An Object-Oriented Command Language

RICHARD SNODGRASS, MEMBER, IEEE

Abstract—This paper describes Cola, an object-oriented command language for Hydra; Hydra is a capability-based operating system that runs on C.mmp, a tightly coupled multiprocessor. The two primary aspects of Cola, that it is a command language for Hydra, and that it is based on the object paradigm, are examined. Cola was designed to effect a correspondence between capabilities in Hydra and objects that are supported by the language. Cola is based on Smalltalk in that it uses message-passing as a control structure to allow syntactic freedom in the expression of commands to the system. Cola objects are arranged in a hierarchy, and the message-passing mechanism was designed to exploit this structure by automatically forwarding an unanswered message up the hierarchy. Two ramifications of this mechanism, automatic inheritance and shadowing, are discussed. An evaluation of the design decisions is also given.

Index Terms—Capabilities, command language, knowledge representation languages, message-passing, multiprocessors, object-based languages, object hierarchies, Simula, Smalltalk.

I. Introduction

COLA is an object-oriented command language which grew out of a need for a comfortable user interface for Hydra [51], a capability-based operating system that runs on C.mmp [48], a tightly coupled multiprocessor. Hydra provides a comprehensive set of facilities to the user, yet the previous command language for Hydra (called the CL[1]) presented these facilities as an unrelated collection of procedures. The overall character of the operating system was not reflected in the way that the user interacted with the command language. In the design of Cola, much effort was expended to mirror the philosophy of the operating system in the command language. The incorporation of *objects*, similar to Simula classes, in the command language was a result of this objective.

That the concept of class might be a valuable addition to command languages is suggested by an analogy between the development of command languages and general purpose languages. When the hardware is first introduced, one programs in assembly language, due in part to a lack of higher level software, an inadequate understanding of what structuring concepts are useful, and a desire to make the most of a scarce resource. Assembly language programs can be characterized by their utilization of the full power of the hardware by building on only the basic facilities available. One command language analog to assembly languages is the OS/360 Job Control Language [31], which shares these same properties. As with assembly language, anything is possible in JCL, and almost everything is difficult. The first step toward making the machine easier to program was Fortran, which provides a few basic control structures, such as DO-loops and subroutines, as well as a few data structures, such as arrays and COMMON storage. The George 3 Command Language [33], which includes conditional and looping statements, as well as user-defined macros, embodies some of the advances found in Fortran.

Algol was the next major development in programming languages; concepts such as data types, block structure, and recursive procedures first appeared in this language. Similar ideas can be found in the Burrough's Work Flow Language [9], IBM's CMS language [18], OSL/2 [2], SCL [6], and the CL, although they have been adapted for a command language domain.[2] For example, SCL uses the block structure of a control program to limit both the scope of variables (as in Algol) and the scope of operating system resources, and the CL interprets certain names to refer to objects in the file system rather than in main memory. The next major step in programming languages (one that is currently still in progress) is the introduction of abstract data types [39]. The concept of class first appeared in Simula [3], with Euclid [25], Alphard [50], and CLU [30] continuing the emphasis on abstraction and modularization. Despite the advantages inherent in these developments, the concept of abstract data types has not yet appeared in command languages.

A similar analogy between general purpose languages and operating systems also suggests incorporating classes into the command language. In one sense, an operating system is merely a large, complex runtime system for the user's program. This was true before multiprocessing, and applies even more with the advent of operating systems for personal computers, which are usually single language machines [26], [35]. The concepts introduced in programming languages tend to be transfered to the runtime systems, as well as the operating systems, which support them [20]. Thus, there has been a flurry of activity in recent years concerning object-based operating systems [21], [34], [46]. In systems such as these

Manuscript received April 13, 1981. This work was supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The author was supported by a National Science Foundation Fellowship.

The author was with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. He is now with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27514.

[1]The CL [37] is a general expression-oriented block structured programming language. The syntax resembles Bliss [49], the system implementation language Hydra itself was written in. Some features have been added (such as a capability data type) or altered (such as a more versatile assignment statement) to allow access to specific Hydra facilities.

[2]Lisp has also been taken as a starting point in the design of a command language [10], [29].

where the concept of object pervades the programming language, its runtime system, and the operating system itself, the command language should provide consistency by also supporting objects.

This paper describes an object-oriented command language (Cola) for an object-based operating system (Hydra). The first part examines the correspondence between objects in Hydra and objects in Cola. The second part considers structuring objects in ways that have been found to be useful in structuring knowledge. Hence, this paper investigates the two main aspects of Cola: that it is a command language for Hydra and that it is based on the object paradigm.

II. Cola as a Command Language

As indicated above, much progress has been made in providing more powerful command languages. There are, however, arguments for eliminating the command language completely, and instead embedding the functionality previously provided by the command language in the programming system, resulting in a programming *environment* with a uniform syntax and semantics [12], [16], [28], [38], [42], [43]. These arguments include not having to learn a different command language, being able to utilize the control and data structures present in the existing language when writing job control programs, and aiding the standardization task for command languages.

