OBJECT ORIENTED MUMPS: A PROPOSED BLUEPRINT

Terry L. Wiechmann
Jerry E. Goodnough
Educational Systems Inc.
211 Vaughn Hill Road
Bolton MA.
(508) 779-2914

Abstract

Object Orientation is not a concept of the future - it's a paradigm that is being rapidly accepted as a **model** for the future.

Educational Systems Inc. has developed a fully functional Object Oriented (OO) programming system called EsiObjects[1] (pronounced 'easy objects'). The goals of this project were accomplished by extending the M language specification to include all necessary OO syntactical elements. ESI would like to share the results with the M community. This paper elaborates upon those areas of the M language and operating environment that were subject to enhancement.

Foreword

This paper assumes the reader has a good understanding of Object Oriented concepts.

The syntax described in this paper evolved as a result of extensive research and prototyping by ESI. It should not be the focus of attention as you read. Concentrate on the functionality since it's the most important aspect of OO.

Two general areas will be covered:

- EsiObjects Environment Overview
- EsiObjects Language Extensions

EsiObjects Environment Overview

EsiObjects is an Object Oriented Application Development Environment. It is built on, and extends, the MUMPS language (hereafter referred to as M). M is an ANSI Standard computer programming language that demonstrates powerful string handling and database capabilities.

Figure I illustrates the layered architecture of EsiObjects.

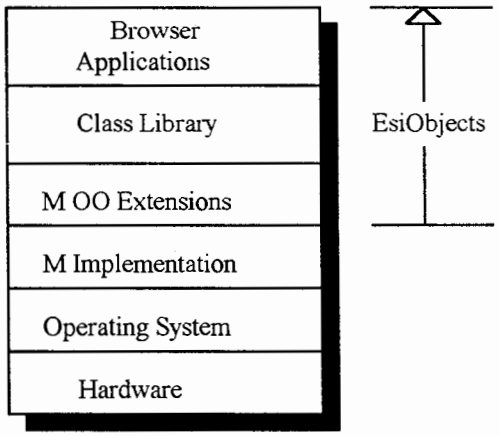

Figure 1: EsiObjects' Architectural View

The full EsiObjects layer consists of several components:

- **Object Oriented M Language Extensions** that are implemented through a language preprocessor and runtime modules.
- A complete **Class Library** of Foundation Classes that are used for application development.
- A complete **Application Development Environment** which includes all necessary browsers and utilities required to develop most applications.

Object Oriented M Language Extensions

The M language is a powerful string handling and database language. Fundamental requirements for object orientation have always been an integral part of M. It is a weakly typed language and implements late binding of values to names. It provides comprehensive linguistic support for the indirection concept which is essential to writing generic code.

Additionally, M provides for *persistence*, the ability to retain objects beyond the lifetime of a job. Persistent objects are implemented using the concept of *global arrays*.

Features missing from the M language for *object* support are:

- **Object Definition and Typing** - The ability to create a self-contained environment for each object and protect its contents by enforcing *encapsulation*.
- **Inheritance Mechanisms** - Control and optimize hierarchical searches for *variables*, *methods*, *labels* and *parameters*.
- **Variable Scoping** - Determines to what extent a variable is visible by the owning object and other objects.
- **Special Variables and Functions** - Special Variables contain the necessary pointers to objects required for quick access and flexibility of coding. Functions return environment values very quickly.
- **Messaging** - It's a mechanism by which objects communicate with one another. It forms a weak binding between objects.

Building on the fundamental features of M, ESI extended the M language to include those features needed to support Object Orientation. This was accomplished by writing a *preprocessor*. The preprocessor implements the following:

- Parses ANSI Standard M code and identifies syntax errors.
- Expands Object Oriented extensions to Standard M code.
- Optimizes hierarchical searches.
- Compiles runtime messaging system.
- Provides runtime support and interpretation.

In keeping with the nature of compilation in all M systems, calls to the preprocessor are hidden from the programmer.

Class Library and Applications

In Figure 1, the Class Library and Browser Applications are application levels. Once the language extensions are in place, the programmer has the necessary linguistic tools to develop a full Class Library. In a truly comprehensive OO system, the Class Library will contain an extensive array of Class definitions. These definitional components are then used to develop full applications.

The Browser Applications within the EsiObjects environment form a full Object Oriented development environment based upon a windowing environment. However, these layers can contain more primitive development tools.

EsiObjects Language Extensions

This section outlines the syntactic extensions to the M language required to implement the Object Oriented paradigm. These extensions are **not** ANSI Standard. They are the result of extensive research and prototyping by ESI.

