o-activator A in the domain of a subject S represents the authority for S to apply the operator o to any objects $q1,..., qk$ in the domain of S, such that for every $i = 1,..., k$ the operand qi matches the operand pattern pi of A (satisfies the condition pi), and that the global condition G of A is satisfied. The activator A also gives S the authority to introduce into its own domain the outcome of the operator o, thus invoked, provided that this outcome "matches" the outcome pattern po of A (that is, satisfies the condition po). To support this interpretation of the activators the following enforcement mechanism is proposed.

Let us define an *instruction* to be the construct

$$A(q1,..., qk)$$

where A is an o-activator for some operator o, and qi are its operands. It is assumed that a subject can form such an instruction only from concrete objects $A, q1,..., qk$ which exist in its own domain. Thus, the set of instructions which are expressible by a subject is directly determined by the content of his domain. Moreover, such an instruction is carried out only if the operands *match* the activation patterns as described above, and if G is satisfied. It is the responsibility of the enforcement mechanism to perform this *pattern matching* and to guarantee that no illegal operations are carried out. Once an operation is carried out, its outcome, if any, is checked. If it matches the pattern po, it is added to the operating domain; otherwise, the value of the operation is lost, and an error procedure may be invoked.

Note that an operand qi may be of two types: it may be a concrete object, such as an integer number, which stands for itself; or it may be a ticket that addresses a shareable object, which is the real operand. Even in the latter case we usually refer to qi as an operand, relying on the context to determine whether qi itself, or the object addressed by it, is meant.

Thus, it is clear that the content of the domain D of a subject S, at a given moment, determines the set of operations that can be carried out by S at this moment. We can say, therefore, that *the domain of a subject determines its capabilities,* or its *authority.*

There is an instructive analogy between the role of the activators in our scheme and the role of *enzymes* as the control devices of the living cell. The function of every enzyme is to facilitate a certain chemical reaction. Such a reaction takes place if there are enough substrates in the cell which

[8] If the operator o does not have an outcome, then the part "$\rightarrow po$" will not appear in our notation. Also, the condition G may be absent. Thus, an activator may be denoted simply by $A = \langle o, p1, ..., pk \rangle$.

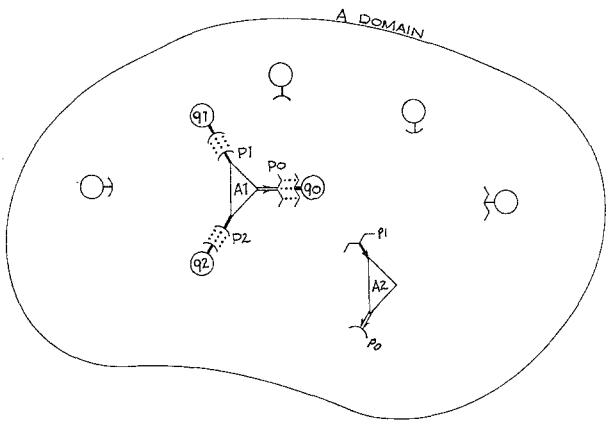

Fig. 2. Objects are represented by circles, while the pattern attached to a circle represents the type, facade, etc. of the object. Activators are represented by triangles with patterns (for the activation patterns) attached to them. The analogy between the activity of a domain and the chemistry of a living cell is quite attractive, although it is not complete. One can think of the activators as the enzymes in a cell, and of the objects as the various substrates. An operation in a domain is analogous to a chemical reaction in the cell, which requires an appropriate enzyme and substrates which fit it. In the figure, activator $A1$ is depicted in the process of being attached to operands $q1$, $q2$, generating $q0$ as an outcome.

fit the *activation sites* on the enzume, in some analogy to the function of our activator (see Fig. 2). Although this analogy between activators and enzymes should not be carried too far, it does provide an interesting viewpoint of the proposed scheme.

Note the similarity between the activators and the *formal parameters specification* (or FPS) of procedures in programming languages: Both determine the legal set of operands of an operator. There is, however, an important difference between these two. Our activator is an independent object, disconnected from the operator that it activates. Moreover, while there is just one FPS per operator, we see below that there may be several different o-activators for the same operator o, *which have different strength.* The concept of *strength of activators* is defined as follows.

Let A be an *activator* of order k (with k operand patterns). We define *range*(A) to be the set of all possible $(k + 1)$-tuples ($q1$,..., qk, qo) of objects, which can be matched with the corresponding activation patterns of A, and which satisfy the condition of G of A.

Let A and A' be two o-activators for a given operator o. We say that A' is *weaker* than A (or, equivalently, A is *stronger* than A') iff

$$\text{range}(A') \subseteq \text{range}(A)$$

We say that A' is *strictly weaker* than A iff

$$\text{range}(A') \subset \text{range}(A)$$

Such an A' is also called a *reduction* of A.

As to the *generation and manipulation of activators*, the following is assumed: First, there is no way to change an existing activator except to erase it. Secondly, new activators can be generated only in the following two ways:

1. When a new operator o is created, an o-activator is generated with it. It is called the *primary o-activator.*

2. Given an o-activator A, it may be possible to generate a new activator A', which is called a *derivative* of A. A' cannot be stronger than A. (Later we see when it is actually possible to generate such a derivative.)

The following properties of the activators follow immediately from the above:

1. The set of all o-activators, for a given operator o, is *partially ordered* with respect to the relation *stronger.*

2. Every activator is stronger than all its derivatives.

3. The primary o-activator is *the strongest o-activator.*

3.3. The Activation Patterns

To be more concrete about the activators and their use we have to suggest a specific structure for the activation patterns. The structure described in this section is designed to support many of the known authority structures in computer systems. Note that the run time overhead attributable to the enforcement mechanism that is necessary to support our scheme depends on the complexity of the activation patterns and that of G. In this paper we do not impose any restriction on this complexity, because such restrictions should depend on the nature of the system to be protected.

3.3.1. Operand Patterns

An operand pattern P, denoted by

$$[I; R; V]$$

is a conjunction of three predicates I, R, V, which are called components, or subpatterns, of P. They are defined as follows.

The subpattern I (which is the only mandatory part of P) is called the *identity-based subpattern.* It is either a type identifier, t, which is meant to

be satisfied by any object of type t, or it is the phrase b: t which is satisfied only by the particular object b of type t. The entire pattern P whose I component identifies a type t is called a t-pattern. The structure of the two other components of a t-pattern depends on t. If a subpattern R or V does not appear in P, it is interpreted as identically TRUE, which means that it does not impose any restrictions on the object matched to the pattern.

The subpattern R, called the *privilege-based* subpattern, is applicable only when t is a shared type. R has the general form

$$R = r1 \,\&\, r2 \,\&\, \cdots \,\&\, rk$$

where each ri is a symbol that belongs to $rights(t)$. R is meant to be satisfied by any ticket of a t-object that contains *at least* the rights $r1,..., rk$.