Despite these advantages, such an approach is not always appropriate. Each programming environment must be developed separately for each language that is desired. Also, it is very difficult to write a system in several different languages if each is supported by its own environment. Due to these reasons, plus the lack of existing systems that provide operating system primitives within the language, it is necessary to provide some kind of user-level interface that can interact with all these language systems. Since the operating system is the one entity shared by the users of the various languages, and it is the operating system that is providing the services that the command language refers to, it is appropriate to align the command language as closely as possible with the operating system [44].

A. Overview of Hydra

Cola is a command language that was designed for Hydra [51], a capability-based operating system that runs on C.mmp [48], a tightly coupled multiprocessor consisting of 16 DEC PDP-11's. The two most important attributes of Hydra—supporting a multiprocessor and implementing an object-based protection scheme—are relatively orthogonal. In Hydra, capabilities are protected pointers that refer to resources, both physical and virtual, called *objects*. Each object consists of an array of capabilities (a *clist*) and an array of integers (a *datapart*). A particular capability is referred to by specifying the index into a clist; similarly, data are referenced by indexing into the datapart. Since every resource in Hydra is associated with an object, all resources can contain data and capabilities. For instance, the datapart of a procedure object contains information relevant for execution and debugging (such as its trap and interrupt addresses, saved registers, etc.); the clist contains capabilities used by that procedure, including capabilities for *page objects*, which contain the code for the procedure.

In addition to using capabilities to support a very flexible protection scheme [7], Hydra provides powerful abstraction mechanisms. Objects are typed, and associated with each type is a set of procedures that can manipulate the representation of objects of that type. The user can define new types by specifying the representation of objects of that type in terms of types that are already defined, and by providing procedures that can perform operations on objects of the new type. In this way, Hydra types are analogous to the abstract data types of Simula or Alphard. Cola was designed to support this notion in a uniform manner.

Although capabilities and Hydra objects are quite different entities, the two terms are often used interchangeably. Hence, instead of refering to 'the length of the datapart of the object pointed at by the capability called "ACapa," ' one would refer to 'the length of the datapart of "ACapa." ' Also, the names of capabilities (and Cola instances, defined later) will be enclosed in double quotation marks, strings in single quotation marks, and the names of Cola classes will be in small capitals. These conventions will be followed in the remainder of this paper.

B. Objects and Message Passing

Cola is also based on the concept of objects. A *Cola object* is a potentially active piece of knowledge that communicates by sending messages composed of objects [17]. Cola is modeled closely after Smalltalk [13], [40]. Although it shares many characteristics with conventional languages incorporating abstract data types, Cola is unusual in that it uses message passing as a control structure to allow syntactic freedom in the expression of commands to the system, an important consideration in the design of an interactive language.

As an example of message passing, evaluating <object> + 4 means the message ' + 4' is sent to <object>, which interprets the message, and returns another object as a reply. In the procedural view, ' + ' would be an infix operator defined in the types within which the plus operator was valid. In Cola, the message ' + 4' is interpreted by the <object> itself. For instance, if the <object> were the integer 3, then the ' + ' in the message interpreted as ordinary addition, with the integer 7 returned; for the <object> 'a string,' the plus sign is interpreted as string concatenation, with the string 'a string4' returned. Each object can interpret a message in any way it sees fit. Message passing is entirely consistent with, and indeed, supports, the information hiding aspects inherent in abstract data types [19].

Every object in Cola belongs to a *class*, which is analogous to a type in other languages. The class, also an object, defines the messages that all its members can accept, as well as the semantics for each of the messages. It also defines the data structures that can reside in each member. The code that is associated with a class is shared by all the members of that class. Hence, the object 4 is a member of the class INTEGER; the object 'a string' is a member of the class STRING. The class structure is actually much richer than described here; more will be said when the object hierarchy is discussed in Section III.

C. The Cola/Hydra Correspondence

Cola was designed to effect a correspondence between objects (actually, capabilities) in Hydra and objects that are supported by the language: every object (capability) in the user's environment in Hydra is associated with an object in the user's environment in Cola. A system call embedded in a program can cause Hydra to perform an operation on an object referred to by capability mentioned in the system call. In the same way, a message can be sent to the Cola object associated with that capability to perform the operation. In responding to the message, the Cola object, as a side effect, executes the system call on that capability. Thus, there is no distinction between Hydra objects (capabilities) and Cola objects, resulting in an isomorphism between the two entities.[3]

To illustrate the Cola/Hydra correspondence, it is useful to examine the Cola equivalents of several object types defined in Hydra. The (Cola) class CAPA includes the operations which apply to all (Hydra) capabilities. The syntax for these operations is similar to record accessing in Pascal (corresponding to the conceptual view of an object as a record consisting of an array of capabilities (the clist) and an array of integers (the datapart), both indexed by integers). In the examples below, A and C denote instances of the class CAPA, S is a STRING, and x and y are INTEGERS. In accessing the datapart,

C . data [x to y]

returns a vector of INTEGERS, and

C . data length

returns an INTEGER specifying the length of the datapart. Similar operations apply to the clist. The operation

C . clist [x to y] vacate

removes the CAPAS in the selected slots of the clist.

