M.I.T. Laboratory for Computer Science

Computer Systems Research Division

July 22, 1978

Request for Comments No.165

AESOP: An Architecture for an Object Based Machine

By Allen W. Luniewski

In recent years there has been considerable interest in programming languages that support both procedural and data abstractions[3,8]. The use of data and procedural abstractions makes program development and verification simpler since only the semantics, and not the implementation details, of the abstractions need be known. Currently these languages tend to be implemented on traditional Von Neumann computers. This document presents AESOP, an Advanced Experimental Small Object Processor, as one possible machine architecture for supporting the use of data and procedural abstractions directly.

1. Introduction

AESOP is a high level description at the instruction set processor (ISP) level of a machine architecture that supports procedural and data abstractions. The description is of the interface of the machine; details that only relate to performance or methods of implementation have been excluded.

The choice of a high level description is a very fundamental one. It enables AESOP to be used as a vehicle to explore various issues involved in the implementation of an object based machine. In particular, the issues of efficient storage of many small objects (the "small object" problem) and the problem of permanent storage and cataloging of these same objects can be explored. AESOP provides a language in which such issues can be discussed.

An underlying design goal is that the machine be able to support languages such as CLU and ALPHARD, that provide objects, in an efficient and "reasonable" manner - that is, the architecture should provide a natural base on which to implement such languages. To this end the machine must support the notion of typed objects - all objects have a type. However the machine itself does not do complete type checking, except for the hardware implemented types. Instead the machine must provide the means for the programs running on it to perform any desired type checking. The decision not to have the machine perform complete type checking is based upon two concerns. First, including these checks in the basic hardware may slow the whole machine down to an unacceptable extent. This is especially true if AESOP is to be implemented by emulation. Second, for the most part a compiler for a high level language can do most such checks at compile time eliminating the need for most runtime checks. CLU, for instance, has this property. As a result of these considerations the architecture provides complete type checking of hardware types and the mechanism to ensure that extended type objects are only manipulated by their type managers. It fails to do complete type checking in that it does not provide an explicit mechanism to enforce the correct type of arguments passed to procedures or a mechanism to enforce the type of variables.

One of the more fundamental issues to be addressed in an object oriented machine is "What is an object?" and the related question "What does a variable contain?". Although it might be nice to avoid these issues by putting them off as implementation issues, addressing them has very fundamental effects upon the way that programs see the world and the ways in which the architecture can develop. AESOP takes the view that variables contain the names of objects and not the objects themselves. Objects contain the names of other objects and nothing more. In particular, objects do not physically contain other objects. The important implications of this approach are that an object may be contained in more than one object (e.g. shared between objects) and shared between processes on the same machine (at the least).[1]

The notion of type is also essential to understanding AESOP. Every object has a type, thus "type" is an attribute of all objects. The type of an object is the name of the type manager that implements operations upon that object. A type manager must, at a minimum,[2] enforce the specifications on objects that have that type. To look at it slightly differently, a type manager provides a collection of objects that have it as their type and a collection of operations on those objects.

2. The Naming Architecture

One of the fundamental architectural decisions that must be made in designing a machine is how objects are named by other objects, including how executing procedures make references. AESOP takes the approach of allowing every non-code object (e.g. not procedures) to directly refer to other objects by containing the name of the object to be used while code objects refer to other objects indirectly through name spaces (or naming contexts[6]).

[1] Note that no position is taken on the important issue of whether objects can be shared across machine boundaries as in a network environment. Allowing such sharing may be an important architectural issue in designing a distributed system, but tackling this problem in the context of designing an object based machine that is intended to illustrate the hard problems in the local case seems excessive. Thus the position in this paper is that if such sharing across machine boundaries is allowed then it does not have architectural effects at this level.

[2] Since a type manager may do more than just enforce type oriented specifications, the more limited terms of cluster as in CLU or form as in ALPHARD are avoided.

Thus there is an asymmetry between the ways in which procedural and non-procedural objects refer to objects: procedures only have context dependent names while non-procedural objects only have context independent names. The template LNS of a procedure, to be described later, provides a means for procedures to have what amounts to context independent names by inserting constant names into the template LNS. There is no mechanism for objects to easily simulate the effects of context dependent names. Although it is interesting to speculate on the need for context dependent data in non-procedural objects, AESOP does not provide such a facility since the need is not well documented in the literature.

Associated with every executing procedure are two name spaces - the local name space (LNS) and the global name space (GNS).[1] A procedure basically uses small integers as names; when it wants to refer to an object it uses an ordered pair of the form (name-space, short-name) where name-space specifies either the current LNS or the current GNS and short-name is a small integer. The effect is that a name is retrieved from the named name space (e.g. LNS or GNS) using short-name as an index into the name space. This name is then directly used to refer to the desired object: Thus procedures name objects indirectly through name spaces and not directly. Therefore by executing using a different LNS and/or GNS, the names used by a procedure change meaning; i.e. they are context dependent names with the pair (LNS, GNS) as the context.

The LNS is an important part of the system. The LNS (and the use of name spaces in general for indirection) allows procedures to be reentrant since the LNS will contain the names of objects that are local to the current invocation of the procedure, e.g. local variables. Parameters are handled by reserving entries in the LNS for each parameter that the procedure expects. Local variables are simply slots in the LNS that are reserved at compile time to contain the names of

[1] Name spaces are very similar to Hydra C-lists[9] and the indirectories of CAP[7].

objects local to each invocation of the procedure. The other items in the LNS will be objects that all invocations of the procedure need to know about such as external procedures and static/own variables.

The need for the GNS is not quite so clear cut. The intent is that the objects named in the GNS will be objects that need to be known to most procedures in the system. For instance, basic types (strings, reals and integers), operating system interfaces and language runtime support routines might appear in the GNS. These uses of the GNS represent an optimization - "factoring" common objects out of the LNS's of a collection of procedures. The alternative is to require that procedures refer to all other objects through their LNS. The existence of the GNS does have the property that by creating a name space containing the names of a new set of support routines and running procedures with that name space as a GNS it is possible to present a new virtual machine interface to those programs. This feature may be very useful for debugging programs and encapsulating programs.

3. Basic Types

This section will present a set of proposed basic types for AESOP. The set is not intended to be complete, but, rather, representative of the types that the basic machine should (and in some cases must) support. The types code segment, procedure, name space, RCG (to be presented later) and one basic data type (any will do) must be supported in the underlying implementation of AESOP; all others can, in principle, be implemented out of these basic mechanisms.

The basic types are probably supported entirely by the hardware although that is an implementation decision not addressed in this paper. Since these are defined as basic types it does not make much sense to try to fit their implementation into the AESOP formalism. Attempts to do so seem to lead to infinite recursion of implementations of one type depending on another (either A depends on B which depends on .... which depends on A ... or dependencies of the form A_1 depends on A_2 ... depends on A_N depends on ...).[1] The following discussion presents the basic types and suggests some operations on those types that AESOP should support.

Boolean is one basic type. There are two boolean objects - **true** and **false**. They are immutable objects and may only be named by other objects in the system; they may never be created or destroyed. They exist from the beginning of time until the end of time. There are operations to get the name of the object **true** and the name of the object **false**.[2] The sixteen binary operations on booleans are supported. In addition there are operations to branch depending on the value of some boolean variable.[3]

Another basic type is the integer type. The objects of this type are again immutable and may not be created or destroyed but only named. An implementation defined subset of the integers are supported. Operations that are supported include **plus minus, times, divide, mod, min, max** and **abs**. The integer type should also support comparison operations such as **less-than, equal** and **greater-than** that take two integers as operands and return a boolean as a result. Branching based upon the value of some integer variable may be supported although boolean based branching plus comparison operations on integers is sufficient.

[1] This is not to say that it all cannot be made consistent. Rather, such an attempt seems to serve no practical purpose.

[2] Alternatively, the names of the objects **true** and **false** could be kept in the GNS in conventionally known places

[3] The manner and form of branches are discussed later in this paper.