The subpattern V, called the *value-based* subpattern, is a predicate defined on the *facade* of the object being matched with it.

Example. Let *doc* be a type of shareable object that carries documents in a military information system. Let the *facade* of doc objects be defined by:

$$\text{facade(doc)} = \{\text{slevel: integer, category: text}\}$$

where *slevel* is an integer that specifies the *security level* of the document, and *category* specifies its category, such as *navy* or *army*. These two attributes are the traditional security parameters in military establishments.[17] Let

$$\text{rights(doc)} = \{U, E\}$$

As we see below, the symbols U and E stand for the rights to update and erase a document, respectively.

Suppose now that there are three doc-operators: *read*, *update*, and *erase*, which are the only operators able to manipulate a document directly (see Sec. 3.1.1). The primary activators of these operators are as follows:

$$\text{READ} = \langle \text{read, [doc]} \rangle$$
$$\text{UPDATE} = \langle \text{update, [doc; } U\text{], [text]} \rangle$$
$$\text{ERASE} = \langle \text{erase, [doc; } E\text{]} \rangle$$

The activator READ can be applied to any doc-ticket displaying the content of the document. The activator UPDATE can be applied to a doc-ticket that contains the U right. The second operand of *UPDATE*, which can be any text object, specifies the nature of the desired update. The activator ERASE can be applied to any doc-ticket that contains the E right, erasing the content of the document.

The right U can properly be considered an *update-right* because U is required by the primary update activator, which means that it would be required by *all* update activators. Thus, the update operator can never be applied to a doc-ticket that does not have the U right. A similar argument would show that E is the *erase-right*.

As has already been explained, the primary o-activator, for any given operator o, allows for the most general use of o. In order to provide for a more limited use of o, one creates *weaker* derivatives of the o-activator. For example, the activator

$$\text{ERASE}' = \langle \text{erase}, [\text{doc}; E, U] \rangle$$

is weaker than ERASE because it can be applied only to a doc-ticket that has both U and E rights in it. The activator

$$\text{ERASE}'' = \langle \text{erase}, [d\text{: doc}; E] \rangle$$

is also weaker than ERASE, because it can erase only a specific document d. Note, however, that there is no ordering relation between ERASE′ and ERASE″. Neither can be a derivative of the other.

To illustrate the use of value-based subpatterns consider a subject S whose domain D contains the following activators:

$$\text{READ}' = \langle \text{read}, [\text{doc};; s\text{level} \leqslant 2] \rangle$$
$$\text{UPDATE}' = \langle \text{update}, [\text{doc}; U; s\text{level} \leqslant 2 \ \& \ \text{category} = \text{“navy”}], [\text{text}] \rangle$$

which are reduced derivatives of READ and UPDATE, respectively. S has the power to read any document whose security level is smaller than or equal to 2 and whose ticket it can get. S can also update *navy* documents with $s$level $\leqslant 2$, provided that it has a ticket with the U right for such a document. However, S *cannot* erase any document because it does not have any erase activators.

3.3.2. *The Outcome Pattern*

The outcome pattern *po* of an activator

$$A = \langle o, \ldots \rangle \rightarrow po$$

is a condition on the outcome of the operator o, when invoked by means of A. This means that only an outcome that satisfies *po* can be added to the operating domain by using A. The structure of the outcome patterns is identical to that of the operand patterns. However, the interpretation of the *R-component* of the pattern is different. Let *po* be the pattern

[I; $r1$ & $r2$ & ⋯ & rn; V]. The rights symbol $r1$,..., rn in this pattern are not treated as conditions on the rights $r(c)$ contained in the ticket c returned as the outcome of the operation. Rather, they serve as a *filter* on $r(c)$, in the following sense: Any right in $r(c)$ that is not represented in $r1$,..., rn would be erased from the outcoming ticket c. Thus, *the component R of po serves as the upper limit for the rights that might be returned as a result of applying the activator.* This means, for example, that the activator

$$\langle o, p1\rangle \rightarrow [I1; r1; V1].$$

is weaker than

$$\langle o, p1\rangle \rightarrow [I1; r1, r2; V1]$$

Returning to our document example, consider an operator *getdoc* that retrieves documents from files. Let the primary activator of *getdoc* be

$$\text{GET} = \langle \text{getdoc}, [\text{file}], [\text{text}]\rangle \rightarrow [\text{doc}; \text{ALL}]$$

The first operand of *getdoc* must be a file in which *getdoc* is supposed to locate a document identified by the second parameter, returning the ticket for the document as its outcome. Note that the outcome pattern [doc; ALL] matches any document ticket. Consider now the following, weaker derivative of GET:

$$\text{GET}' = \langle \text{getdoc}, [f1\text{: file}], [\text{text}]\rangle \rightarrow [\text{doc}; U; \textit{s}\text{level} = 1]$$

GET′ can get documents only from a specific file $f1$; moreover it can produce only tickets for documents whose security level is equal to 1, and these tickets can have no more than the U right in them.

3.4. Control over the Generation of Objects

As we saw in Sec. 3.3, one can control the use of individual *existing* shareable objects by the distribution of their tickets. We now show how the *generation* of new objects can be controlled. (Only the essentials of such control are discussed, leaving some details unspecified.) This serves as a further illustration of activators and their patterns.

First, we assume that there is a primitive operator *gen-type* that is able to generate new type objects (the objects in the second level of the tree in Fig. 1). Let the primary activator of this operator be

$$\text{GEN-TYPE} = \langle \text{gen-type},\ldots\rangle \rightarrow [\text{template}; \text{ALL}]$$

This activator has a sequence of operand patterns, not specified here, that determine the types of arguments required by gen-type. Invocation of GEN-TYPE would generate a template object returning a ticket for it with all its possible rights. Obviously, only a subject who has the GEN-TYPE activator, or some derivative of it, can generate new types.

We now assume that, together with a new type object t, the following "instantiation activator" is generated:

$$\langle \text{gen-}t, p1,..., pk\rangle \rightarrow [t; \text{All}]$$

where *gen-t* is an operator that generates instances of type t, and $p1,..., pk$ determine the arguments required by *gen-t*. Application of this activator to an appropriate sequence of operands returns a ticket to the newly formed t-object, with all the rights(t) in it. For example, the primitive instantiation activator for the type *doc* may be

$$\text{GEN-DOC} = \langle \text{gen-doc, content:[text], slevel: [integer], category: [text]}\rangle \rightarrow [\text{doc; ALL}]$$

(To distinguish between the various operand-patterns we use labels such as "content: [...]".) The three operands of *gen-doc* determine the initial state of the generated document: its *content*, *security level*, and *category*. A subject having this activator can generate documents with arbitrary security level and category, obtaining a ticket for the generated object with all its possible rights. However, a subject having the following derivative of GEN-DOC:

$$\text{GEN-DOC}' = \langle \text{gen-doc,..., category: [text;; value = "navy"]}\rangle \rightarrow [\text{doc; } E]$$

can generate only documents whose category is navy, getting for them tickets without the U right. Note that documents generated by GEN-DOC′ can never be changed, because there can be no tickets with the U right for them.