A Hydra *catalogue* [1] is conceptually an array of capabilities indexed by strings (since it is still a Hydra object, it is implemented as a datapart and a clist). The Cola class CATALOGUE supports this conceptual view by interpreting a record access as a lookup operation on the CATALOGUE. So the operation

C [S]

looks up the entry S in CATALOGUE C and returns a CAPA. To remove an entry, execute

C [S] vacate

This correspondence is qualitatively different from the view of Hydra presented by the CL. The CL appears to the user as a set of predefined procedures which can operate on capabilities. These procedures correspond directly to the system calls available to Bliss programs running on Hydra [8]. Hence the CL is effectively an interpreted Bliss (indeed, if Bliss were an interactive language, then the CL as implemented would have added little functionality.)

Although the use of Bliss as a base for the command language resulted in a rather powerful user interface, it suffered from the restriction that communication with the operating system can occur only via system calls. For example, the conceptual view of objects as records consisting of arrays is not supported in the CL. Instead, one retrieves the data of an object by executing a system call (actually, in this case an ad hoc extension was made to the CL to allow array accessing to be done on Hydra capabilities). Thus the CL does not provide an adequate conceptualization of the objects defined in the operating system. In addition, the CL lacks the ability to *dynamically* define new types at the command level, and to define operations associated with these types which are reflected via system calls to the Hydra objects that are referred to by the command objects.

Cola provides this functionality and, as a result, presents to the user a different perspective on the operating system. Cola integrates the objects supported by the operating system into the language itself (due to the Cola-Hydra object isomorphism), eliminating the cumbersome system call interface. Note that the implementation still uses system calls, but this detail is hidden from the user. Instead, the user interacts with the command language using the abstractions with which he or she is familiar, namely those supported by the operating system.

D. Nonobject-Based Operating Systems

The object concept in Cola can be usefully incorporated into command languages for operating systems that are not themselves based on the object model. The function of an operating system is to provide resources for user jobs that can be manipulated by the job by executing system calls. These resources are essentially typed objects (such as files, directories, I/O ports, memory) with operations defined on them (such as print, add entry, send a character, reserve), although they are not always implemented as such in the operating system. It is this thinking that has motivated research in object-based operation system, and the use of objects in the command language is merely an extension of this concept, independent of the use of objects in the underlying system.

III. The Object Hierarchy

Although the main impetus for this research was the design of a command language for Hydra, the language that evolved out of this effort is interesting in its own right. Cola objects are useful not only as command language surrogates for objects (capabilities), but also have many properties that make them useful in models of computation [15] and in personal computer languages [14], [22], [45]. Since an object is an active piece of knowledge, one aspect of this research considered structuring objects in ways that have been found to be useful in structuring knowledge [32]. Although this aspect does

[3]This is not strictly true, since there are Cola objects such as INTEGER which do not have a Hydra analog. (Ideally, Hydra would handle integers as objects, but the implementation makes small objects inefficient.)

not concern user interfaces directly, the mechanisms which evolved were useful in a command language domain. Cola uses a hierarchical ordering of classes coupled with an execution semantics and binding mechanism to represent static and dynamic knowledge within the class structure.

A. Simula Subclassing

Simula, the first language to incorporate classes, used a *subclassing* mechanism to structure objects. A *subclass* is a refinement of a class: it inherits all of the procedures and data structures of its parent class, and augments these with its own procedures and data structures. Subclasses can also be refined further by their own subclasses, resulting in a tree structure. An *instance* of a class contains the values of all the data structures defined in all of its defining classes. The subclassing mechanism of Simula is a static, compile-time structure that almost completely disappears in the runnable version of the program. Smalltalk-76 has a similar mechanism that is partially interpreted at runtime [19].

There are several advantages inherent in such a scheme. Since the subclass inherits all of the traits of its defining class (its *superclass*), the code for the superclass need not be duplicated. Instead, the subclass uses all of the code it needs from its superclass, and adds traits of its own. Thus the mechanism provides a powerful structuring capability to the language. Other advantages, due to the message passing mechanism, will be discussed shortly.

The disadvantages of subclasses stem from the decision to place all of the data structures in the instance. Information associated with a superclass is replicated in all of the instances of that class. This arrangement complicates the modification of data associated with a class located several levels above the instance, and invokes the traditional consistency problems associated with redundant data (such as how does one make sure that *all* the instances of replicated data have been updated). There is thus an asymmetry in the distribution of data structures and procedures in Simula: procedures are shared by all subclasses of the class containing the procedure; data structures are not shared at all, but exist separately in every instance (note, however, that the *names* of the data structures are shared in the same manner as procedures). The Cola subclassing mechanism has been designed to allow the sharing of data structures while retaining the advantages mentioned above.

The Cola subclassing mechanism orders all objects in a hierarchical fashion. At the top of the hierarchy is the class (or object, since all classes are objects) called OBJECT. OBJECT is associated with a set of classes (called *subclasses* of OBJECT) through the relation *subclass*. Similarly, each of these classes is associated with OBJECT through the relation *superclass*. Subclass is a many-to-one relation—a class can (and usually does) have many subclasses. Superclass is a one-to-many relation—a class is restricted to having a unique superclass, but several classes can have the same superclass. Thus the subclass-superclass relation produces a tree structure of classes, as in Simula, with the class OBJECT being the root node.