Another basic type is that of character (not character string). Once again characters are immutable and may only be named, not created or destroyed. The actual set of characters is implementation defined. One operation allowed on characters is the comparison operation which returns true if and only if its two operands are the same character. Other operations are allowed to convert characters to an integer representation (e.g. ASCII value) and back.

A mechanism is needed to aggregate a number of objects into a larger one. The mechanism to do this is the string. We consider strings to be a type for this discussion although they are more properly considered type generators.[1] A string is just a vector of names indexed by positive integers. Thus a character string might be formed by calling the string type manager to create a string of some length and then filling in the entries in the string with the names of characters. The same mechanism is used to construct integer vectors, name spaces, procedures and type managers. From these examples it should be clear that strings need not have elements of uniform type within them. This is not to say that strings of uniform type are not useful. In fact, strings of some or all of the basic types should be supported by the hardware for efficiency reasons (for instance character strings). The basic operations on strings are to create a string of some length, to lengthen (shorten) a string, to extract an element of a string, to insert an element into a string and to return the current length of a string. Some potentially useful operations are derived from the analogy of character strings: concatenate two strings, extract a substring of a string, replace a substring of a string and search for a substring in a string. It is also possible to construct record objects like those of CLU, Algol-W or Pascal using strings so that the record mechanism need not be explicitly included.

[1] String-of-integer is a type as is string-of-any but string is not a type. The discussion that follows, though, can be regarded as discussing the type string-of-mixed-types as a basic type.

A name space is represented as a **string-of-mixed-types**. A name space is more than that since it may be used as an LNS or GNS while an object of type **string-of-mixed-types** may not be used in this way.

Another basic type is the code segment. A code segment is the basic executable unit of AESOP. It contains instructions that the processor can execute. The only operations allowed on code segments are to create them (from some as of yet unspecified external form) and to execute them using some given (LNS, GNS) as the pair of name spaces (the naming context) for the executing code segment. Execution of a code segment always begins at the beginning of a code segment and continues until the end.

Procedures are represented by a string of two elements: the first is a template LNS and the second is a code segment (see figure 1).[1] When the procedure is called, the template LNS is copied

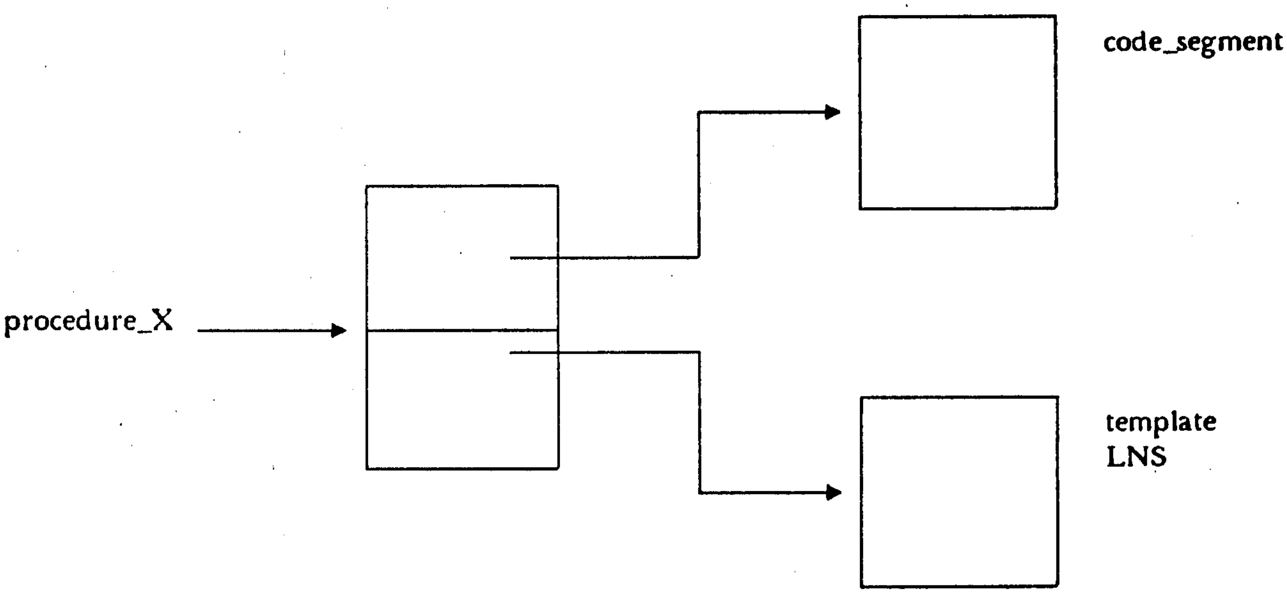


Figure 1. The format of the procedure object.

[1] Actually the string contains the names of these objects but the terminology here is used where confusion will not result.

into a newly created name space and the copy becomes the LNS for the procedure activation.[1] The template LNS will name all objects that the procedure needs to refer to during its execution (other than parameters and local variables). In particular it will contain the names of other procedures and type managers that the procedure may wish to call. The code segment in the procedure's representation is the code segment that will be initially executed when the procedure is called. The template LNS may name other code segments that will be used as part of the execution of the procedure.

Conceptually a type manager is a string of procedures, one for each operation that the type manager supports (see figure 2).[2] When an attempt is made to use an operation on some type, the string representing that type manager is examined and the name of the procedure that implements that operation is extracted. Then that procedure is called to perform the operation.

The representation chosen for procedures and type managers has a very interesting, and useful, property: their implementation may be changed (compatibally of course) without changing the names used by their callers. Moreover, the implementation can be changed without the danger of overwriting a piece of code that is being executed, or of destroying a template LNS that is being copied. In the case of procedures, the two pointers in the representation of the procedure only need to be changed atomically. For type managers only the name of the procedure being replaced need be changed in the operation list.

[1] The complete procedure call mechanism, including the passing of parameters, is described in a later section.

[2] Actually a type manager is two strings of procedures, one string for each of the two sets of operations that a type manager provides. The details of this are not important here and will be presented in a later section.