3.5. The Two Types of Control Objects: Their Role and Behavior

Our protection scheme is based on two primitive types of objects, *activators* and *tickets*, which we call, collectively, *control objects*. The distribution of these control objects throughout the system serves to determine its authority state; namely, such distribution determines who can do what in the system. The roles of the two types of control objects are reviewed in this section, and their transport is discussed.

There is a *symmetry* in the functions of activators and tickets in our scheme. A *ticket* for object b residing in the domain D of a subject S represents the privileges that S has for b, in the sense that the ticket determines the set

of operators that may be applied by S to b. Analogously, an o-activator that resides in D represents the privileges that S has for the operator o, in the sense that it defines the set of objects to which o can be applied by S. Tickets and activators play *complementary roles* in our scheme: neither one of them alone is sufficient for the application of an operator to a shareable object. For this, one needs both an activator and a ticket (or several tickets) that fit the activator.

The complementarity of activators and tickets allows us to *formalize the semantics of the rights symbols.* Let the symbol $r1$ belong to *rights*(t) for a given type t. We define the *privileges associated with* $r1$ to be the set of t-patterns of the various activators in the system that require $r1$. Moreover, for a given domain D we define the *local privileges associated with* $r1$ to be the set of t-patterns *in* D which require $r1$. Note, for example, that this set may be empty, rendering $r1$ useless in the context of D, even if the set of global privileges of $r1$ is nonempty. For instance, the right U of Sec. 3.1 would be useless within a domain that has no update activators.

The two types of control objects exhibit some similar structural and behavioral characteristics, which are best seen by comparing the following two sets: the set $T(b)$ of all tickets for a given object b, and the set $A(o)$ of all o-activators. $T(b)$ and $A(o)$ are partially ordered sets with respect to the relation *stronger*, defined for tickets and activators, respectively. Every ticket in $T(b)$ is a direct, or indirect, derivative of the primary ticket of b, which is created together with the object b itself. Likewise, every activator in $A(o)$ is a dirivative of the primary o-activator, which is created together with the operator o. A control object, whether it is an activator or a ticket, is stronger than all its derivatives.

Control objects are to be distributed by means of two primitive *transport-operators*: *k-copy* and *k-move*. (*k* stands for *kernel*, as these operators should belong to the kernel of the system, which is discussed in Sec. 3.8). Each of these operators, when applied to a control object *co*, generates a new control object *co'* in some other place in the system; such a *co'* cannot be stronger than *co*. The difference between the two transport operators is that *k*-copy does not affect the original control object, whereas *k*-move erases it. Thus, *k*-move, in effect, *moves* a control object from one place to another, possibly reducing it in the process.

To get a degree of control over the transportability of individual control objects, the following facility is introduced. The *facade* of a control object of either type consists of two boolean components, "copy" and "move," to be called the *intrinsic rights* of the object. The operator *k*-copy can be applied to a control object *co* only if "it has the copy right," namely, if the "copy" component of co is TRUE. Likewise, *k*-move can be applied only to a control object that has the "move" right. Thus, a control object with neither intrinsic

rights is *untransportable*. Of course, the control object *co'* generated from *co* by one of these operators cannot have more intrinsic rights than *co*, but it can have less.

It is obviously vital to have some control over the use of the transport operators. Such a control can be achieved by the distribution of their activators. These activators are discussed in Sec. 3.8.

3.6. The Structure of Domains, and Their Dynamic Behavior

As has already been explained, the content of a domain D at a given moment in time T determines the set of operations that can be carried out at time T by the subject S associated with D. But what can we say about the *future* capabilities of the subject S? To answer this question one must be able to predict the future content of D. This in turn requires an understanding of the dynamic behavior of domains.

3.6.1. External and Internal Changes of Domains

We distinguish between two types of domain change, to be called *external changes* and *internal changes*. An *external change* of a domain D associated with subject S is a change caused by an operation invoked by *another* subject S'. In particular, it is such a subject S' that created D in the first place. To predict the dynamic behavior of a given domain D under external changes, one must be able to tell which subjects have the capability of changing D, and what they are up to.

An *internal change* of a domain D is one that is caused by an operation invoked by *its own subject* S, as follows: Let A be the following activator

$$\langle ... \rangle \rightarrow po$$

in D, and let $A(q1,..., qk)$ be an operation invoked by S. The outcome of this operation, if any, is added to D. *This outcome is a concrete object that satisfies po.* (Note that, depending on *po*, the outcome that is added to the domain may be a primitive object such as integer, a ticket for a shareable object, or even an activator.) Thus, the nature of the possible internal changes of a domain D is determined explicitly by the content of D itself.

Now, it seems reasonable to assume that in a well-designed system there would usually be only a small number of subjects S' that are able to change an existing domain D. Moreover, even these subjects are not likely to exercise their ability to change D very often, so that an external change of a domain is

likely to be a relatively rare event. Therefore, we continue our discussion of the dynamic behavior of domains, taking only internal domain changes into account.

3.6.2. *The Structure of Domains*

Until now, domains have been presented as monolithic structures. We now distinguish between two parts of a domain, to be called *permanent* and *transient* parts. The *permanent part* of the domain of a subject S is created together with the subject, and is attached to it throughout its lifetime. We sometimes refer to this part simply as *the domain* of the subject. The *transient part* of the domain, to be also referred to as the *workspace* of the subject, exists only for the duration of a single *activity period* of the subject. In the case of a user an activity period is a single *session* of user-system interaction. An empty work space is attached to the domain of the user at the beginning of a session, only to disappear when the session terminates. An activity period of an operator is the period between its invocation and its return. When an operator is invoked, a new work space, which contains all the operands, is attached to its domain, to be deleted when the operator returns control.

We now introduce the convention that, unless specified otherwise, an internal change of a domain effects its transient part only. This means that the outcome of an operation is usually stored in the work space of the subject, leaving the permanent part of the domain invariant of the activity of its own subject. An example can clarify all that.

Example. Consider a subject S whose domain, or rather, the permanent part of whose domain, is given in Fig. 3. This domain contains two file tickets, for files $f1$ and $f2$ which are assumed to contain documents. The domain also contains five activators: the activators GET′ and GET″ for operator getdoc, which are reductions of the activator GET (cf. Sec. 3.3.1). Operator getdoc gets a docticket identified by its second operand, from the file identified by its first operand. GET′ can be applied only to a ticket of one file, $f1$. Namely, GET′ can be used to generate tickets with the U right, for documents stored in $f1$. We denote the set of all such tickets by $F1$. The activator GET″, which can be applied to the ticket $c2$ of the file $f2$, can generate a set $F2$ of tickets of documents stored in $f2$ whose *s*level $= 1$. These tickets would have only the E right in them.