B. Naming

Associated with every class are three kinds of variables, permitting flexibility in the placement of the values of the variables within the object hierarchy. These variables are class variables, instance variables, and temporary variables. *Class variables* are named in the class and are associated with values that reside in the class. They correspond to **Own** variables, in that their value is shared by all instantiations of the class. (In Fig. 1, "B" is a class variable of "Three," with value "0.05".) *Instance variables*, on the other hand, are named in a class, but are associated with values that reside in the *immediate subclasses* of the class. They correspond to the data structures defined in Simula classes, except that the values are stored in the next lower level, rather than in the leaves (i.e., the instances) as in Simula. ("C" and "D" are instance variables of "Three," with values in "Four" and "Five.") Values for *temporary variables* are created on every invocation of the object they are associated with using the traditional stack discipline, and are destroyed when the object returns. They correspond to variables designated as **Local**, **Var**, or **Recursive** in other languages. ("E" and "ThisTemp" are temporary variables.) When a class is defined, the names of these three types of variables are declared. The name of the superclass must also be declared when the class is defined.

C. Instances

Instances are objects that differ from classes in only one way: there is no code associated with instances, whereas there must be code associated with classes. This distinction is not necessitated by the logical framework developed so far, but occurs because of the way that classes are defined. The restriction that results is that instances cannot create subclasses or subinstances. Therefore, instances appear as leaf nodes in the hierarchy produced by the subclass-superclass relation. (In Fig. 1, "Six" and "Seven" are instances, each containing values for the instance variables declared in "Five.")

Instances correspond to values of variables, where the type of the variable is the superclass of the instance. Hence the instance 3 has as a superclass the class INTEGER. The name associated with the instance corresponds to the variable itself.

Since instances do not have any code associated with them, they also do not have either class variables or temporary variables associated with them. Instances contain only the values of instance variables declared in the superclass of the instance.

Most of the statements expressed in the remainder of this section apply to all objects; where there exists differences for classes and instances, the differences will be noted.

D. Execution Semantics and the Binding Mechanism

The control flow is tied to the object hierarchy and allows procedures contained in a class to be shared by its subclasses (the sharing mechanism will be detailed in the next section). When an object is sent a message, the code for that object is invoked. If there is no code associated with the object (i.e., the object is an instance), or if the class does not recognize the message, then the superclass of the object is sent the same message (called *forwarding* the message). This process continues until some class recognizes the message, or the class OBJECT is invoked (OBJECT recognizes every message and responds with some default reply). A class recognizes a message by *returning* a reply. As an example, if the instance "Seven" (in Fig. 1) were sent a message, that message would be forwarded to the class "Five." If "Five" did not recognize the message, it would be forwarded again.