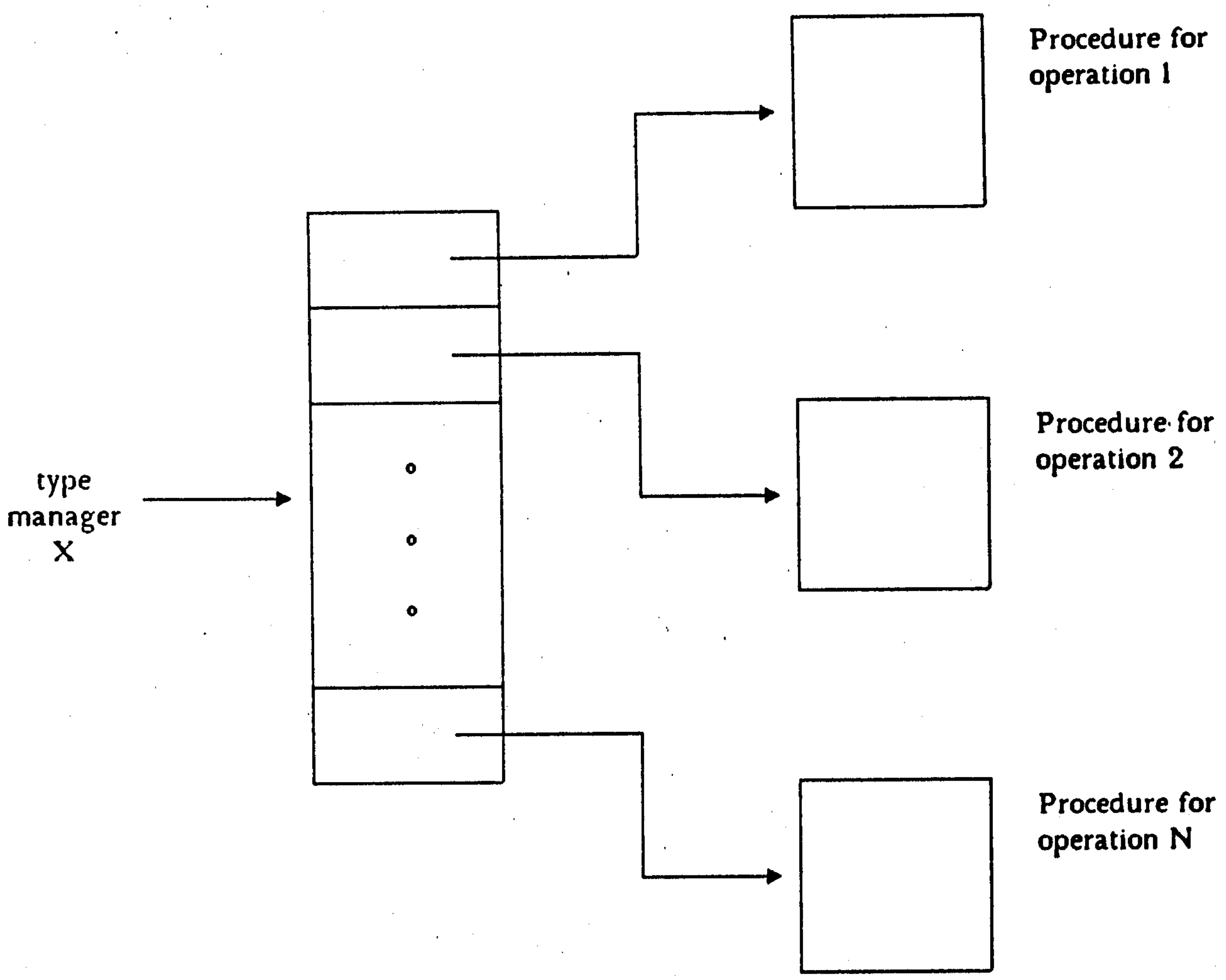

type manager X
o
o
o
Procedure for operation 1
Procedure for operation 2
Procedure for operation N

Figure 2. The format of a type manager.

4. Operation Name Spaces

Every type manager supports a collection of operations on objects of that type. Some operations, however, will have the same high level semantics in most, if not all, type managers. The existence of such operations is the motivation for operation name spaces.

The operations that a type manager supports are divided into two classes: the local operations and the global operations. The local operations are those operations on the type that are unique to that type; that is, most types do not support a similar operation. For instance in a stack abstraction there is a push operation to put an element on the stack although most abstractions do not have an operation that even vaguely resembles a push operation (consider for instance integers!). The global operations are those operations that are common to most types. Some examples of global operations are:

Get the type of an object.

Copy the name of some object.

Test to see if two names name equivalent objects (The analog of LISP's EQUAL function).

Test to see if two names reference the same object (this is LISP's EQ function).

ORD - for ordered types only, given an object return the index of that object in the ordered type (this is a generalization of Pascal's ord function). For example convert a character to its ASCII representation.

ORD^{-1} - Given an index (integer) return the corresponding element of the ordered type (this generalizes Pascal's chr function).

Create a copy of an object.

All of these operations seem to have the property that most types will support them. It is not necessary that a type manager support all of these global operations, although it is not unreasonable to expect most type managers to support most global operations. Rather, it provides a means for providing global names for certain kinds of operations.

Some possible operations (and the first two above) exist so that the hardware can do some operations on all types directly without actually invoking the type manager involved. One possible use of global operations is that it makes it easy to write procedures that accept objects of variable type so long as the operations on that type that are needed by the called procedure are global operations. In this case any object of a type that supports the needed global operations can be passed to that procedure. For instance a sorting procedure could easily be written using the ORD global operation.

5. Instruction Format

AESOP has only one instruction, call_type_manager, and it is the purpose of this section to describe this very versatile instruction. First though it will be useful to motivate the need for only one instruction - call_type_manager.

Every object in AESOP is typed. This means that the object can only be manipulated by its type manager and not by any program that decides it should examine the object. For this reason it makes no sense to have an instruction that is not call type manager as it could not do anything. At first the reader might object that a procedure call operation is needed or that a "Go To" instruction is needed. However a procedure is just an object and, as such, all operations on it are provided by its type manager. Thus to "call" a procedure involves calling the procedure type manager and asking it to perform the call operation on a specified procedure using given arguments. Similarly a go-to operation involves calling on some type manager to cause the current instruction counter (?) to be changed.[1]

The basic format of an instruction is given in figure 3. The effect of the instruction is to call some type manager, passing the objects specified in the instruction as arguments and possibly returning some objects as results.

[1] This is in principle what happens in AESOP although the actual details involve the code segment type manager and basic data types and are described in a later part of this paper.

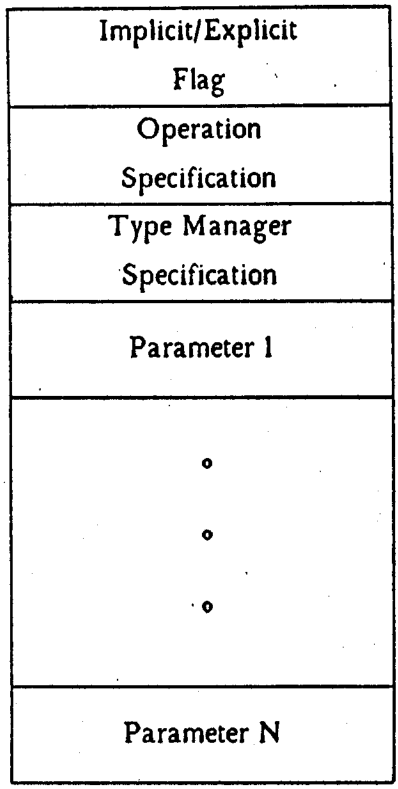


Figure 3. Basic instruction format.

The name of the type manager to be called is derived based upon the value of the Implicit/Explicit flag. If the flag says "Explicit" then the field type_manager is present and names the type manager to be called. If the flag says "Implicit" then the type_manager field is not present and, instead, the name of tne type manager is obtained by examining the first parameter. The type manager called in this case is the type manager corresponding to the type of the first parameter. Thus implicit calls to a type manager only work if the first parameter of that call is an object of that type.

At first it might seem that explicit type manager calls are all that are needed. However there is one case in which it is not sufficient to have just explicit type manager calls. Consider a procedure that has no expectations as to the type of one or more of the arguments passed to it. In

CLU terms these arguments are of type any. In some manner the procedure must be able to call the type manager of that object as otherwise there was little point in passing the object as an argument. The implicit method of calling a type manager provides an easy, and obvious, way to solve this problem. One might argue that instead of using the implicit type manager call mechanism the procedure could simply determine the type of the object and then use explicit type manager calls. If the type of the object is passed as a parameter (e.g. the name of the type manager is passed) then there is no problem. If not, the called procedure must find out the type of the object in some way and this is a problem. In the AESOP formalism the only way to derive this information is to perform some operation on the object but this involves knowing the name of the type manager which ... (Catch-22![1]). Thus some escape is needed and AESOP has chosen the implicit type manager call mechanism.[1] This approach also has the desirable property of efficient encoding of instructions since in many cases the type manager field will not be needed. On the other hand it is basically a special case mechanism and is somewhat "unclean" because of that. An equally dirty and special case mechanism would be to provide two instructions in the machine - the explicit type manager call and an instruction to extract the type manager name corresponding to an object (thus showing the "magic" through to the architectural level). A third possibility, not considered in depth to this point in the design of AESOP, is that the extract type manager name operation is not the responsibility of the type manager of that object but, rather, is the responsibility of some other type manager. In this case no problem exists so long as the name of that type manager is globally known. On the other hand, explicit type manager calls are themselves needed since not all operations on a type take an argument of that type (e.g. create and ORD^{-1}).

[1] It should be clear, however, that the underlying implementation must have some way of determining the type of any object without calling a type manager in order to make implicit type manager calls work. The way of doing this is simply "magic" at the architectural level.