The activators READ′, UPDATE′, and ERASE, already introduced in Sec. 3.3, can be applied to the doc-tickets generated by GET′ and GET″. READ′ can be applied to any document whose *s*level<2. Note, therefore, that some of the documents whose tickets may be generated by GET′ cannot be read by S. UPDATE′ can be used to update any navy document, provided

```
    c1 = (f1:file)
    c2 = (f2:file)
 GET'  = <getdoc,[f1],[text]> → [doc;U]
 GET"  = <getdoc,[f2],[text]> → [doc;E;slevel=1]
 READ' = <read,[doc;;slevel<2]>
UPDATE'= <update,[doc;U;category="navy'],[text]>
 ERASE = <erase,[doc;E]>
```

Fig. 3. The permanent part of a domain. It contains two file tickets and five activators. GET′ and GET″ can operate on the file tickets, generating doc-tickets into the transient part of the domain. The last three activators operate on these doc-tickets.

that its ticket has the *U* right. This means that none of the documents on $f2$ can be updated by *S*, and only some of the documents on $f1$, the navy documents, can be updated. Finally, using ERASE, *S* can erase all the documents obtained from $f2$, but none from $f1$. (Note that our subject cannot generate new documents, because he does not have a gen-doc activator.)

Note that all the doc-tickets generated by our subject would be inserted into its workspace, which is the transient part of the domain that disappears at the end of the session. That is to say, the subject *S* cannot have any doc-tickets for extended periods of time. This property of our scheme is very important, as it reduces the need for revocation.

A Comment. The domain in Fig. 3 is incomplete in the sense that it contains no activators for basic operations such as integer addition or manipulations of text variables. Such activators are necessary because, by our definitions, *no* operation can be carried out by a subject without having the proper activator in his domain. However, following Minsky,[11] we assume that all control objects that are necessary to authorize the use of operators and objects we do not wish to restrict are included, by default, in all domains.

3.7. The Global Condition of Activators

We define the condition *G* of activators by the following two properties:

1. *G* is a conjunction of predicates: $g1 \,\&\, g2 \,\&\, \cdots \,\&\, gk$
2. The reduction of a global condition can be performed by adding a conjunct to it, not by a change of existing predicates.

At this point no restrictions are imposed on the individual predicates *gi*. In particular, *gi* can be defined on all the operands of the activators, as well

as on other objects in the system that are not otherwise involved in the operation. Moreover, we will allow *gi* to have side effects. Here are some applications of the global condition.

3.7.1. Correlation Between Operands

The operand pattern *pi* has been defined to be a condition on the *i*th operand. Since *G* can be defined on all operands, it can correlate them. For example, let *copyd* be an operator that copies the content of one document (doc-object) into another. Consider the following copy*d* activator:

$$\langle \text{copy}d,\ d1\text{: [doc]},\ d2\text{: [doc; } U\text{]} \mid d1.\text{category} = d2.\text{category} \\ \&\ d1.s\text{level} \leqslant d2.s\text{level} \rangle$$

The only restriction imposed by this activator on the individual operands is that $d2$ must have the U right (without which the update of $d2$ would not be possible). However, the G part of this activator requires that the two operands be of the same category, and that the second operand should not have a lower security level then the first. (This is a very common type of restriction in military information systems.)

3.7.2. Conditions on Global Status Variables

Suppose, for example that there is a global variable T in the system which represents the *real time*. An activator

$$\langle \cdots \mid t1 \leqslant T \leqslant t2 \rangle$$

can be used only in the time period $(t1, t2)$ because its G part would return FALSE at any other time. In a similar way one can construct activators that are conditioned on other global variables in the system.

3.7.3. Self-destructive Activators

Consider a predicate *countdown* of the form

```
BEGIN  OWN N;
       N ← N − 1;
       IF N ⩽ 0 RETURN FALSE;
END
```

If this predicate is used as a component of the G part of an activator A, it

limits the number of times that A can be used. If, for example, the own variable N of such a countdown predicate is initialized to 2 when the activator is created, then after two applications of A it will return false, preventing further use of A.

3.7.4. Revocation of Activators[9]

One of the classical problems in capability-based protection is how to revoke a privilege already granted. Revocation of tickets has been studied extensively by Redell,[15] Cohen,[1] and others. Here we see how activators can be revoked.

Consider a subject $S1$ having an activator

$$A1 = \langle \cdots \mid g \rangle$$

in its domain $D1$. Let $a1$ be a boolean variable local to $D1$. Suppose that $S1$ generates a derivative $A2$ of $A1$:

$$A2 = \langle \cdots \mid g \,\&\, a1 \rangle$$

storing it in the domain $D2$ of subject $S2$ (see Fig. 4). It is quite obvious that $A2$ can be used only as long as the boolean variable $a1$ is TRUE. Thus, although $A2$ belongs physically to $S2$, it is still controlled by $S1$, which can prevent the activation of $A2$ simply by turning off the variable $a1$. Moreover, every derivative of $A2$ would be controlled by $S1$ in the same way because it is impossible to remove a conjunct from G. Furthermore, one can add additional controls in a similar way. For example, let $a2$ be a boolean variable in $D2$; suppose that $S2$ generates a derivative

$$A3 = \langle \cdots \mid g \,\&\, a1 \,\&\, a2 \rangle$$

of $A2$, storing it in the domain $D3$ of $S3$ (see Fig. 4). Now, $S1$ can deactivate and reactivate both $A2$ and $A3$ by turning $a1$ off and on, while $S2$ can control $A3$ in a similar way by means of its own variable $a2$.

Note that in a similar way one can construct a variety of revocation patterns. In particular, the variables $a1$ and $a2$ above may be stored inside some shared object whose tickets are distributed in a certain way.

3.7.5. Monitoring the Use of Activators

Using the ability of G to produce side effects, one can monitor the use of activators, as follows. Suppose that a subject $S1$ has the activator

$$A1 = \langle \cdots \mid g \rangle$$

[9] I am indebted to Dorothy Denning for suggesting this important application of the global condition of activators.