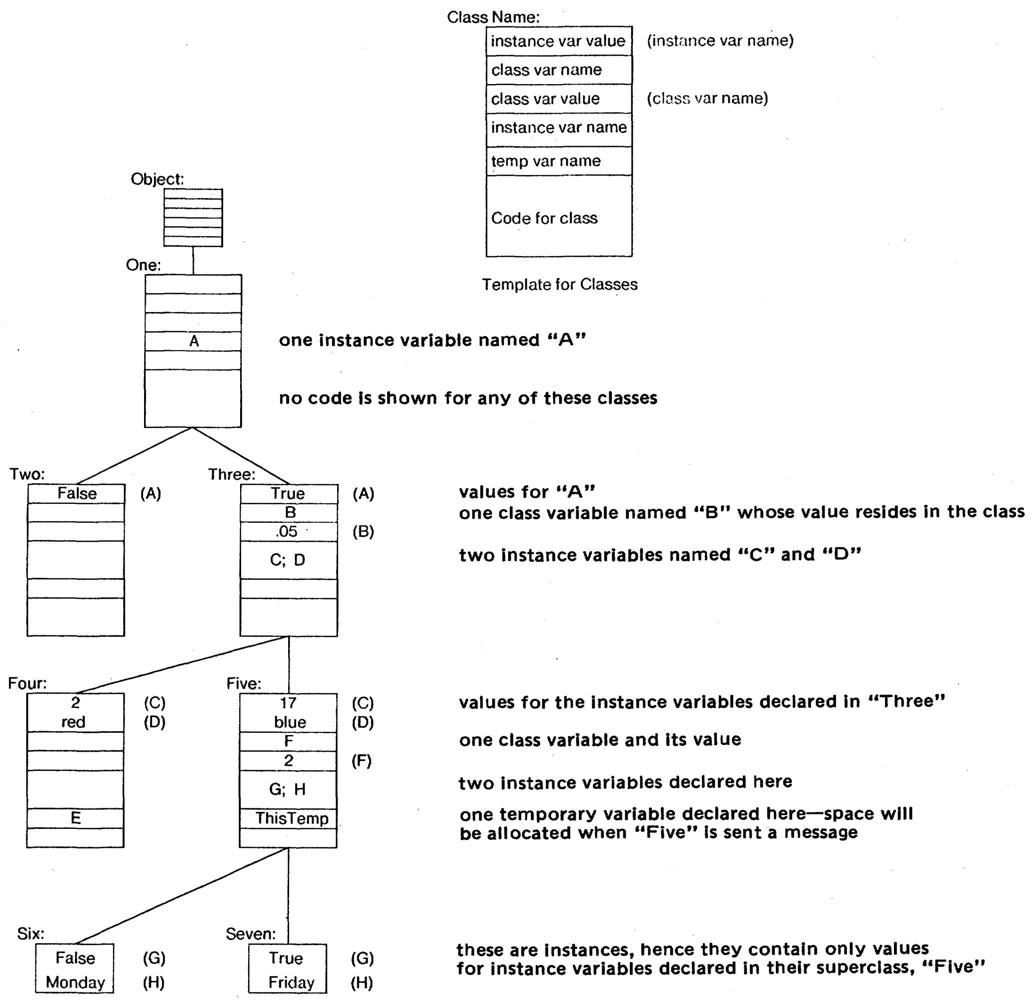

Class Name:
instance var value
(instance var name)
class var name
class var value
(class var name)
instance var name
temp var name
Code for class
Template for Classes
Object:
One:
A
one instance variable named "A"
no code is shown for any of these classes
Two:
False
(A)
Three:
True
B
.05
C; D
(A)
(B)
values for "A"
one class variable named "B" whose value resides in the class
two instance variables named "C" and "D"
Four:
2
red
E
(C)
(D)
Five:
17
blue
F
2
G; H
ThisTemp
(C)
(D)
(F)
values for the instance variables declared in "Three"
one class variable and its value
two instance variables declared here
one temporary variable declared here—space will be allocated when "Five" is sent a message
Six:
False
Monday
(G)
(H)
Seven:
True
Friday
(G)
(H)
these are instances, hence they contain only values for instance variables declared in their superclass, "Five"

Fig. 1. A class hierarchy with the data structures shown (the arcs represent the subclass–superclass relation).

The binding mechanism is also tied to the object hierarchy and allows data structures to be distributed. The binding mechanism is a combination of static binding (for class and temporary variables) and dynamic binding (for instance variables). When a class is defined, the names of the class, instance, and temporary variables are given, as well as the code to be associated with the class. The binding mechanism for class variables appearing in the class code simply binds the name to the value that resides in the class. Temporary variables are bound to the storage allocated at the time of invocation. The binding of temporary variables remains in effect until the object returns a reply, at that time the storage is recovered and the binding broken.

The binding mechanism for instance variables is more complicated due to the possible forwarding of a message. If a message is being forwarded from a subclass or instance, then the instance variables are bound to the values which reside in that subclass or instance. If the class was the class that was originally sent the message, then there are no values to bind to the instance variables, and an error occurs if they are referenced. For example, if "Seven" (in Fig. 1) were sent a message which was forwarded to "Five," then "Five" would have access to the instance variable "G" bound to the value residing in "Seven." However, if "Five" is sent a message directly, there is no value to bind to "G." Cola provides ways to determine if the current class was the one originally sent the message, and to name the class that *was* sent the original message. It is thus possible to associate different procedures with messages sent to a class and messages sent to a subclass of the class (analogous to *triggers* and *traps*, respectively, in KRL [4].)

The binding mechanism restricts the set of variables that may be referenced by the code in any given class. Since definitions must proceed in a top-down fashion (the superclass must be specified in the definition of a class), it makes no sense for a class to refer to a variable declared in one of its subclasses. Similarly, variables declared in a separate branch of the hierarchy are also inaccessible. To access a variable that is defined in a superclass, the class must send a message to its superclass requesting the value of the variable; this message is forwarded up to the appropriate superclass, which responds with the value of the desired variable. This process also applies to information contained in classes that are above the object originally sent the message, yet below the class currently dealing with the message, since a request for that information can be sent by a class to the class that was originally sent the message.

E. Automatic Inheritance and Shadowing

One of the ramifications of the execution semantics and the data structuring is the *automatic inheritance* by a class of the ability to respond to all the messages that the successive superclasses of the class can respond to.

As an example, the class CAPA responds to the message 'is ?'

by returning the STRING 'capability.' When the subclass CATALOGUE is defined, it automatically inherits the ability to respond to the message 'is ?' by simply forwarding the message up the CAPA. The subclasses FILE and TERMINAL also inherit this ability, as well as all the other operations defined in CAPA (and in the superclasses of CAPA, including OBJECT).

The mechanism of automatic inheritance is very useful. Simula [3] applied a form of automatic inheritance to build up an entire sublanguage suited to the construction of simulation programs. Most knowledge representation systems incorporate the concepts of *prototypes* (i.e., classes), *entities* (subclasses and instances), and *property inheritance* [47]. KRL [4], for example, includes a subclassing mechanism with automatic inheritance through the use of *perspectives*. However, sometimes this mechanism is not desired, as in the case of a subclass that needs to respond differently to a message than its superclass. In Cola, one can override automatic inheritance through the use of *shadowing*.

To illustrate shadowing, suppose that CATALOGUES were to respond to the message 'is ?' with the STRING 'catalogue' rather than 'capability' (the latter is correct, but the former is more precise). One way to do this is to have CATALOGUE *itself* respond to 'is ?' *without* forwarding the message up to CAPA. This enables CAPA to contain the commonality of its subclasses, and enables any subclass to respond differently to any particular message if it sees fit to do so. A second example of shadowing is the handling of messages containing 'vacate'. Most subclasses of CAPA do not recognize such messages, resulting in these messages being forwarded up to CAPA. However, CATALOGUE responds to 'vacate' *itself*, effecting a separate system call, since a different semantics is associated with this message when it is sent to a CATALOGUE.