Operation Type Flag	Operation Number

Figure 4. Operation specification in an instruction.

The operation to be called is specified by the operation field of the instruction (see figure 4). The operation field consists of two subfields - the operation type flag and the operation number. If the operation flag says "Global" then operation operation_number in the global operation name space of the specified type manager is invoked. If the operation flag says "Local" then operation operation_number in the local operation name space of the type manager is invoked. The operation number is just a positive integer and is used to index into the appropriate operation name space of the called type manager. This representation requires that the operation number be known at compile time. This may be somewhat restrictive when arguments of variable type are passed to a procedure as arguments since the correct operation number may not be known at compile time. This can, however, be countered in at least three ways. One way is to modify the operation number field to allow either an index or the specification of an LNS/GNS slot that contains the name of an integer. In this way the correct operation number can be passed as a parameter and then specified in the instruction appropriately. A second possibility is that the procedure that is to use the operation with an unknown operation number may instead be passed a type manager that implements the operation desired with an interface to that operation that the called procedure expects (e.g. the correct operation number and parameters). In a similar way, the third possibility is to pass a procedure as a parameter and allow that procedure to perform the operation. In each of the second two choices, the called procedure has, in effect, defined away the problem by modifying its interface to require an object as an argument that can be manipulated in an interface specified manner. Since the number of cases in which the operation number is unknown is probably small, we reject modifying the instruction format to allow a variable operation number. Instead, one of the second two methods should be used when a variable operation number is needed.

The type_manager field, if present, names the type manager to be called. It does so by specifying a slot in the executing procedure's LNS or GNS in a manner identical to that used by the parameters, as is described in the next paragraph.

The instruction contains specifications of the parameters to be passed to the called type manager. Each parameter is specified in a manner pictured in figure 5. The local/global flag specifies whether the LNS or the GNS is to be used to find the object to be passed as a parameter. The index is the index of the entry in the LNS or GNS (as appropriate) in which the name of the object to be passed will be found. The last-parameter? flag is turned on if this parameter is the last parameter to be passed to the type manager.

The instruction format allows the instruction interpreter to know precisely how many parameters (>0)[1] are to be passed and it allows a variable number of them to be passed. The

Local/Global Flag	Index	Last Parameter?

Figure 5. Parameter format.

[1] Note that procedures with no parameters are not possible. If this turns out to be too restrictive, an additional flag, any-parameters?, can easily be added to the instruction format.

variability is clearly needed for flexibility in writing type managers and procedures while allowing the precise determination of the number of parameters passed allows some run time checking of the correctness of the call by the hardware.

6. RCG's

Now this paper will examine the mechanism used to implement extended type objects and provide the capability for access restriction and access revocation. The surprising fact is that these three activities, seemingly different at first, can be regarded as being special cases of the same general mechanism. The property that all three of these activities have in common is that they all permit different users of an object to have different views of that object. Extended type objects hide the representation of an object from its users and only allow its interface specifications to show through; access restriction presents an object that does not support all of the operations normally associated with objects of that type and access revocation takes this to the extreme that no operations are supported on that object. Thus the common property of these activities is that they present a colored view of the object, thus the mechanism used to implement these activities is called the RCG (Rose Colored Glasses) mechanism.

In thinking about RCG's the following analogy may help. An RCG is like a window that allows you to only see some aspects of an object while other aspects are hidden by the window. The window is unbreakable so that the restricted view that it provides cannot be avoided unless another window is found. Moreover, the the presence of the window cannot be detected in any way.

The basic mechanism used to perform all three functions is illustrated in figure 6. It is inspired by the access revocation mechanism proposed by Redell[4]. An RCG is a string of three items. The first is the name of a type manager. This name specifies the type of object that is seen

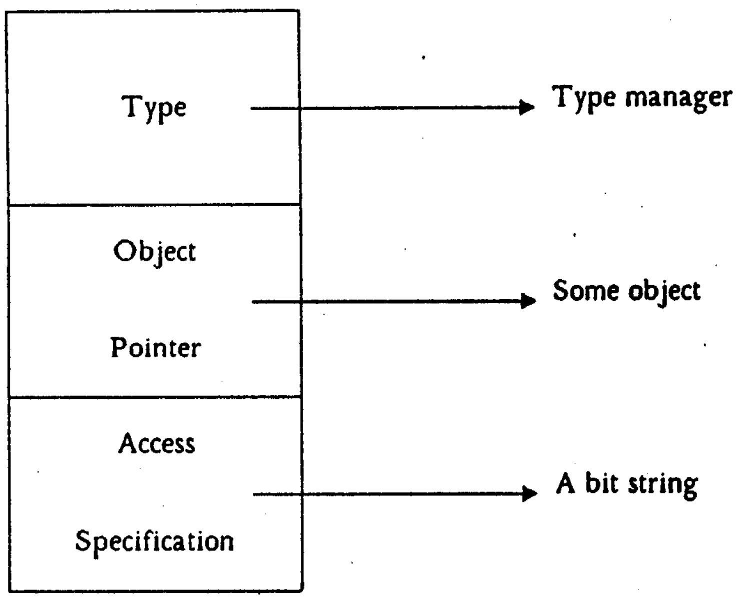


Figure 6. An RCG.

when the RCG is referred through. The second field is the name of an object. It is the actual object referred to when an attempt is made to reference through the RCG (e.g. use the name of the RCG). The third field is a bit string (actually the name of a bit string!) which specifies which operations can and cannot be performed upon the viewed object (the object "seen" in the RCG provided "window"). This field is used to restrict access to the object provided by the RCG. In order to illustrate the interactions of these three fields in the RCG the next few paragraphs will show how this basic mechanism can be used to solve the varied problems mentioned earlier.

The first problem is that of access restriction. Suppose that a program wants to pass an object Y to another program but does not wish certain operations to be performed upon that object. For instance modifications of the object may be undesirable. In this case the first program creates an RCG by calling RCG$access_restrict(Y, AR, X) producing the structure of objects illustrated in figure 7 and passes the name X to the called program. The RCG in this case specifies that the name X refers to an object of type FOO (the same as that of Y). The representation, in some sense,

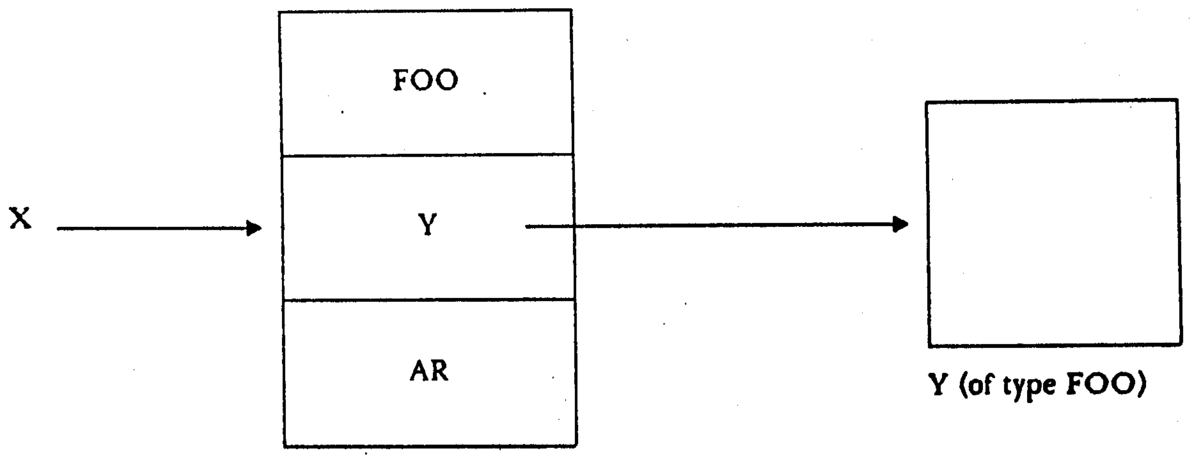


Figure 7. An RCG, X, being used for access restriction.

of the object X is the object Y so that references to X are actually references to Y. The bit string AR specifies the restrictions upon the way that the called program may use the object it has been passed (e.g. it specifies the permissible operations that can be performed on the object Y).