The extensions can be organized into six groups:

- Object Definition
- Types of Objects
- Messaging
- Label Inheritance
- Variable Scoping and Inheritance
- Special Variables and Functions

Object Definition

As outlined previously, the M language contains a good foundation for object orientation. The fundamental linguistic capabilities of the language provides a base for implementing OO structures and functions. Fundamental to all OO is the concept of an object - its definition and contents.

An **object** is a unique, self-contained entity that shares much in common with the standard M partition. It contains:

- Identity
- State
- Behavior

Identity is the attribute that makes one object unique from another. Within the computer system it is identified by a unique Object Identifier (OID). When a new object is created, it is assigned an OID.

State is the sum of all static and dynamic properties of the object. These properties are stored as values under symbol names within the object. Therefore, a significant part of an object definition is devoted to a symbol table. EsiObjects symbol names conform to the Standard M specification. However, they may be up to 31 characters in length. The M portability requirements for symbol names should be extended to permit more meaningful names.

Behavior is how an object responds to a state change as a result of receiving a message to do something. Behavior is contained in blocks of M code that are an integral part of the object's definitional component - a Class. These blocks of code are known as *methods*. Methods are executed as a result of the object receiving a message to do something. A method name maps to an M entry reference which is stored in the M routine directory.

Figure 2 illustrates an object and its components.

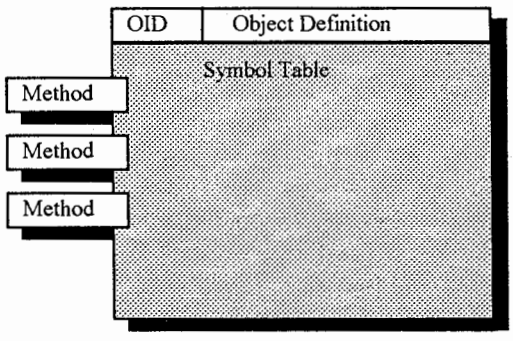

Figure 2: Illustration of an Object Environment

The header area containing OID and Object Definition are internal to the object. The **OID** gives an object its identity. As stated previously, it is a unique identifier. Uniqueness of course is a complex issue when dealing with concurrent and distributed systems. The **Object Definition** area contains overhead information about the object, such as pointers, security codes, etc.

An OID is quite simply a partial symbol name that can be used in subscript indirection. The OID is an internal handle on the object and can be used by the M programmer as an object pointer.

Methods are terminating points for messages. A method contains the code called when an object receives a given message. They are the only mechanism that can access the object's symbol table and manipulate variables in the table. Protecting an object's data is called *encapsulation*. The sum of all methods gives an object its total behavior.

Having illustrated and explained the component parts of an object, it should be apparent to the astute observer that there is a great deal of overlap between an object concept and an M partition. The partition exists as a definition in the M implementations code. When a user signs onto the system, the system stamps out an instance of the partition called a Job. A job is given a unique identifier which resides in the special variable $JOB. Within the partition resides a symbol table and a pointer to the UCI or Namespace that contains the global data and routines.

This concept must be slightly enhanced and made available to the programmer at a finer degree of granularity. In other words, the programmer must have control over creating, manipulating and deleting object environments within the context of an M job. This obviously can be viewed as a recursive definition of the object concept.

The M array structure could be used to implement an object environment. Unfortunately, this raises object environmental issues to the M programmer level. The application code would have to enforce encapsulation among other maintenance tasks that are better delegated to the operating environment.

Types of Objects

Within any object oriented system, two types of objects exist:

- Instance
- Class

An **instance** can be viewed as a real world object. It has identity, state and behavior.
Let's use human beings as an example of the two types of objects. Viewing all human beings, we notice that we all

share common characteristics and behavior. In a computer model of an object oriented system, rather than storing these common characteristics redundantly, as is done in humans, they are stored in a definitional object called a *class object*. The class object contains the information and methods necessary for creating an instance of the class, for example, a specific human. Since the common characteristics of the instance are stored in the class object, a pointer within the instance's overhead area is used to keep track of the class of objects it belongs to. This pointer forms one link in an ***inheritance path.***