The general mechanism can be summarized as follows: each class represents static knowledge (in the form of class and instance variables) and dynamic knowledge (in the form of the ability to reply to certain messages). Subclasses and instances automatically inherit the knowledge that is found higher in the hierarchical tree, and augment this knowledge with further knowledge. Through the use of shadowing, it is possible for a subclass to respond differently to a particular message than its superclass would have, thus making possible the expression of exceptions without nullifying the knowledge that exists higher in the tree.

IV. Conclusions

Essentially all of Cola has been implemented. The interpreter is written in Bliss/11 [49], with Cola code augmenting the low level objects. The static and dynamic structures are very similar to Smalltalk-76 [19], even though they were designed independently. It should be emphasized that Cola is an experimental system and very little effort has gone into tuning the implementation for efficiency.

There are several conclusions to be drawn from this effort. A useful correspondence between the entities supported by the command language (Cola objects) and those supplied by the operating system (Hydra capabilities) has been achieved. This correspondence is indeed "natural," in that there exist facilities in the language that support typed objects and the notion of operations (messages) that may be performed on these objects just as the operating system does. It is argued that the Cola paradigm can be successfully incorporated into a command language for a conventional operating system, although this premise has not been demonstrated concretely with an implementation.

Arguments for the message passing (versus procedure call) mechanism are less conclusive. At a basic level, this issue does not apply, since a duality exists between message-oriented languages and procedure-oriented languages just as it does in operating systems [29], [36]. As an example, the command

A print

in Cola (which sends the object "A" the message *print*, causing "A" to print a representation of itself on the terminal) is equivalent to the statement

print(A)

in the CL, with the instance (in this case, "A") passed as a parameter. The entire message forwarding mechanism can be simulated in Simula, although it must be done explicitly using additional procedure calls.

The primary advantage of message passing is that it is simple and does not impose a strict grammar on the language. The latter is usually considered to be a disadvantage in general purpose languages, but is convenient for a casual user interacting with a system at a terminal instead of carefully composing his or her programs before typing them in. A secondary advantage of message passing is that the mechanism lends itself naturally to multiprocess(or) systems [52]. Although C.mmp, on which Cola runs, is a multiprocessor, this aspect was not dealt with in the design.

The primary disadvantage of message passing is that it is inefficient when implemented in the obvious fashion. This drawback is not as worrisome as might first be expected, for several reasons. In a command language, efficiency is not a primary concern, since, on the average, a relatively small number of statements are executed as a result of a command typed by the user [24]. It can be argued that a command language procedure that is unacceptably slow should be rewritten in one of the languages supported by the operating system [5]. In addition, the general message passing mechanism can be avoided most of the time in the interpreter (the semantics of the language would, of course, still be defined in terms of message-passing) by applying a few relatively simple transformations to the source before it is interpreted and by designing the interpreter so that it uses the local state to circumvent the message assembly mechanism except for special cases, primarily when an error occurs [23].

The object hierarchy, coupled with the message-passing and -forwarding mechanism, has been shown to have several useful attributes concerning the structuring of objects. Static knowledge, in the form of class and instance variables, is stored as high in the hierarchy as possible, eliminating redundancy at lower levels. Similarly, dynamic knowledge, encoded in the ability to respond to certain messages, is shared among many classes and instances. The mechanism allows flexibility in the placement of both procedures and data structures within the hierarchy.

One disadvantage is the restriction that the values of instance variables declared in a class must reside in each subclass of the class. This restriction can be removed by allowing more flexibility in the sharing of *names* of data structures within the object hierarchy. The names of class variables, for instance, are not shared at all, since both the names and the values reside in the class they were declared in. In Cola, the names (but not the values) of instance variables are shared by the immediate subclasses of the class they were declared in. In Simula, the names of instance variables are shared by *all* the subclasses. However, the optimal placement of the value (and the name) of an instance variable depends to a large extent on the semantics desired for the information contained in that variable (and the sharing of the name of the variable), and mechanisms for allowing more variability for the binding and storage of instance variables need to be developed.

There are several other areas where additional research is needed. It is not clear how one would incorporate multiple process(or) concepts into the language. Several possible alternatives seem likely, including utilizing the message-passing mechanism and/or allowing multiple objects to execute concurrently, but the ramifications these various schemes have on the language have not been investigated at all. More work is necessary to make the paradigm of objects communicating via messages a viable one in terms of efficiency (although much has been done in this area by the Learning Research Group at Xerox PARC [19]). Lastly, the techniques used to mirror the abstractions provided by Hydra in the command language should be applied to other operating systems, including more conventional ones, in order to assess the applicability of these concepts.

Acknowledgment

The CL [37] provided valuable insight into some of the inherent advantages and disadvantages of a command language under Hydra. Most of the underlying language concepts can be traced directly back to Smalltalk-72 [13]. The concept of classes and instances originated with Simula [3], and some of the concepts of the class hierarchy (especially the concept of shadowing) were developed independently by S. Fahlman [11] and were hinted at in the Pygmalion system [41].

References

[1] G. Almes and G. Robertson, "An extensible file system for Hydra," Dep. Comput. Sci., Carnegie-Mellon Univ., Feb. 1978, available as CMU-CS-78-102.