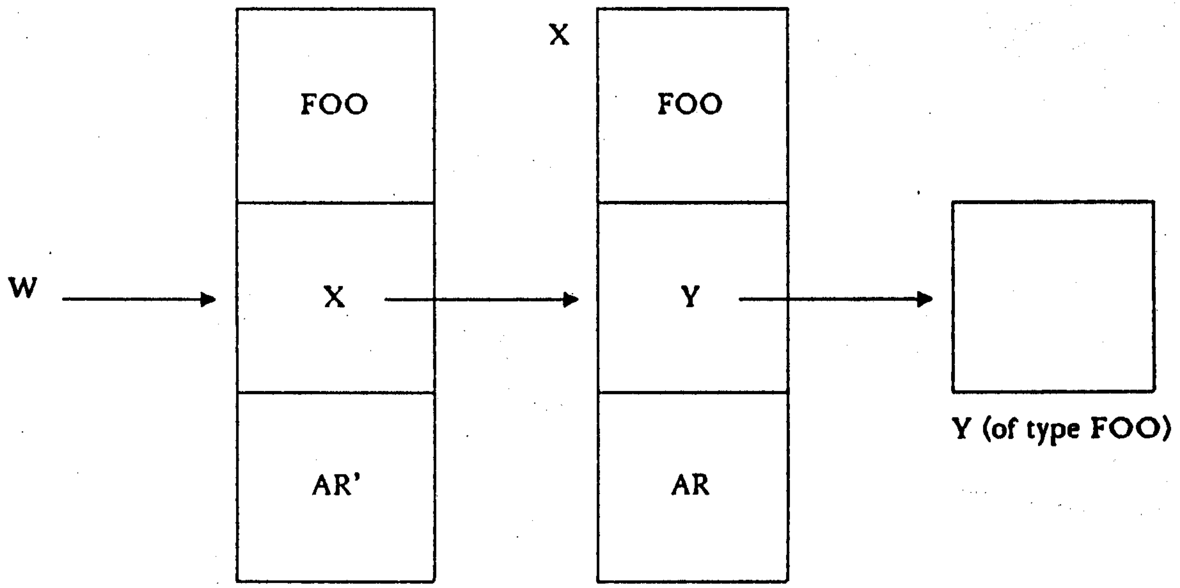


Figure 8. Chained RCG's in the access restriction mode.

If the second program wanted to pass along the object X to a third program and restrict the third program's access to the object it can also do so. In this case a picture such as that in figure 8 results after calling RCG$access_restrict(X, AR', W). The second program then passes the name W along to the third program. W is an RCG that specifies that the viewed object is of type FOO, is represented by the object X and has the access restrictions specified by AR'. When the third program uses the name W it "sees" the object Y of type FOO. Any operations performed by using the name W result in operations on the object Y. This means that the only way that users of W (and X) can possibly detect the presence of the RCG(s) is by noting that certain operations on the object do not work. The access allowed to the object X by users of the name W are specified by the minimal access rights specified by AR and AR'. Thus if AR and AR' specify access restrictions then the resulting access restrictions are the OR of AR and AR' and if AR and AR' specify access privileges then the resulting access privileges are the AND of AR and AR'. In this way the effect of RCG's as access restrictors is cumulative.

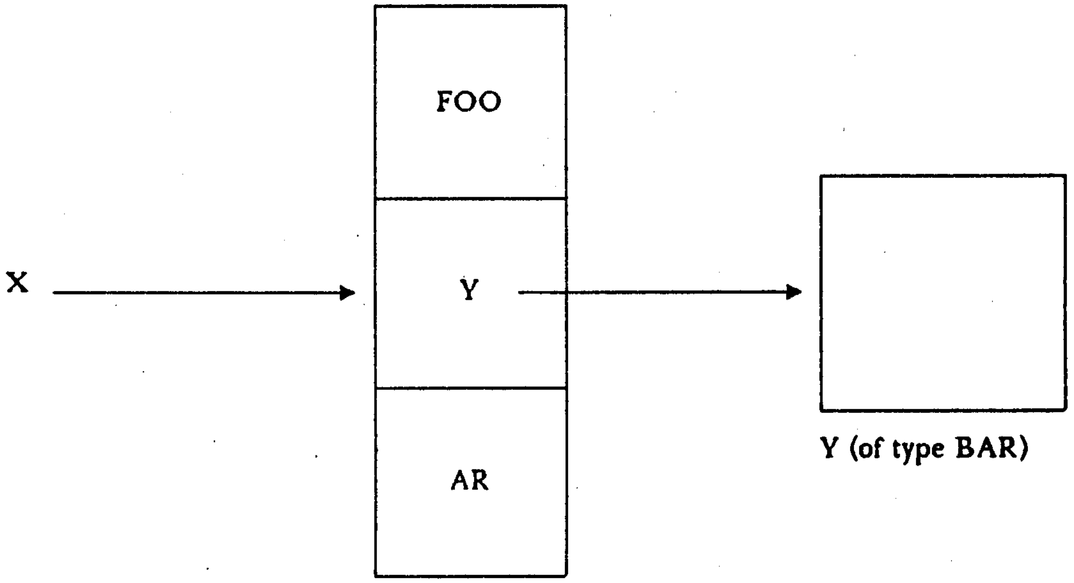


Figure 9. Type extension using RCG's.

An interesting side issue is whether EQ(W,X) is true in figure 8.[1] It is certainly the case that EQUAL(W,X) is true since W and X ultimately refer to the same object. EQ, however, is another story. Since the two views of Y provided by W and X are not (necessarily) the same, then EQ(W,X) would seem to be false. On the other hand, any changes effected by using one of W and X results in a direct and immediate effect in what users of the other name see. Thus from this point of view EQ(W,X) would seem to be true. The case is not particularly strong in either direction so AESOP makes the arbitrary choice that EQ(W,X) is false. This choice has the advantage that EQ(W,X) can be implemented by simply looking at the names (e.g. UID's) W and X and comparing them. In the long run the addition of a third kind of equality, RCG equality, may be needed to handle the case represented by figure 8.

Another use of RCGs is for type extension. Suppose that it is desired to create an extended type object, which will be named X, of type FOO from an object named Y of type BAR. Figure 9 illustrates how RCG's would be used to accomplish this. In this figure X is an RCG that specifies that the use of the name X refers to an object of type FOO and that that object is represented by the object Y. The access restrictions AR specify which operations on the object may be performed when using the name X. This enables the creator of the extended type object X (presumably the type manager for objects of type FOO) to create objects of that type with varying restrictions on access to that object.[2] The only things that can be done with the name X are to copy the name and to pass the name X to the type manager FOO to have operations performed on the extended type object X. The type manager FOO, however, can "unseal" the RCG and get at the representation object Y. This is performed by the operation:

[1] And also the more general issue of when EQ(W,X) is true for two RCG's W and X such that W and X specify access restrictions on (ultimately) the same object.

[2] This could also be done by creating the object X with full access rights and then creating a second RCG that specifies access restrictions.

RCG$unseal(X, rep_object, access_restrictions)

Unseal is passed the name of the object X, an RCG, and returns in the variable rep_object (which is just a slot in the current LNS) the name Y (the representation of the object X) and the access restrictions AR are returned in the variable access_restrictions. This then allows the type manager to:

1. Determine which operations are permitted on the object X by examining the variable access_restrictions.
2. Perform operations on the representation object Y.

No other facilities seem necessary at the architectural level to allow the type manager to perform its job.