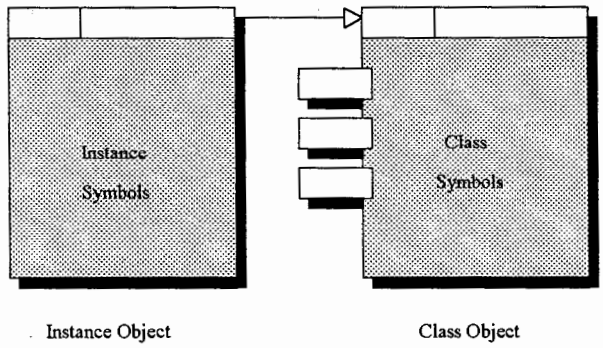

Figure 3: Instance and Class Types of Objects

Figure 3 illustrates an instance and a class object linked together by a pointer. The pointer is known as an inheritance pointer. It is used to link the specific object to the more general object that contains its definitional symbols and methods. It is then used by the system to make those common characteristics and methods available to the instance. It provides an inheritance path for all instances to the class object that created the instance.

Notice that the methods are attached to the class object. Methods are always stored with the class and never with the instance. However, symbols are stored at all levels of the hierarchy and are appropriately named. **Instance symbols** are stored at the instance level and **class symbols** are stored at the class level.

Carrying this concept to an extreme, Figure 4 illustrates what a generic hierarchy would look like.

A class hierarchy exists in most object-oriented systems. It is a further separation of common characteristics and methods the higher up the definitional structure, the more general the characteristics and methods become. All classes and instances below inherit the characteristics and methods. The further down the tree, the more specific the objects. An instance is the most specific object.

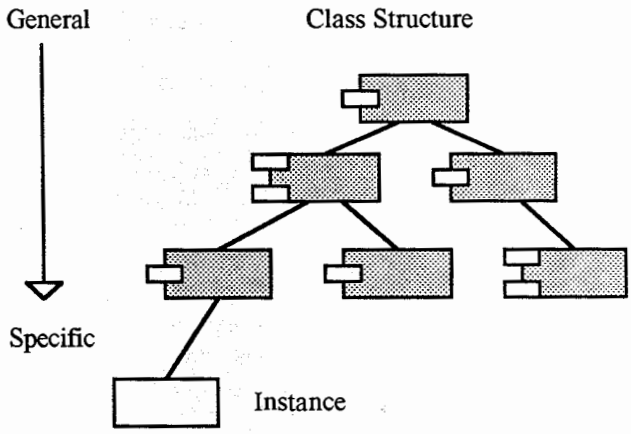

Figure 4: A General Inheritance Hierarchy

Class objects (shaded boxes) are linked together to form an *inheritance* tree. Since class objects contain common methods and symbols, the concept of inheritance is used to make these entities available to any object further down the inheritance tree. For example, if an instance object receives a message to execute a method, each class in the instance's inheritance path is searched for the method. When it is found, the method is executed. In reality, actual searching does not take place, searches are compiled out.

Messaging

Objects communicate with each other by messaging. If object A wants to know something about object B, it must send B a message. The message is actually a method and object selection mechanism that carries along with it a set of parameters that contains specific information about the message. The ABCL/1[2] message syntax heavily influenced the design of EsiObjects' message syntax.

Messaging forms a weak binding between two objects. It respects the encapsulation of other objects.
A special EsiObjects syntax exists for a message as follows:

[Method.Object.KeywordList](ParameterList)

Messages always produce a value much like an extrinsic function within the M language.

The **Method** and **Object** within the square brackets, separated by the period, identifies a specific method to be executed within the specified object. Functionally, it acts like an M entryref, where the label and routine name identify an entry point into a routine. In this case, the method object construct identifies a delivery point for the message in an object environment. To use a postal letter analogy, the entire message is analogous to the envelope and the method.object portion of the message is the address label.

Method and Object can be prefixed with an underscore character to tell the compiler that the method and object are in the Class structure. For example, the message **[_New._WindowObject](Name="MyWindow")** would create a new object **MyWindow** using the definitional class **WindowObject**.

The **KeywordList** is a list of keywords that are separated by commas. These keywords apply to the message only. They are not meaningful to the recipient of the message. Message keywords can be used for a variety of functions such as specifying concurrency parameters, activating the debugger, etc.

The **ParameterList** is a list of parameters separated by commas. Each parameter may be a *Keyword* or *Keyword Alias* that maps to another keyword. Positional parameters were initially used with keywords in early prototypes of EsiObjects, however, they were dropped. Keywords are much more flexible (non-positional) although they must be defined as a part of the method prior to compilation. This is an implementation issue.

Keywords and their aliases are created, by the programmer, through a parameter browser (or utility) for each method within a class. The programmer then uses them in the parameter list. When the method is executed, the keywords and values are instantiated into parameter variables which are described in the next section.