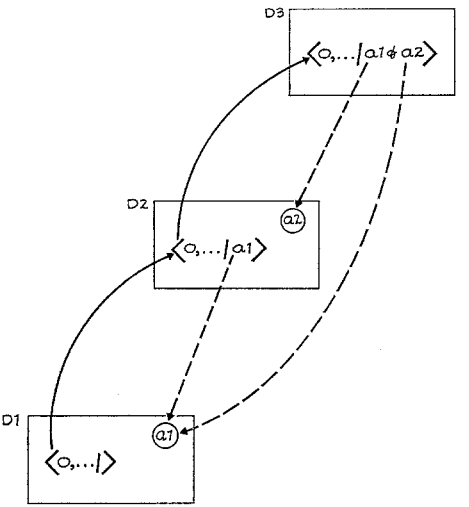

Fig. 4. Revocation of activators. The solid arrows represent the sequence of derivation of activators. The dashed arrows represent the dependency of the activator on the boolean variables $a1$, $a2$.

in its domain $D1$. Let

$$A2 = \langle \cdots \mid g \,\&\, m \rangle$$

be a derivative of $A1$ where m is a predicate that always returns TRUE, and is programmed to write a record into a file accessible to $S1$, reporting about each invocation in which it participates. Thus, $S1$ would have an audit trail of all activations of $A2$ and of any derivative of it, because m cannot be removed from an activator. The users of $A2$, or of its derivatives, may not be aware of such audit trail being formed, and they certainly cannot do anything about it, because no part of G can be removed from an activator.

3.8. The Kernel of the Protection Mechanism[10]

The purpose of this section is to clarify and support some of the assumptions made in preceding sections. In particular, it has been assumed that for every type t there is a set of *privileged operators* with respect to t, which have the exclusive ability to modify and observe t-objects directly. Here we show how this exclusiveness can be imposed by means of the basic

[10] This section can be skipped on first reading.

protection mechanism. This discussion brings us to the foundations of the protection mechanism, which is frequently called its *kernel.* However, only some aspects of such a kernel are discussed here. Its complete study is beyond the scope of this paper, because the kernel is likely to be strongly dependent on its context. For example, the kernel would surely be very different in the case of operating systems, databases, and programming languages. Therefore, the following discussion should not be viewed as a proposal for a specific implementation.

3.8.1. Segments and Their Operators

In an attempt to find a uniform implementation for all types of shareable objects, we first define a primitive type called *segment*, which is to serve as a host for shareable objects of all types.

A *segment* is essentially a chunk of storage divided into a sequence of *slots*, each of which can host one concrete object of a given primitive type. For example, there may be integer slots, text slots, ticket slots, and activator slots. The various slots in a segment are addressed by their relative position with respect to its origin. This division of a segment into slots is called the *structure* of the segment. A segment is allocated, to host an object of a given type t, by the *gen-t* operator (defined in Sec. 3.4). The structure of this segment is determined by t, and is fixed for the lifetime of the object.

We treat the set of all segments as a type. It is a special kind of type, as it *includes*[11] all other types of shareable objects, since, by our definition, an object of any type is also a segment. Like any other type of shareable object, the type "segment" has its own rights, which, following Hydra, are called *kernel rights.* We assume that

$$\text{rights(segments)} = \{k\text{-read}, k\text{-write}\}$$

A ticket $(b: t; r)$ of a shareable object b can be viewed also as a ticket for the segment hosting an object b, provided that we generalize the r-component of a t-ticket as follows:

$$r \subset \text{rights}(t) \cup \text{rights(segment)}.$$

That is to say, the kernel rights are common to all types, and may appear in any ticket.

[11] The concept of *type inclusion* could have been introduced formally in Sect. 3.1.1. We avoided this for the sake of simplicity, and we are using type inclusion, in an ad-hoc manner, only in this case.

We now introduce two operators that operate on segments: the already mentioned transport operators, *k-copy* and *k-move*. The primary activator of *k*-copy is

$$K\text{-COPY} = \langle k\text{-copy}, s1\text{: [segment; } k\text{-read], loc1: [integer],}$$
$$s2\text{: [segment; } k\text{-write], loc2: [integer], reduction: [text]}\rangle$$

The operator *k*-copy copies a concrete object from slot *loc*1 of segment *s*1 into slot *loc*2 of *s*2. The fifth parameter, *reduction*, specifies the reduction to be applied to the copied object, if it happens to be a ticket or an activator. Namely, it specifies in what way the new object is weaker than the original. (The syntax of the reduction specification is not discussed here.) The conditions for the copy operation to be carried out are of two types: explicit and implicit. The conditions explicit in the activator are that the ticket for *s*1 must have the "*k*-read" right, and the ticket for *s*2 must have the "*k*-write" right. In addition to these explicit conditions we assume that the following restrictions are built into the operator itself.

1. The concrete object in slot *loc*1 of *s*1 must fit the type of slot *loc*2 of *s*2. This simply means that the *k-copy* operator does not violate the type specification of the structure of segments. Thus, only a ticket can be stored in a ticket slot, only an activator can be stored in an activator slot, etc. Moreover, a ticket slot may be earmarked for ticket of a certain type of objects only, and there may be a limit imposed on the rights that may be contained in the slot. All such specifications must be honored by *k*-copy.

2. If the content of the loc1-slot of *s*1 is a ticket or an activator, it must have the intrinsic right "copy". That is, it must be a copyable control object.

The second transport operator, *k-move*, has a similar activator, *K*-MOVE. The only differences between these two operators are:

1. *k*-move erases the content of slot *loc*1 of *s*1.

2. In the equivalent of restriction 2 above, the intrinsic right "move" (rather than "copy") is required.

Note that, because of restriction 1, a slot behaves like a variable in a typed language; namely, it has a specified range of objects that may be stored in it. A particularly important case is that of a *ticket variable*, which can contain tickets only for a given type of objects, and with a *specified maximal set of rights*. Such a facility, which is similar to the ticket variables introduced by Jones and Liskov,[6] can contribute greatly to proving correctness of policies.

As we will see next, the activators *K*-COPY and *K*-MOVE are too powerful to be contained in any domain. We have to use restricted derivatives of them.

3.8.2. The Implementation of Privileged Operators

For a given type t consider the following derivative of K-COPY:

$$\text{READ-}t = \langle k\text{-copy}, s1\text{: } [t; k\text{—read}], \text{loc1: [integer]}, \\ s2\text{: [myself]}, \text{loc2: [integer]}, \text{reduction: [text]}\rangle$$

This is a reduction of K-COPY in two ways. First, the $s1$ pattern can be matched only to tickets of t-objects (which is a smaller set than all segments, which are allowed by the pattern $s1$ in k-copy). Secondly, the phrase *myself*, in the $s2$ pattern, is a context-dependent pattern that matches only the *operating-domain.* This allows a subject having this activator in its domain, together with a t-ticket:

$$c = (b\text{: } t; k\text{-read},...),$$

to copy information from object b addressed by c, into its own domain. Or, in other words, *READ-t allows reading of t-objects.* In a similar way the activator

$$\text{WRITE-}t = \langle k\text{-copy}, s1\text{: [myself]},..., s2\text{: } [t; k\text{-write}],...\rangle$$

allows one to copy information from the operating domain into t-objects, that is, to *write into t-objects.* One can also construct a pair of similar derivatives of K-MOVE, which move data into and from t-objects.