Clearly if the RCG named by X is to truely "seal" the object, then permission to perform the unseal operation must be restricted to the type manager FOO. To this end AESOP checks that the procedure attempting to perform the unseal operation is part of a type manager and that the name of that type manager is the same as the name specified as the viewed type in the RCG that is being unsealed. Thus the name of the type manager is the key that allows the RCG to be unsealed. If both conditions are satisfied then the unseal operation is allowed, otherwise an error is indicated.

This implies some additional complication in the procedure call mechanism since a procedure implementing an operation on some type may be calling procedures that it does not want to have the unseal capability. When a type manager makes a procedure call it must be careful to specify whether the called procedure is or is not a part of the current type manager for unsealing (and, as will be seen, for sealing) purposes. To this end there are two kinds of procedure calls: internal_call and external_call. An internal_call does not change the fact that some type manager is being executed while an external_call suspends the ability to unseal (and seal) until the called procedure returns.[1]

In a similar manner the ability to perform the "seal" operation must be restricted. If it were not restricted then any program could seal an object making an object of arbitrary type. When this object was passed to the type manager for that type the representation of the extended type object might be incorrect. This could result in incorrect operation of the type manager. The type manager could protect itself from this kind of misbehaviour,[2] however this seems to be an added burden on the programmer that is best avoided (especially since the programmer is unlikely to remember to perform the necessary checking at all of the appropriate times). For this reason the seal operation is restricted in the following manner:

1. It may only be performed by a type manager.
2. The field "type" in the created RCG is filled in with the name of the type manager performing the seal operation.

With this restriction only a type manager may create an RCG that makes an object look like an object of that type (or equivalently, only a type manager may create extended type objects of that type).

[1] One alternative is to "unseal" all of the appropriate objects upon entry to the type manager. This is workable although it does add complication to the procedure call mechanism. This was not chosen, however, since it prevents the type manager from ever regarding the objects that it implements as anything but their representation. Another possibility is to allow unseal in the first procedure called in a type manager and not in any subsequent ones.

[2] Perhaps by including an unforgeable key in the representation for each object (this is similar to the mechanism proposed by Henderson[2]). Such a key might be the name of an object that the type manager never allows to be passed beyond its control.

The operation that creates an RCG for use as a type extender allows an access restriction field to be specified for the newly created RCG. The possible high level semantics of this are interesting. It allows one type manager to implement a collection of "types",[1] each a restriction of one basic type. For instance, with file as a basic type, by setting AR appropriately the "types" read-only-file, append-only-file, stream-file, direct-access-file and others can be created.

In order to illustrate the revocation of access to an object it is necessary to show how RCGs may be modified once created. Figure 10 shows the basic mechanism. X is an RCG that has as its type field RCG and its object pointer names another RCG, Y. Assume for the moment that the access restrictions mentioned in X, AR, permit all access to the RCG Y. By using the name X it is then possible to modify the RCG Y. In particular the access restrictions and the object pointer in Y can be modified. Note, however, that the type field cannot be modified for to allow such modification would lead to the same problems as mentioned in the case of restricting the ability to perform the "seal" and "unseal" operations. If AR does not allow full access then, as usual, some operations may not be allowed on Y (such as modifying the object pointer). A program possessing the name X clearly has great power over the contents of the RCG Y and so such possession must be carefully controlled.

In this regard the question arises as to how the object X came into being. It cannot just be created at random whenever the need for it arises, rather its creation must obey restrictions so that the security provided by RCG's is not compromised. To this end the only time that objects such as X are created is as part of the creation operation of other RCG's. Looking at figure 10, assume that object Y is an RCG that was created by the seal operation. As part of that seal operation the object

[1] These are not types in the normal AESOP sense since there is a one-to-one relation between AESOP types and type managers.

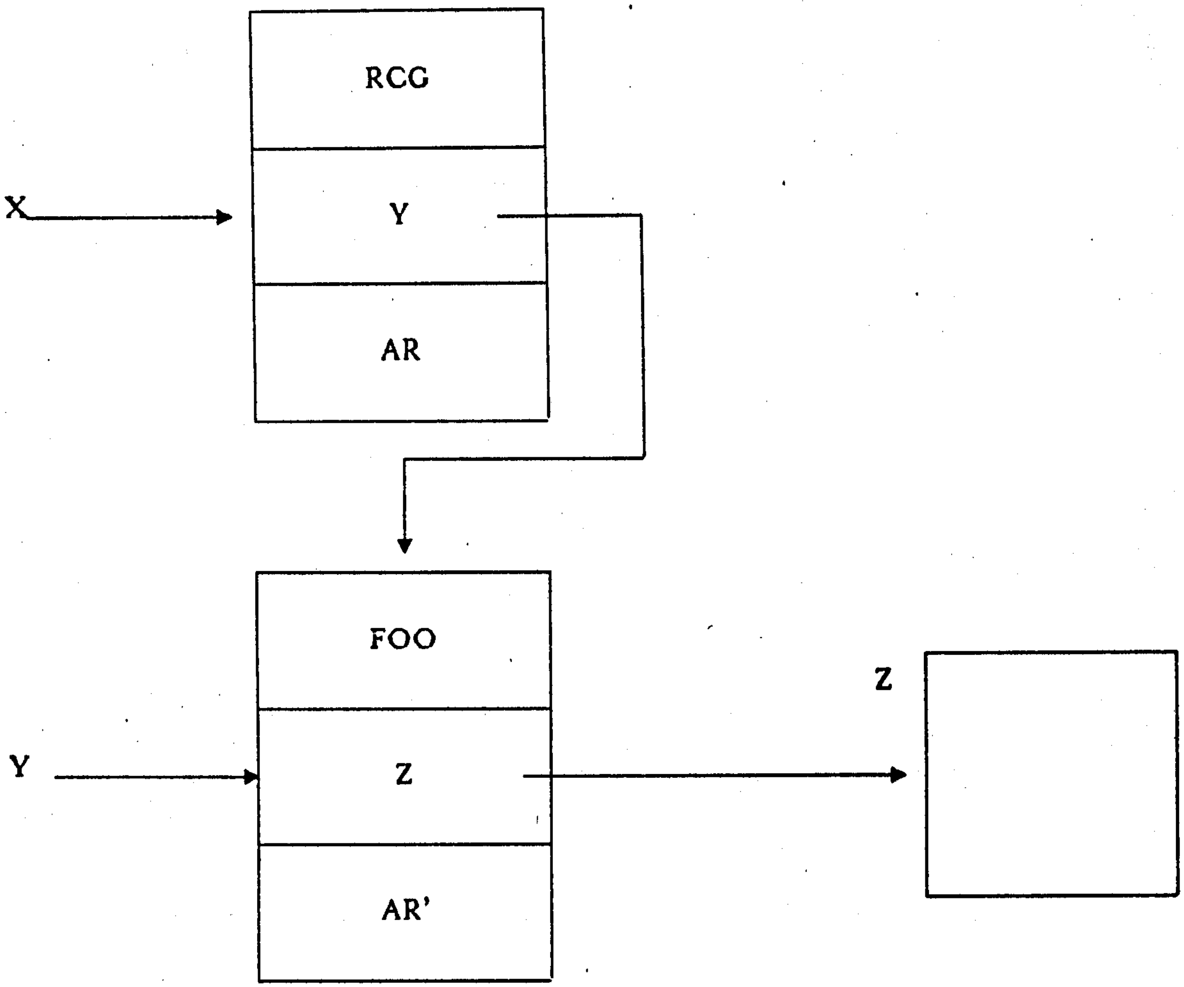


Figure 10. The use of an RCG to modify another RCG.

X should optionally be created and its name returned. If, the other hand, Y was simply performing an access restriction function then the object X should optionally be returned at the same time that the object Y was created. In this way the ability to create a given RCG also gives the ability to modify that RCG and there is no other way to gain that ability. Thus only the type manager for a given type may manipulate the sealing RCG's for objects of that type. Similarly, a program restricting access to an object by creating an RCG is the only program that can modify that RCG.

In this way the security provided by RCG's is protected by the correctness of the procedures creating these RCG's.