Parameters contain the specifics of the message that are meaningful to the receiving method. In terms of the postal letter analogy, the parameters are analogous to the contents of the envelope.

Figure 5 illustrates the concept of messaging between objects. In this example, the message PrintReport is being sent to the Printer object. Specifically, Printer is being told to print INVOICES by specifying the keyword Report.

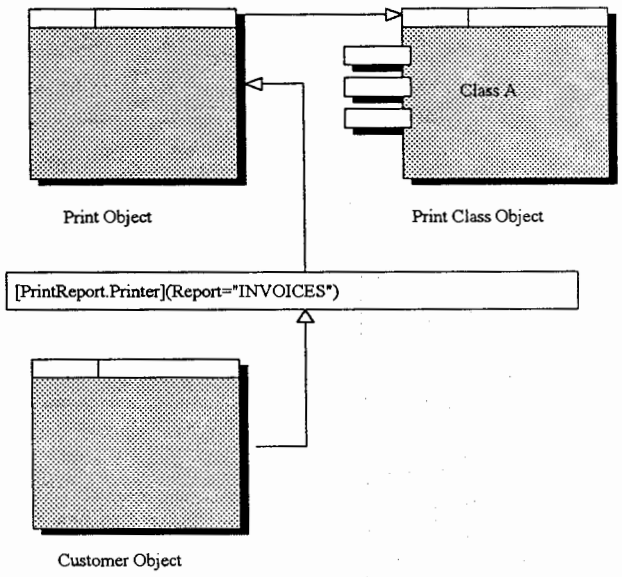

Figure 5: Messaging between Objects

Method Inheritance

Methods, as explained, are message names. Methods are attached to classes within the class library hierarchy and are mapped to a block of M code (routine). The concept of an M routine is simply viewed as a physical storage mechanism. The logical concept of a method maps to that physical entity.

Within any object-oriented system that supports inheritance, the concept of *polymorphism* is a very powerful feature. One form of polymorphism permits the same method names to be used throughout the class library (one per class). Within the inheritance path of a class, the first class with the message's method name will be executed. Objects of different classes implement the same methods differently, behaving in a manner appropriate to their class.

Figure 6 illustrates the concept. A class inheritance hierarchy is shown for a Parts - Tires relationship. An instance of the Tires exists named X26. Notice that when X26 receives the message

[AddElement.X26](Quantity=10)

the system will search the inheritance path for the method name AddElement. The first method found with this name will be executed. In the case illustrated, AddElement method of Tire's class will be executed first.

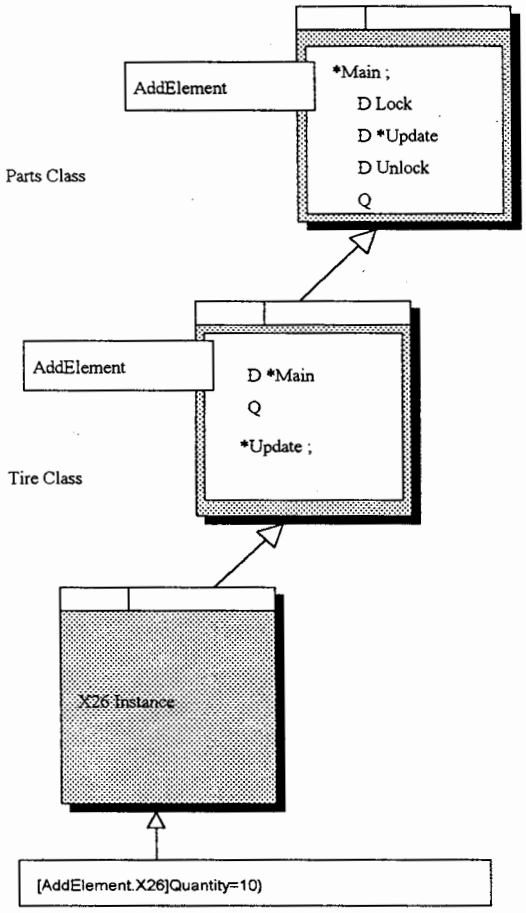

Figure 6: Example of Label Inheritance

Assume that AddElement of Tires did not exist. In this case, AddElement of Parts would be executed. Inserting the method AddElement at the Tire's class *overrides* the AddElement above it. It is this capability that permits code to be specialized to a particular application vendor or user site's needs or any other means of artificial separation. This same capability is extended to symbol names which will be discussed in the Variable Scoping and Inheritance section.