[2] P. Alsberg, "OSL/2: An operating system language," Ph.D. dissertation, Cen. for Advanced Comput., Univ. Illinois, Urbana-Champaign, 1971.

[3] G. M. Birtwistle, O.-J. Dahl, B. Myhrtag, and K. Nygaard, *Simula Begin.* Philadelphia, PA: Auerbach, 1973.

[4] D. G. Bobrow and T. Winograd, "An overview of KRL: A knowledge representation language," Xerox PARC, July 1976, available as CSL-76-4.

[5] S. R. Bourne, "The Unix shell," *Bell Syst. Tech. J.*, vol. 57, part 2, pp. 1971-1990, July-Aug. 1978.

[6] R. F. Brunt and D. E. Tuffs, "A user-oriented approach to control languages," *Software-Practice and Experience*, vol. 6, pp. 93-108, 1976.

[7] E. Cohen and D. Jefferson, "Protection in the Hydra Operating System," in *Proc. 5th Symp. Operating Syst. Principles*, ACM, Austin, TX, Nov. 1975, pp. 141-160.

[8] E. Cohen *et al.*, "Hydra kernel reference manual," Dep. Comput. Sci., Carnegie-Mellon Univ., Nov. 1976.

[9] R. M. Cowan, "Burroughs B6700/B7700 work flow language," in *Command Languages*, C. Unger, Ed. North-Holland, 1975, pp. 153-171.

[10] J. R. Ellis, "A LISP shell," *SIGPlan Notices*, vol. 15, pp. 24-34, May 1980.

[11] S. E. Fahlman, *Netl: A System for Representing and Using Real-World Knowledge.* Cambridge, MA: M.I.T. Press, 1979.

[12] P. H. Feiler and R. Medina-Mora, "An incremental programming environment," Dep. Comput. Sci., Carnegie-Mellon Univ., Apr. 1980, available as CMU-CS-80-126.

[13] A. Goldberg and A. C. Kay, Eds., *Smalltalk-72 Instruction Manual.* Palo Alto, CA: Xerox PARC, 1978.

[14] A. Goldberg and D. Robson, "A metaphor for user interface design," in *Proc. 12th Hawaii Int. Conf. Syst. Sci.*, 1979, pp. 148-157.

[15] I. Grief and C. Hewitt, "Actor semantics of PLANNER-73," in *Proc. 2nd Conf. Principle of Programming Languages*, Jan. 1975.

[16] A. N. Habermann, "An overview of the Gandalf project," Dep. Comput. Sci., Carnegie-Mellon Univ., CMU Comput. Sci. Res. Rev. 1978-1979, 1980.

[17] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intell.*, vol. 8, pp.323-364, 1977.

[18] *IBM Virtual Machine Facility/370: CMS Command and Macro Reference Manual*, 1980. Order GC20-1818.

[19] D. Ingalls, "The Smalltalk-76 programming system: Design and implementation," in *Proc. 5th Conf. Principles of Programming Languages*, ACM, Jan. 1978, pp. 9-16.

[20] A. K. Jones, "The narrowing gap between language systems and operating systems," in *Proc. IFIP Conf.*, 1977.

[21] A. K. Jones, R. Chansler, Jr., I. Durham, K. Schwans and S. Vegdahl, "StarOS: A multiprocessor operating system for the support of task forces," in *Proc. 7th Symp. Oper. Syst. Principles*, Pacific Grove, CA, Dec. 1979, pp. 117-127.

[22] A. C. Kay and A. Goldberg, "Personal dynamic media," *IEEE Computer*, vol. 10, pp. 31-41, Mar. 1977.

[23] A. C. Kay, personal communication.

[24] B. W. Kernighan and J. R. Mashey, "The Unix programming environment," *Sofware-Practice and Experience*, vol. 9, pp. 1-15, 1979.

[25] B. W. Lampson, J. J. Horning, R. L. Lampson, J. G. Mitchell, and G. L. Popek, "Report on the programming language Euclid," *SIGPLAN Notices*, vol. 12, Feb. 1977.

[26] B. W. Lampson and R. Sproull, "An open operating system for a single-user machine," in *Proc. 7th Symp. Operating Syst. Principles*, ACM, Pacific Grove, CA, Dec. 1979, pp. 98-105.

[27] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," in *Proc. 2nd Int. Symp. Operating Sys.*, IRIA, Oct. 1979, reprinted in *Oper. Syst. Rev.*, vol. 13, pp. 3-19, Apr. 1979.

[28] S. Lauesen, "Program control of operating systems," *BIT*, vol. 13, pp. 323-337, 1973.

[29] J. Levine, "Why a Lisp-based command language?," *SIGPLAN Notices*, vol. 15, pp. 49-53, May 1980.

[30] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Commun. Ass. Comput. Mach.*, vol. 8, pp. 564-576, Aug. 1977.

[31] G. H. Mealy, "The functional structure of OS/360," *IBM Syst. J.*, vol. 5, no. 2, 1966.

[32] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, P. Winston, Ed. New York: McGraw-Hill, 1975, pp. 211-277.

[33] M. D. Ostreicher, M. J. Bailey, and J. I. Strauss, "GEORGE 3-A general purpose timesharing and operating system," *Commun. Ass. Comput. Mach.*, vol. 10, pp. 685-693, Nov. 1967.