The way of performing access revocation should now be clear: To give revocable access to the object Z in figure 10 the RCG Y should be created and at the same time the RCG X should be created. The name Y should be passed to the program that is to be given revocable access and the name X should be remembered. Later when it is desired to revoke some, or all, access to the object Z the object X provides the means to do so. If complete and permanent revocation is desired, then X can be used to destroy object Y, making any outstanding references to it invalid. If partial revocation, access enhancement or non-permanent, complete revocation is desired then the object X can be used to modify the access restriction field in Y appropriately. This is done by calling RCG$set(X, Y, AR') to change the access restriction field to AR' from AR.

Up to this point this section has presented the basic RCG mechanism and examples of how to use it in solving various problems. The next few paragraphs will present a few rules for using RCG's.

The first issue is finding the actual object referred to when using a name that is an RCG. Let X be the RCG that is named. Then the object that is referred to is:[1]

1. If X.object_pointer names another RCG and if X.type = (X.object_pointer).type then the object referenced by X.object_pointer (i.e. recurse on these rules).
2. If X.object_pointer names another RCG and if X.type ≠ (X.object_pointer).type then X.

[1] The notation X.Y is used to denote field Y in the RCG X.

3. If X.object_pointer does not name another RCG and if X.type = type_of(X.object_pointer) then X.object_pointer.

4. If X.object_pointer does not name another RCG and if X.type ≠ type_of(X.object_pointer) then X.

Cases 1 and 3 correspond to following a chain of RCG's being used as access restrictors. Cases 2 and 4 correspond to the use of RCG's as type extenders. The effect of the rules are that chains of RCG's are followed until an RCG being used for type extension is found or until a non-RCG object is found.

The determination of permitted access to an object that is referred to through an RCG must also be specified. Again there are three cases to consider:

1. If X.object_pointer names another RCG and if X.type = (X.object_pointer).type then min_access(X.access_restrictions, access_to(X.object_pointer)).

2. If X.object_pointer names another RCG and if X.type ≠ (X.object_pointer).type then min_access(X.access_restrictions, (X.object_pointer).access_restrictions).

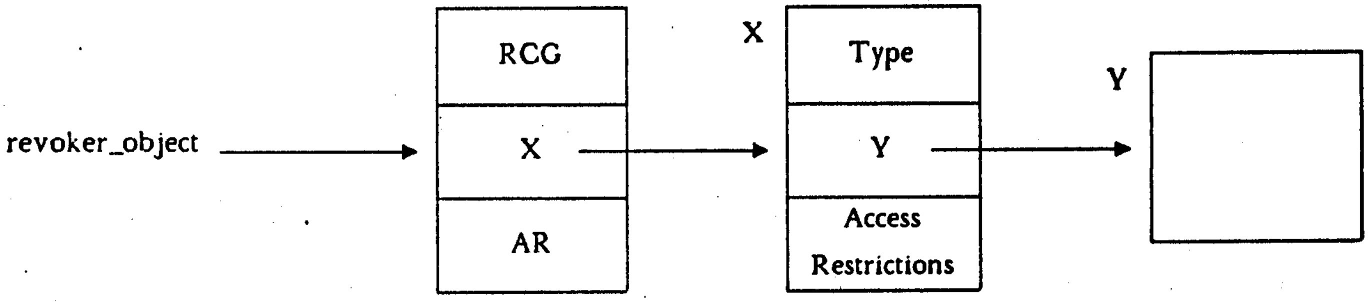
3. If X.object_pointer does not name another RCG then X.access_restrictions.

Case 1 corresponds to the case of RCG's in a chain being used as access restrictors. Case 2 is RCG's in a chain being used as type extenders. Case 3 covers the case of non-chained RCG's. The access is used to determine whether or not an operation may be performed upon some object. At the time that an object is passed to its type manager the type manager must be able to exercise the name and discover what the effective access to the object is. The type manager can then use this information to determine whether or not the requested operation can be performed upon that object. Thus the enforcement of access restriction is the responsibility of the type manager while the hardware provides the basic mechanism to implement the access policy.

These rules for RCG's now allow some of the philosophy behind RCG's to be discussed. The rules presented clearly differentiate between the two uses of RCG's - as type extenders and as access restrictors. In fact the two uses almost represent different objects. AESOP has recognized that the two uses are close enough that with a slight push a single mechanism can provide them both. The second interesting philosophical issue is "Who provides RCG's?". This discussion has presented RCG's as a type. In fact, another way of thinking of them is as provided by the name interpretation mechanism. If thought of in this way, the rules in the preceeding paragraphs may seem more rational.

Finally we review the various operations permitted upon RCG's (see figure 11). The allowable operations are the following (output arguments are in **bold face**):

RCG$seal	(sealed_object, access_restrictions, **sealing_object, revoker_object**)
RCG$unseal	(sealing_object, **sealed_object, access_restrictions**)
RCG$access_restrict	(protected_object, access_restrictions, **protected_object, revoker_object**)



Where X is a sealing or protecting object and
Y is the sealed or protected object (e.g. the "rep" object).

Figure 11. Example of the use of operations on RCG's.

RCG$extract. (revoker_object, type, sealed_object, access_restrictions)

RCG$set (revoker_object, rep_object, access_restrictions)

Another operation that is intimately related to RCG's is the effective_access operation:

RCG$effective_access(X, access_restrictions)

which returns the effective access permitted to the object named by X. (If X is not an RCG then access_restrictions is returned to indicate full access).

These are all of the operations that can be performed on RCG's. Using these operations it is possible to implement type extension, access restriction, access revocation and even change the representation object for extended type objects. No further mechanisms are provided or needed in AESOP for these purposes.

7. Control Structures

A number of architectural features of AESOP have been described thus far in this document. The one important aspect of the architecture that has not been described is the method of flow control in the machine. This section describes the method of procedure calls, branching within a procedure, type manager calls, error mechanisms and extended control mechanisms.

The basic unit of execution within AESOP is the code segment which is a linearly ordered set of instructions. The instructions within a code segment are executed sequentially from beginning to end and there is no way to begin execution of a code segment except at the beginning or to cause execution of a code segment to skip execution of some instructions within the code segment.

Type manager calls are the basic mechanism of controlling the point of execution in AESOP. Every instruction is a type manager call. When an instruction is decoded the result is the name of a type manager to call, the name of the operation to be performed and a list of objects to be passed to the type manager as arguments. The fact that a type manager is in execution is remembered, as is the name of the called type manager. From this point on the call of the type manager looks the same as the call of a procedure.[1]

The basic procedure call mechanism consists of determining the name of the procedure to be called, supplying the input arguments to the procedure and specifying where the output arguments are to go (in the LNS or GNS of the calling procedure). Once the name of the procedure is determined an LNS is created and filled in with the template LNS of the called procedure. At this point the names of the input arguments are copied into the correct spots in the newly created LNS (by convention these are the first N slots in the LNS for a procedure that has N input parameters). Execution is now continued using the newly created LNS as the LNS, the GNS remains the same as the callers GNS and the newly executing code segment is the initial code segment of the called procedure. When the procedure returns, the output parameters are copied from the LNS of the called procedure into the correct slots in the LNS and GNS of the calling procedure. Procedure return is initiated in one of two ways. If the initial code segment of the called procedure should ever be completely executed then a return is initiated at that time. The other way is to execute a procedure$return call explicitly. The second mechanism is provided in case it should prove awkward to write all procedures so that the last instruction in the initial code segment is the last instruction that the procedure wishes to execute.

[1] In some sense the procedure call mechanism is the basic mechanism in AESOP although from an ISP point of view the procedure looks like another extended type. This brings up a tricky point as to which comes first: procedures or extended types. The view taken here is that they are intertwined to a point that trying to separate them would serve no useful or practical purpose.