Label Inheritance

If we can control the generalization and specialization of whole methods by overriding names, it makes sense to extend this capability to labels within the body of a method. EsiObjects does this by letting the programmer declare a label as private or public within the inheritance path.

Again, look at Figure 6. Notice that within the AddElement method of Tires, a label **Update** is preceded by an asterisk (*). This tells the preprocessor that this label is to become public to any method within the class's inheritance path. Labels that are not preceded by the * are treated as normal M labels and are private to the method.

Looking at the Parts class, notice the **D *Update** command. The * informs the preprocessor to compile in a lookup for the public label Update.

To understand how this works, let's walk through the example in Figure 6.

1. Starting at the bottom of the class structure is an instance of the class Tires called X26 which is a particular type of tire.

2. A message is sent to X26 which tells it to add the value 10 to the variable Quantity.

3. Upon receipt of the message, the inheritance tree is searched for the method AddElement. (An actual search may not take place since all method inheritance searches should eliminated by the compiler). The method AddElement at the class Tires is found and executed.

4. The first line of code is **D *Main**. The asterisk indicates a public label. Execution of this DO argument forces an inheritance lookup **within all methods named AddElement only**.

5. The Main method will be found in the Parts class and executed. The first line of code contains a **D Lock.** The lock label is private. A normal M label search will take place within the body of the routine.

6. Next, the **D *Update** will be executed. Again, it is a public label. An inheritance lookup will be performed within the context of all known methods named AddElement that are in the inheritance path.

7. The Update method will be found in the Tires class and executed. How the parameter **Quantity=10** is passed on to the Update method is not shown. This will be covered in the next section on **Variable Scoping and Inheritance**.

8. Once the Update subroutine finishes, control is passed back to the Main routine in Parts where the **D Unlock** is executed. Of course, Unlock is private and will reside within Main.

Notice what the inheritance structure and lookup mechanism does for you. Executing different blocks of code from a main routine no longer requires conditionals to determine which labels to execute and eliminates all extraneous branching conditions. This is a subtle, but powerful feature. The structure of the routines is built into the class structure. It's a behavioral structure based upon the structure of the class abstraction levels. Correctly designing the class structure is very important.

Variable Scoping and Inheritance

By definition, a system that enforces encapsulation, as EsiObjects does, requires an explicit method of scoping variables.

In non-object oriented M systems, variables are scoped at three simple levels. First, local variables are confined to the M job's context. This simply means that other jobs cannot access these variables. Secondly, within this context, the scope of all variables is across all execution domains unless explicitly confined to a specific domain by the NEW command. Third, the scope of a variable can be extended across a class of users by using globals.

An Object Oriented M system must implement a completely different form of scoping. The object definition and the concept of encapsulation dictate scoping.

First, locals and globals are **not** used as a scoping mechanism. Globals are used to implement *persistence*. An object is persistent if it lives beyond the lifetime of a job. Obviously, local variables are then referred to as non-persistent. Therefore, the local variables are used to temporarily store an object and its variables - where an M global is used to store an object until it is explicitly deleted.

Variable scoping refers to accessibility to one object's variables by another. Notice the Symbol Table in the object diagram of Figure 2. Only that object can refer directly to these symbols. Other objects can only obtain their values by requesting them through a direct method call to the object.

Variables can be confined to several levels in EsiObjects:

- Temporary
- Instance
- Class
- Parameter
- Universal
- Named Pool

Figure 7 illustrates the concept of variable scoping in an object environment. Temporary, Parameter, Instance, Class and Universal scopes are illustrated (in italics) relative to the objects.

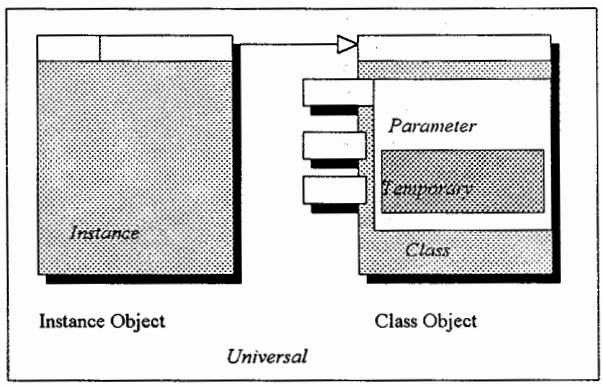

Figure 7: Scoping of Variables

Because EsiObjects added the concept of variable scoping to the M language, a syntactic mechanism had to be added to implement the concept.