We thus have four activators for a given type t which can be used for reading from and writing into t-objects. Assuming that the primary activators of k-copy and k-move are not themselves available, only a subject that has in its domain one of these four activators is able to manipulate or observe t-objects directly. Thus, *an operator is privileged with respect to t* if and only if it has at least one of the above four activators in its own domain. *Note that an operator can be privileged with respect to a number of different types.* This is not possible under the implementation of "abstract-types" in CLU, for example.

4. DISCUSSION

The purpose of this section is to evaluate the proposed operation-control (OC) scheme along a number of dimensions, such as conceptual adequacy and simplicitly, efficiency, and expressive power. To keep the discussion in perspective, we compare our scheme with the capability-based access-control (AC) scheme, specifically the Hydra version of it.[1,5,18] Such comparison is appropriate because, as we see next, the OC scheme can be viewed as a natural extension of the capability-based version of the AC scheme.

4.1. Conceptual Simplicity

In comparing our scheme to the access-control one it may seem that we are sacrificing a great deal of conceptual parsimony by adding another type of control object. This, however, is not the case. We show next that the proposed scheme can be viewed as a *natural extension of the AC-scheme*, which results from the removal of an unwarranted restriction in it.

Note that the enforcement mechanism that is necessary to support the OC-scheme is essentially identical to that of the AC-scheme. Under the AC-scheme an operation $o(q1,..., qn)$ is considered legal if the tickets of the operands $q1,..., qn$ satisfy the requirements imposed by the *formal-parameter specification* (FPS) part of the operator o (this part is called *template* in Hydra). Thus, the FPS of an operator is functionally equivalent to our *primary activator*. Moreover, having the right to call an operator o under the AC-scheme is equivalent to having the primary o-activator under the OC-scheme. Thus, the OC-scheme can be viewed essentially as an AC-scheme with the *additional degree of freedom* which allows the formation of a whole set of o-activators of different strength. These activators represent varying degrees of authority with respect to the operator o, just as the set of tickets of a given object b represents varying degrees of authority with respect to b. This *symmetry* in the treatment of objects and operators, which does not exist under the AC-scheme, is important because it reflects a common feature of authority structures. The capabilities of an actor (subject) in computer systems, as well as in the real world, frequently results from the type of operations that it can perform, not only to the privileges he has with respect to specific objects. Thus, our scheme is conceptually cleaner and more complete than the AC-scheme.

4.2. Expressive Power

We say that a policy is *expressible* in a given scheme if it can be specified and enforced by means of the formal devices provided by the scheme. (Note that expressibility, so defined, is a stronger concept than implementability. Indeed, any policy can be implemented on any kind of system, simply by programming it into an interpreter that carries out every operation on the system.) The difference in expressive power between the AC- and the OC-schemes is primarily along two dimensions: *value dependency* and the *ability to handle interactions*. It is not that value-dependent policies and policies with respect to interactions cannot be expressed in the AC-scheme (although this is true for some such policies). The main problem is that the implementation of such policies tends to be cumbersome and inefficient to the point of being impractical.

4.2.1. Value-Dependent Policies

We define a *value-dependent policy* to be one under which the legality of an operation depends on the value (or state) of the operands. In particular, we are interested in the case in which the value dependency itself depends on the subject that invokes the operation. Such policies can be represented in our scheme by the distribution of activators with appropriate value-based patterns. On the other hand, the only way to represent such a policy under the AC-scheme is by a suitable *value-dependent distribution of tickets*. That is to say, the *placement* of tickets depends on the values of the objects addressed by them. As we demonstrate later by an example, such a representation may be so costly and arror prone that it becoms completely impractical.

4.2.2. Handling of Interactions

The concept of *interaction*, mentioned in Sec. 2, is defined more rigorously as follows:

> For a given subject S, an *interaction* is an n-ary operator, for $n > 1$, which cannot be expressed by S as a sequence of legal unary operations on its individual operands.

Note that by this definition an operator that is an interaction for one subject may not be an interaction for another. Consider, for example, a procedure *appoint*(e, j) which appoints employee e to job j in a corporate information system. Suppose that this appointment is actually done by planting a pointer to j in the record which represents e, and vice-versa. Suppose also that such tinkering with pointers is not allowed outside of the procedure *appoint*, for obvious reasons. Thus, for any subject other than the procedure *appoint* itself, the operator *appoint*(e, j) is an *interaction*, because there is no way to decompose it into a sequence of (more primitive) operations on e and j separately.

Now, consider the following policy with respect to the interaction *appoint*. Let $E1$ and $E2$ be two sets of employees and let $J1, J2$ be sets of jobs. Let S be a subject that is to be allowed to appoint employees in $E1$ to jobs in $J1$ and those in $E2$ to jobs in $J2$, but is not allowed to appoint employees in $E1$ to jobs in $J2$, or those in $E2$ to jobs in $J1$.

Under our OC-scheme, let $pe1, pe2, pj1, pj2$ be patterns that match members of the sets $E1, E2, J1, J2$, respectively. The desired policy is realized by giving S the activators ⟨appoint, $pe1$, $pj1$⟩, and ⟨appoint, $pe2$, $pj2$⟩.

To see the difficulty under the AC-scheme we now consider several attempts to represent this policy. First, suppose that the operator *appoint* requires the right $r1$ from its first argument and $r2$ from the second. Now,

S can be given a ticket with $r1$ for every member of $E1$, $E2$; and a ticket with $r2$ for every member of $J1$ and $J2$. The problem is that this would allow S to make cross-appointments, of members of $E1$ to jobs in $J2$, etc.

The desired policy can be *implemented* in a system such as Hydra as follows: one may maintain a table inside the operator appoint, which identifies the set of triples (S, e, j) such that S is allowed to appoint e to job j. The operator *appoint* may be programmed to obey such specifications. However, this is not a *representation* in terms of the AC-scheme, and it is quite contrary to its underlying philosophy of the capability-based approach to authorization.

A correct but unnatural and very inefficient representation of our policy in terms of the AC-scheme is the following: For every "appointable" pair (e, j) we create an object ej that represents the pair. The operator *appoint* can be considered as a unary operator on such pair objects. The authority of our subject can be defined by giving him a ticket for every pair-object ej that S is allowed to appoint. Such representation of authority, apart of being highly artificial and inconvenient, may be extremely inefficient. For example, if the cardinality of each of the sets $E1$, $E2$, $J1$, $J2$ is N, then S will have to maintain $2N**2$ tickets, as opposed to *two* activators, which are necessary under our scheme. Moreover, if the membership of an employee e in one of the sets $E1$, $E2$, is determined by the value of e, then the maintenance of the authority structure is very difficult. Whenever e stops to be a member of $E1$, say, one has to revoke all existing tickets for appointable pairs ej.