The instruction execution mechanism presented so far only allows unconditional execution of code. There is, however, a clear need for the ability to execute sequences of code conditionally. To this end it is possible to call the code_segment type manager and ask it to execute some given code segment using a given pair of name spaces as the LNS and GNS for the execution. This mechanism is sufficient to implement the needed conditional execution of code segments once the appropriate operations are provided on one or more of the basic types. A sufficient mechanism is to provide the operation

```
boolean$branch_t_f(boolean_value, true_code_segment, false_code_segment,
    LNS, GNS)
```

that tests the value of the boolean variable boolean_value and if it is true executes true_code_segment using (LNS, GNS) as the environment for the execution and executes false_code_segment using (LNS, GNS) as the environment for the execution if boolean_value is false. For efficiency, other basic types may also provide branching constructs. Examples might be operations to branch based upon the relative values of two integers or upon the relation between two characters or character strings.

This mechanism for flow control has been chosen since it seems to lend itself to current methods of writing structured programs. Moreover, this architecture almost forces this style of programming which may be a good idea.

Using the basic conditional execution mechanism it is possible for the user to write his own control structures. For instance to write a "do while" looping construct the following procedure will suffice:[1]

[1] The code is written in a PL/I-like style and the meaning should be fairly obvious.

```
do_while:procedure(boolean_variable, code_seg, lns, gns);
        dcl boolean_variable name_space_slot boolean,
           code_seg code_segment,
           (lns, gns) name_space;

        if boolean_variable
          then do;call code_seg$execute_code_seg(code_seg, lns, gns);
                  call do_while(boolean_variable, code_seg, lns, gns);
                end;
        return;
        end do_while;
```

This procedure is relatively straightforward. A boolean variable is tested and if it true the code segment, which is the body of the loop and should modify the value of the loop control variable, is executed in the correct environment and then do_while is called recursively to re-test the variable and possibly reexecute the code segment. Note that this recursive call is an example of tail recursion and so need not be expensive. The only unusual item in this procedure is the declaration of the variable boolean_variable. It is intended to be a name that allows access to a particular slot in the callers LNS so that the do_while procedure may test the value of the loop control variable (conceptually boolean_variable is a pointer to a name slot that can contain boolean values). Note that just passing a boolean variable (e.g. dcl b boolean; ...call do_while(b,cs,lns,gns);) does not work since AESOP passes arguments by assignment (in the same way as CLU does). The parameter boolean_variable must be a cell that takes on boolean values and whose contents is changed by the body of the loop. It can be implemented in at least two ways. A substring type could be created that allows access to a substring of another string but nothing else. Then a substring object could be created that has as its representation the appropriate slot in the caller's LNS. This object could then be passed to the do_while procedure. This approach has the advantage that the substring type manager, which is probably in hardware, enforces the access restrictions. The second approach is that what is passed to do_while is a pair (name_space, index) which specifies that the variable to be examined is in slot index of the given name space. This approach has the advantage that no additional architectural mechanisms are needed to implement it (as it is a language defined runtime convention) and the disadvantage that to provide access to the one variable of interest it is necessary to give access to the entire name space. This method is workable and safe provided that the compiler is capable of performing the needed checks or if such checks are deemed unnecessary. Both mechanisms are sufficient although the enforced protection of the first mechanism seems to be worth the added complexity of a new basic type.

The reader will note that the procedure above can not really exist in AESOP in the given form. Rather the procedure consists of the two code segments A (the initial code segment) and B. They have the following form:

```
A:call boolean$branch_if_true(boolean_variable, B, lns, gns);
  end A;

B:call code_seg$execute(code_seg, lns, gns);
  call do_while(boolean_variable, code_seg, lns, gns);
  end B;
```

This structure occurs because there is no go-to statement, rather only a means to execute code segments conditionally.

The subject of error control in the machine is an important one. Procedures need to be able to signal errors, the hardware needs to signal errors when an error in using the basic types occurs and when hardware errors (failures) occur. This area is one where further thought is needed. The only fairly firm decision that is proposed is that when a type manager detects an error, the error must be reflected at the point of call and not at the actual point of error. If this were not the case, there might be a means to compromise the integrity of the type manager. The ability for procedures to "catch" errors is probably needed so that errors can be detected and corrected by procedures and not reflected back to the caller unnecessarily.

8. Remaining Issues, The Ultimate Truth and Conclusions

This paper has presented the basic architecture of AESOP. The description is not cast in concrete nor is it complete. This description is intended as a starting point for a continuing design effort and comments regarding the design are solicited.

There are a number of issues that remain to be specified in AESOP. The notion of process is missing. Should the basic architecture support processes and if so in what form? Perhaps the virtual processor mechanism of Reed[5])? If processes exist, must the notion of processor state be introduced? Is the processor state an object that can be manipulated? Or is it outside of the typed environment? What synchronizing mechanisms are needed? What is the interprocess communication mechanism?

The whole area of I/O is unexplored. How should standard devices such as disks, tapes and line printers be integrated into the system? How should communication networks be added? In what ways should the architecture support objects that are shared across machine boundaries?

What about protection mechanisms? The current protection mechanism in AESOP consists of RCG's and the fact that names cannot be forged (i.e. it is a capability machine). This means that the only way to get access to an object is if someone who already has access to the object gives it away. RCG's provide the means to limit the amount of access given away. The mechanisms as presented certainly seem sufficient to implement hierarchical models of protection. Solving problems such as the mutual suspicion problem does not seem to be as easy to do using the given mechanisms. Are the mechanisms sufficient to solve all of the "interesting" protection problems that need to be answered? If one views AESOP as the model for a personal computer, as opposed to a multi-user shared processor, are the protection mechanisms sufficient?

This paper has very carefully avoided the issue of storage management. Are objects deleted automatically when no longer needed (i.e. garbage collection) or are objects only deleted explicitly? Is a hybrid scheme the right approach? Does the architecture need to have "hooks" in it to help solve various aspects of the small object problem? Must it provide "hooks" to allow cataloging and permanent storage of objects?

In conclusion it is useful to note a number of unique features of AESOP. There is, in reality, only one type of basic object in AESOP and that is "string_of_names". Every other object is constructed out of such objects. There are other primitive types such as integers and booleans but they are objects that are probably never realized explicitly, rather the names of these objects are sufficient to represent them. The architecture only supports one instruction - type manager call. All of the other "obviously needed" operations are in actuality calls on one type manager or another. The RCG mechanism is a uniform mechanism that serves to solve the problems of type extension, access restriction and access revocation. The uniformity of the mechanism seems to be unique in the area of machines that support objects. Finally AESOP is a machine architecture that is designed to support completely the notion of typed objects.

Acknowledgement

The initial design of AESOP (then known as the 510 Machine) was performed in January, 1978. It was the collabrative effort of the author, Warren Montgomery, Dave Reed and Karen Sollins and their efforts are gratefully acknowledged. Many of the ideas presented in this paper are the result of that design effort but the author takes full responsibility for any errors of omission or commission that are present in this paper.

References

[1] Heller, Joseph, Catch-22, New York, Modern Library (1966, c1961).

[2] Henderson, D.A., "The Binding Model: A Semantic Base for Modular Programming Systems," MIT-LCS TR-145, February, 1975.

[3] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," *CACM 20*, 8 (August 1977), pp. 564-576.

[4] Redell, D.D., "Naming and Protection in Extendible Operating Systems," M.I.T. LCS Technical Report TR-140, November 1974.

[5] Reed, D.P., "Processor Multiplexing in a Layered Operating System," M.I.T. LCS Technical Report TR-164, June 1976.

[6] Saltzer, J.H., "Naming in Information Systems," chapter 5 of 6.033 notes, fall, 1976.

[7] Walker, R.D.H., "The Structure of a Well Protected Computer," Ph.D. Dissertation, University of Cambridge, England, December, 1973.

[8] Wulf, W.A., "ALPHARD: Towards a language to support structured programming," Carnegie-Mellon University Dept. of Computer Science, April 1974.

[9] Wulf, W.A., Levin, R. and Pierson, C., "Overview of the Hydra operating system development", *Proc. Fifth Symposium on Operating Systems Principles*, 131, November 1975.