According to portability rules of the M language, a symbol must start with an alphabetic or percent (%) sign followed by 1-7 alphanumeric characters. For example:

```
DATE="09/01/92"
%system="OMEGA"
```

In the early releases of the M language, symbols with the leading % character were conventionally reserved for system utilities. It implied a form of scoping that was enforced only for global symbols by some M implementations.

EsiObjects extended the % symbol syntax to the following:

SCOPE%NAME(S1,S2,...Sn)

where:

NAME is the symbol name and **S1, S2,...Sn** are the subscripts.

SCOPE is case sensitive and identifies the scope of the variable. It can be:

T	Temporary
I	Instance
C	Class
P	Parameter
U	Universal
N	Named pool

Table 1: Variable Scoping Codes

Scope permits the programmer to visually determine the scope of a variable in the source code. Additionally, the length of the variable has been extended to 31 characters. For example:

S I%CustomerName="Doe, John"

This example creates an instance variable whose scope is confined to the currently active object.

Temporary variables are confined to the block of code between the first occurrence of the variable and the next explicit or implicit QUIT command, but not into the method body of any sub-message.

Instance variables are confined to the scope of the object. They are accessible by all the object's methods but hidden from all other objects.

Class variables are confined to a class object. A class object contains the definitional components required to create other objects known as instances.

Parameter variables are identical to temporary variables in most respects. They are created by the compiler from a parameter *keyword*. Keywords are defined for a method like a temporary, its scope is confined to the execution of the method - from the entry point to the QUIT (implicit or explicit) that terminates the method's execution. If the method sends a message to another object, the parameter variables do not extend into the body of the recipient of the message.

Referring back to Figure 6, the parameter on the message

[AddElement.X26](Quantity=10)

contains the keyword Quantity which carries the value 10. Quantity must be defined as a keyword belonging to the method AddElement.

At runtime, Quantity becomes a parameter variable which is treated like a temporary variable.

Universal variables are available across the scope of all objects. They are universally accessible by all methods of all object types - instance or class. Universal variable use is discouraged. They violate the concept of encapsulation which should be guarded against in an object oriented system.

Named Pool variables are a way of sharing variables and values between objects. A named pool is really an object. Remember that an object has its own symbol table that may contain symbols. Multiple objects may have access to a named pool. Additionally, Named pools may be linked into hierarchies such that the variables are then available through the normal inheritance paths.

This construct is a very powerful feature of the EsiObjects system. It permits the application programmer to create inheritance driven symbol tables that are organized by a particular relationship. For example, the Class relationship is often referred to as a *kind-of* relationship. Other relationships exist such as a *part-of* relationship. The point is, the programmer must have this generic capability to build any inheritance relationship required by the application. Named Pools fulfill that requirement.

Special Variables and Functions

Special variables and functions have been added to EsiObjects. Special variables hold important values the programmer will need to access. Most special variables hold pointers to commonly accessed objects. Some of the most common special variables available in the

EsiObjects' system are:

$ZSELF	OID of the object's caller.
$ZSUPER	OID of the object's super-class. That is, if the current object is an instance of a class, $ZSUPER points to its super-class
$ZLAST	contains the OID of the last object a message was sent to. It defaults to $ZSELF.
$ZCALLER	OID of sender of the current message.
$ZRETURN	Contains the return value of a message.

These are only a few of the special variables implemented in the EsiObjects' system.

A few functions have been added to selectively access values available to the programmer. Some of the variables are listed below:

$ZEXISTS(expr)	returns a 1 if the object specified by the parameter exists. The expr is an expression that must evaluate to an OID.
$ZGETPARM(expr)	returns the value of the keyword specified in expr. The expr must evaluate to a keyword.

Not all of the special variables and functions have been listed. Additionally, it should be understood that many of them can be written as extrinsic functions.

Summary

The ANSI Standard M language contains a natural foundation for Object Orientation. At its current level of definition, object based applications can be built. However, to implement a full object oriented environment, the following enhancements must be made to the language and operating environment:

- object environment definition
- enforcement of encapsulation
- fast inheritance searching of methods and symbols
- full messaging capabilities

[1]Wiechmann, Terry and Goodnough, Jerry, **EsiObjects: An Object Oriented Application Development Environment**, MUMPS Computing, Sept. 1992

[2]A. Yonezawa, Shibayama E., T. Takada, Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in "Object-Oriented Concurrent Programming" edited by A. Yonezawa and M. Tokoro, MIT Press, 1987.