4.3. Efficiency

The efficiency of a protection scheme should be measured along two dimensions: *efficiency of the representation* of policies, and the *efficiency of* their *enforcement*. To explain what we mean by efficiency of representation, we now introduce a number of concepts:

We use the term *control material* for the *overall distribution of control objects throughout a system*. For a given policy P, we are interested in the following properties of the control material which is necessary for the representation of P.

1. The *volume* of the control material, which is the number of control objects in it.

2. The *complexity of the distribution* of the control material. We say that the distribution of the control material is more complex if there are more rigorous requirements as to the *placement* of the various control objects.

3. The *volatility* of the control material. By the term *volatility* we

mean, loosely speaking, the amount of change in the control material necessary to maintain a given policy during normal operation of the system, or to support incremental changes in the policy itself. (The term *stability* is also used, as the opposite of volatility.)

These aspects of a protection scheme and their relevance to the efficiency of the representation of policies are discussed in the succeeding two sections. Efficiency of enforcement is discussed in Sec. 4.3.3.

4.3.1. The Volume and the Complexity of the Control Material

In this section we argue that the volume of the control material necessary for the implementation of a given policy under the OC-scheme tends to be smaller and less complexly distributed than that under the AC-scheme. An important reason for this is that activators can have *various degrees of generality* (*or, specificity*), which can be adapted to the nature of the policy at hand. For example, the activator ⟨read, [doc]⟩ can be used to read any document, while the activator ⟨read, [*d*: doc]⟩ can be used only to read the specific document *d*. Tickets, on the other hand, have fixed degree of specificity: every ticket represents privileges with respect to one specific object. Thus, one may need many tickets to represent a capability which, under the OC-scheme, can be represented by a single activator. We now demonstrate this tendency by an example, which is a generalization of an example used by Jones and Wulf.[5]

Example: Let *memo* be a type of object that carries memoranda. Suppose that in addition to the text itself, which can be retrieved by the operator *read*, every memo object has a set of boolean attributes

$$X = \{x1,..., xn\}$$

associated with it. We say that a memo *m* *satisfies a certain attribute xi* if $xi(m) = \text{TRUE}$.[12] Suppose also that for every subject *S* there is a set $Y(S) = \{y1,..., yk\} \subset X$ of boolean attributes, representing his privileges with respect to memoranda, as follows: *S should be allowed to read all memos, and only such, which satisfy all yi in Y(S).*

An *OC-representation* of this policy is the following. Let

$$\text{facade(memo)} = \{x1,..., xn\}$$

and let the primary read activator be

$$\text{READ} = \langle \text{read, [memo]} \rangle$$

[12] A possible interpretation of this is the following: There are *n* nondisjoint categories of memo objects. An object *n* belongs to the *i*th category if $xi = \text{TRUE}$.

Suppose that a subject S is given only the following reduction of READ:

$$\text{READ1} = \langle \text{read}, [\text{memo};; y1 \,\&,\ldots, \& \, yk] \rangle$$

Suppose also that the set of tickets $\{(m\text{: memo})\}$, one for each memo object in the system, is stored on a file *dir* from which all subjects can copy tickets. It is obvious that the desired policy is satisfied under these conditions.

The salient feature of this implementation is that the various subjects have effectively different "power" with respect to memo objects, due to the different read activators in their domains. That is why they can safely share the same set of tickets, contained in file *dir*, and still have different privileges. As we see next, the situation is quite different in the AC-case.

Under the AC-scheme we assume that the operator *read* demands that the right "read" is in the operand ticket. Since all subjects involved must have the right to invoke *read*, the difference between the subjects can only be in terms of the memo tickets, each with the "read" right, which are available to them. Thus, the desired policy can be established as follows:

For a given memo object m, let

$$Z(m) = \{z1,\ldots, zj\} \subset X$$

be the set of boolean attributes satisfied by m. Let *target*(m) be the set of subjects S such that for each of them

$$Y(S) \supset Z(m)$$

$Y(S)$ is exactly the set of subjects that, by our policy, should be allowed to read m. Therefore, the ticket (m: memo; read) should be available to these subjects and to none other. To establish such a distribution of tickets, we suppose that every subject S in our system has a file, memo*s*(S), that is readable only by it. Whenever a new memo object m is created, a *noncopyable* ticket (m; memo; read) should be stored in memo*s*(S) for every S in target(m), and nowhere else. This is essentially the solution given by Jones and Wolf to a similar problem.[5]

Let us now compare the control material that is necessary for these two implementations of our policy:

As to the *volume of the control material*, suppose that there are NS subjects in a system, and M memo objects. Let K be the average number of subjects that are allowed to read a memo object. The AC-implementation requires $K*M$ memo tickets to be stored in the system, while under the OC-representation only M tickets are required. (Also, the AC-implementation needs NS tickets for the operator *read*, which is comparable to the NS-activators needed under our scheme.)

Even more important than the volume of the control material is the *complexity of its distribution.* The AC-implementation requires a very specific distribution of the memo tickets among the *NS* files memos(*S*). This distribution of tickets is itself a formidable task. Moreover, every file memos(*S*) must be well protected and readable by the specified subject *S* only.

The situation under the OC-implementation is much simpler. Once the *NS* different read activators are correctly distributed among the various domains, we can store all the *M* tickets in one file, which is readable by everybody and does not have to be especially protected. This is obviously much less complex than the case under the AC-implementation.

4.3.2. Stability of the Control Material

Stability is a very desirable property of the control material for a number of reasons:

1. The maintenance of highly volatile control material may take much effort, particularly when it is involved with revocation.

2. The probability for making mistakes in the distribution of control objects increases with their volatility.

3. Stability of the control material facilitates compile-time checking. We argue that the control material tends to be more stable (less volatile) under our OC-scheme, mainly because of the following reasons:

1. Volatility clearly increases with the complexity and the volume of the control material, which tends to be higher under the AC-scheme.

2. There is a difference between the *lifetime* of tickets and activators. A ticket $(b; r)$ cannot exist prior to the creation of object b, or after its destruction. An activator $\langle o, [t] \rangle$, on the other hand, can live as long as the operator o and the type t exist; the meaning of this activator does not depend on the existence of any particular t-objects. Thus, an implementation of a policy under an AC-scheme, which is based entirely on tickets, has an *a priori* temporary nature, and is therefore likely to be relatively volatile.

Moreover, the variations in control material that do exist under the OC-scheme tend to concentrate in the transient part of domains, while their permanent part, which contains mostly activators, is relatively stable. This is important because it is mostly the permanent part of the domain of a subject that determines his authority (cf. Sec. 3.6). We now illustrate some of these observations in the context of our memo example. We start by examining *volatility under a fixed policy*, the one presented in Sec. 4.3.1, when the population of memoranda is changing.