[34] J. Ousterhout, D. Scelza, and P. Sindu. "Medusa: An experiment in distributed operating system structure," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 92-104, Feb. 1980.

[35] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. "Pilot: An operating system for a personal computer," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 80-91, Feb. 1980.

[36] L. G. Reid, "Control and communication in programmed systems," Ph.D., dessertation, Carnegie-Mellon Univ., Sept. 1980, available as CMU-CS-80-142.

[37] A. Reiner and J. Newcomer, Eds. *The Hydra Users Manual.* Carnegie-Mellon Univ., Dep. Comput. Sci., 1977.

[38] E. Sandewall, "Programming in the interactive environment: The LISP experience." *Comput. Surveys*, vol. 10, pp. 35-72, Mar. 1978.

[39] M. Shaw, "The impact of abstraction concerns on modern pro-

gramming languages," *Proc. IEEE* vol. 68, Sept. 1980.

[40] J. F. Shoch, "An overview of the programming language Smalltalk-72," *SIGPLAN Notices*, vol. 14, pp. 64-73, Sept. 1979.

[41] D. Smith, "Pygmalion: A creative programming environment," Ph.D. dissertation, Stanford Artificial Intell. Lab., Stanford Dep. Comput. Sci., Stanford, CA, June 1975, available as STAN-CS-75-499.

[42] T. Teitelbaum, "The Cornell program synthesizer: A microcomputer inplementation of PL/CS," Cornell Univ., July, 1979, available as TR 79-370.

[43] W. Teitelman, *INTERLISP Reference Manual.* Xerox PARC, 1978.

[44] S. Treu, "Interactive command language design based on required mental work," *Int. J. Man-Machine Studies*, vol. 7, pp. 135-149, 1975,

[45] S. Warren and D. Abbe, "Rosetta Smalltalk: A conversational, extensible microcomputer language," in *Proc. 2nd Symp. Small Syst.*, ACM SIGPC, Dallas, TX, Oct. 1979.

[46] M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and Its Operating System.* New York: Elsevier North-Holland, 1979.

[47] T. Winograd, "Breaking the complexity barrier (again)," *SIGPLAN Notices*, vol. 10, pp. 13-30, Jan. 1975.

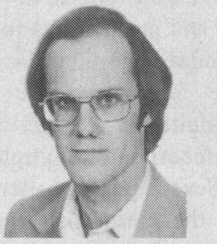
[48] W. A. Wulf and C. G. Bell, "C.mmp–A multi-mini-processor," in *Proc. 1972 Fall Joint Comput. Conf.*, pp. 765-777.

[49] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. D. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler.* New York: Elsevier North-Holland, 1975.

[50] W. A. Wulf, R. London, and M. Shaw, "Abstraction and verification in Alphard: Introduction to language and methodology," Dep. Comput. Sci., Carnegie-Mellon Univ., June, 1976.

[51] W. A. Wulf, R. Levin, and S. P. Harbison, *C.mmp/Hydra: An Experimental Computer System.* New York: McGraw-Hill, 1981.

[52] A. Yonezawa and C. Hewitt, "Modeling distributed systems," in *Proc. 5th Int. Joint Conf. Artificial Intell.*, ACM, MIT, Aug. 1977, pp. 370-376.



Richard Snodgrass (M'81) received the B.A. degree in physics from Carleton College, Northfield, MN, in 1977 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1982.

He is currently an Assistant Professor at the Department of Computer Science, University of North Carolina, Chapel Hill. His research interests include user interfaces and the application of database and artificial intelligence concepts to programming environments.

Mr. Snodgrass is a member of the IEEE Computer Society, the Association for Computing Machinery, and Sigma Xi.

A Symbol Table Abstraction to Implement Languages with Explicit Scope Control

ROBERT P. COOK AND THOMAS J. LEBLANC

Abstract—**We are concerned with languages in which the programmer has explicit control over the referencing environment of a name. Several modern programming languages, including Ada, Euclid, Mesa, and Modula, implement these control capabilities. This paper describes a simple technique which uses the traditional concepts of a hashed symbol table and lexical level to solve many of the symbol table implementation problems associated with explicit scope control. The primary advantage of this technique is that a single symbol table abstraction can be used to simply and efficiently solve most problems in scope control.**

Index Terms—**Lexical level, scope control, symbol table.**

Manuscript received January 10, 1981; revised August 3, 1982. This work was supported in part by the National Science Foundation under Grants MCS-7903947 and MCS-803341.

The authors are with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.

I. Introduction

THE classical scope rule for block structured languages is that an identifier is known in the block in which it is first declared and in all the enclosed blocks in which it is not redeclared. A block structure which maintains this rule is referred to as an *open* scope since the current referencing environment is automatically inherited by each new block. The opposite extreme, in which no names are inherited, is referred to as a *closed* scope.

Several modern programming languages, including Ada [9], Euclid [5], Mesa [7], and Modula [11], allow a user to explicitly control the scope of an identifier. Scope control can occur in many different forms. In Modula, the programmer may optionally augment each block heading with a "define" or "use" list of symbols. If neither list occurs, the scope is open; otherwise, the scope is closed, except for the listed