Under our OC-scheme each subject S has an activator that determines the type of memo object that can be read by it. This gives S a general authority with respect to memoranda, which is independent of the particular memo objects in the system. Indeed, no change is necessary in the domain of S when new memo objects are generated or when existing ones are deleted or changed. The authority makeup of S is stable under such changes. On the other hand, under the AC-implementation of this system, whenever a new memo object is created, its tickets must be given to all subjects allowed to read it. Even worse, when some of the attributes xi of m are changed, the tickets of m may have to be *revoked* from those subjects that are not to be allowed to read it any longer. Thus the control material must be constantly changed to maintain the given policy.

Next, let us consider *volatility under an incremental policy change.* Using again the memo system, suppose that in addition to *read* there is another operator, *update*, whose primary activator under the OC-mechanism is

$$\langle \text{update, [memo], [text]} \rangle$$

Suppose, also, that the set of memoranda that a subject is allowed to read may be different from the set he is allowed to update. Imitially, S may have the following two activators:

$$\langle \text{read, [memo;; } x1\text{]} \rangle$$
$$\langle \text{update, [memo;; } x1\text{], [text]} \rangle$$

This means that S can read and update memos that satisfies $x1$.

Now, suppose that we wish to change this policy allowing S to update only memos satisfying both $x1$ and $x2$. All we have to do is to replace his update activator with

$$\langle \text{update, [memo;; } x1 \,\&\, x2\text{], [text]} \rangle$$

For the AC case suppose that all the subjects have tickets that allow them to call both *read* and *update*, and that the operator *update* requires the right "update" from its argument just as read requires the right "read." Initially S must have access to the set of tickets

$$\{(m; \text{read, update}) \mid \text{where } m\text{-satisfies-}x1\}$$

for all memos satisfying $x1$. To change the privileges of S as above, *we must replace all its memo tickets with the two sets*

$$C1 = \{(m; \text{update}) \mid \text{where } m\text{-satisfies-}x1\text{-and-}x2\}$$
$$C2 = \{(m; \text{read}) \mid \text{where } m\text{-satisfies-}x1\}$$

Thus, the same policy change that requires the replacement of a single activator under an OC-scheme requires the replacement of many tickets under the AC-scheme.

4.3.3. *Efficiency of Enforcement*

Two factors affect the enforcement efficiency:

1. The complexity of the computation necessary for the validation of a given operation.

2. The degree to which validation can be performed at compile time. We already saw that the enforcement mechanism that is necessary to support the OC-scheme is essentially identical to that of the AC-scheme. In both cases the operands must be checked against the operand patterns of the given activator, which is the "template" in the case of Hydra. Of course, the complexity of such parameter checking depends on the complexity of the activation patterns. Our scheme *allows* for essentially arbitrarily complex patterns, but it does not *require* such complexity. If the OC-scheme were used for the protection of operating systems, one would probably impose severe restrictions on the syntax and semantics of activation patterns and of G. Much more general patterns can be used in the context of information systems, without a significant relative increase in the overhead due to protection.

As to the second factor which affects efficiency of enforcement, we claim that our scheme facilitates compile-time validation, due to the greater stability of its control material. In particular, it appears that the compiler can do much of the necessary checking by analyzing the relatively static "permanent-part" of the domain of a subject. However, more study is necessary to substantiate this claim.

5. CONCLUSION

The operation-control (OC) scheme introduced in this paper is a natural generalization of the capability-based version of the access-control (AC) scheme developed for operating systems. This generalization is achieved by the introduction of the *activators*, which play an analogous role to that of the tickets under the AC-scheme, and which do not require any new enforcement effort. The use of activators together with tickets has a profound effect on the authorization scheme: The representation of complex policies becomes easier and more natural. The control material necessary for the representation of policies tends to be less voluminous, less complex, and more

stable. The stability of the control material reduces the need for revocation and facilitates compile-time enforcement. It is also believed that this stability and simplicity of the control material would facilitate the proof of policies. It should be pointed out, however, that the proposed scheme has yet to prove itself in the context of a real system. There is work in progress that attempts to base the protection of information systems on this scheme.

ACKNOWLEDGMENTS

I wish to thank Joseph Stein from the Hebrew University of Jerusalem, David Levine and Matthew Morgenstern from Rutgers, Dorothy Denning from Purdue, and an anonymous reviewer for reviewing this paper, and for very useful comments.

REFERENCES

1. E. Cohen and D. Jefferson, "Protection in the Hydra Operating System," in *Proceedings of the Fifth Symposium on Operational System Principles* (November, 1975).
2. O. J. Dahl and C. A. R. Hoare, "Hierarchical Program Structures," in *Structured Programming*, Dahl, Dijkstra, and Hoare, eds. (New York, Academic Press, 1972).
3. G. S. Graham and P. J. Denning, "Protection-Principle and Practice," in *Proceedings of the 1972 SJCC* (AFIPS Press, 1972).
4. M. H. Harrison, *et al.*, "On Protection in Operating Systems," *Proceedings of the Fifth Annual SIGOPS Conference (1975)*.
5. A. K. Jones and W. A. Wulf, "Towards the design of secure systems," *Software Pract. Exper.* 321–336 (1975).
6. A. K. Jones and B. Liskov, "An Access Control Facility for Programming Languages," Technical Report, Carnegie Mellon University (1976).
7. B. Lampson, "Protection," in *Proceedings of the Fifth Princeton Symposium on Information Science and Systems* (March 1971), pp. 437–443.
8. B. Lampson and S. Sturgis, "Reflections on an operating-system design," *Commun. ACM* (May, 1976).
9. T. A. Linden, "Operating system structures to support security and reliable software," *Surv. ACM*, to be published.
10. B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Not.* (April, 1974).
11. N. Minsky, "Intentional resolution of privacy protection in database systems," *Commun. ACM* (March, 1976).
12. N. Minsky, "An Activator-Based Protection Scheme," Rutgers Technical Report (July, 1976).
13. M. Rosenblit and N. Minsky, "On the Decidability Problem of the Safety of Protection Systems," Rutgers University Technical Report (February, 1976).
14. G. Popek and C. S. Kline, "Verifiable Secure Operating System Software" [AFIPS (1974 NCC), 145–151], *Proceedings of the 1974 NCC* (AFIPS Press, 1972), pp. 145–151.
15. D. Redell, "Naming and Protection in Extendible Operating Systems," Ph.D. thesis, University of California, Berkeley (1974).

16. J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE* **63**(9) (September, 1975).
17. C. Weissman, "Security controls in the ADEPT-50 time-sharing system," in *1969 AFIPS Conference Proceedings*, Vol. 35, pp. 119–133.
18. W. Wulf, "HYDRA: The kernel of a multiprocessor operating system," *Commun. ACM* **17**:337–345 (June, 1974).
19. W. A. Wulf, "ALPHARD: Towards a Language to Support Structured Programs," CMU Technical Report (April, 1974).