Object Structure in the Emerald System

Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy

Department of Computer Science, FR-35
University of Washington,
Seattle, Washington 98195

Emerald is an object-based language for the construction of distributed applications. The principal features of Emerald include a uniform object model appropriate for programming both private local objects and shared remote objects, and a type system that permits multiple user-defined and compiler-defined implementations. Emerald objects are fully mobile and can move from node to node within the network, even during an invocation. This paper discusses the structure, programming, and implementation of Emerald objects, and Emerald's use of abstract types.

1. Introduction

Distributed systems are inherently more complex to program than non-distributed systems. In an effort to reduce this complexity, much recent work has focused on tools that assist in the construction and programming of distributed systems and applications. Examples include message-based operating systems such as Accent [22] and V [12], remote procedure call facilities such as Xerox RPC [5], and languages such as Argus [20] and the Eden Programming Language (EPL) [7].

These three approaches to distribution represent a succession of abstractions. Message-based systems require the programmer to deal with the details of locating message targets, packaging messages, and with asynchronous communication. Remote procedure call hides the details of packaging and process control and presents the programmer with a standard procedure call paradigm; however the programmer is responsible for locating the target of the call. Object-based languages such as Argus and EPL provide location-independent invocation of distributed objects; location is implicit. As one moves along this spectrum from message-based systems to distributed languages, flexibility and control are traded for simplicity and ease of programming.

We have designed and are prototyping an object-based language called *Emerald* whose goal is to simplify distributed programming through language support while also providing acceptable performance and flexibility, both locally and in the distributed environment. The notion of object is fundamental to Emerald. We believe that objects are an excellent way to structure a distributed system because they encapsulate the concepts of process, procedure, data, and location. In Emerald, objects are the units of programming and distribution, and the entities between which communication takes place. However, we

This work was supported in part by the National Science Foundation under grants MCS-8004111 and DCR-8420945, by Københavns Universitet (the University of Copenhagen), Denmark under grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

do not believe that all aspects of distribution should be hidden from the programmer, and therefore the Emerald language has explicit notions of location and mobility.

In the following sections we describe Emerald objects and the Emerald type system. Emerald is strongly typed and has been carefully designed so that types may be resolved statically (i.e., at compile time). Static typing permits the most efficient code to be generated, at some loss in flexibility. If this cost is significant, the programmer can explicitly delay type checking until run time. This facility might be used, for example, when invoking an unknown object obtained from a file service.

2. Emerald Objects

All entities in the Emerald system are objects. This includes small entities, such as Booleans and integers, and large entities, such as directories and compilers. While different objects may be implemented with different techniques, all objects exhibit uniform semantics. An object can be manipulated only through invocation; no external access to an object's data is permitted. Objects can be invoked remotely and can move from node to node.

Each Emerald object has four components:

1. A *name*, which uniquely identifies the object within the network.
2. A *representation*, which consists of the data stored in the object. The representation of a programmer-defined object is composed of a collection of references to other objects.
3. A set of *operations*, which define the functions and procedures that the object can execute. Some operations are exported and may be invoked by other objects, while others may be private to the object.
4. An optional *process*, which operates in parallel with invocations of the object's operations. An object with a process has an active existence and executes independently of other objects. An object without a process is a passive data object and executes only as a result of invocations.

An Emerald object also has several attributes. An object has a *location* that specifies the node on which that object is currently resident. Emerald objects can be defined to be *immutable*; this simplifies sharing in a distributed system, since immutable objects can be freely copied. Immutability is an assertion on the part of the programmer that the abstract state of an object does not change; it is not a concrete property and the system does not attempt to check it.

Emerald supports concurrency both between objects and within an object. Within the network many objects can execute concurrently. Within a single object, several operation invocations can be in progress simultaneously, and these can execute in parallel with the object's internal process. To control access to variables shared by different operations, the shared variables and the operations manipulating them can be defined within a monitor [10, 16]. Processes synchronize through builtin condition objects. An object's process executes outside of the monitor, but can invoke monitored operations should it need access to shared state.

Each object has an optional *initially* section – a parameterless operation that executes exactly once when the object is created and is used to initialize the object state. When the initially operation is complete, the object's process is started and invocations can be accepted.

3. Abstract Types

Central to Emerald is the concept of *abstract type*. An abstract type defines a collection of *operation signatures*, that is, operation names and the types of their arguments and results. All identifiers in Emerald are typed: the programmer must declare the abstract type of the objects that an identifier may name. An abstract type is represented by an Emerald object that specifies such a list of signatures. For example, if the variable MyMailbox is declared as:

var *MyMailbox : AbstractMailbox*

then any object that is assigned to MyMailbox must implement (at least) the operations defined by AbstractMailbox.

We say that the abstract type of the object being assigned must *conform* to the abstract type of the identifier. Conformity is the basis of type checking in Emerald. Informally, a type S conforms to a type T (written $S \triangleright T$) if:

1. S provides at least the operations of T (S may have more operations).
2. For each operation in T, the corresponding operation in S has the same number of arguments and results.

3. The abstract types of the results of *S*'s operations conform to the abstract types of the results of *T*'s operations.
4. The abstract types of the arguments of *T*'s operations conform to the abstract types of the arguments of *S*'s operations (i.e., arguments must conform in the opposite direction).

Conformity is defined formally in [8]; it is similar to type compatibility in Owl [24].

The relationship between abstract types and object implementations is many-to-one in both directions. A single object may conform to many abstract types, and an abstract type may be implemented by many different objects. Although Emerald requires that the abstract type of each identifier be manifest, the type of the object that is to be assigned to an identifier may not be known until run time. In such a case, the conformity check will be performed at run time. However, very often enough information will be available at compile time for conformity to be checked statically.

It is important to note the difference between type conformity in Emerald and subclasses in Smalltalk [14]. In Emerald, the relationship between an object and the abstract type(s) that it implements is one of shared *interface*. An object supports a superset of the operations defined by its abstract types and each the supported operations must conform to the corresponding operations in the abstract types. In Smalltalk, the relationship between a subclass and its superclass is one of shared *implementation*. A subclass is free to redefine the signatures of the messages that it receives, but it necessarily shares the superclass's representation (instance variables) and typically shares many methods as well.

We expect that Emerald's strong typing will have several benefits. First is early detection and notification of programming errors. In Smalltalk, errors of the "message not understood" variety can be generated only at run time. We would like to detect many such errors through compile time type checking. In cases where we cannot completely type check an assignment at compile time, our run-time messages can be more explicit, e.g., "object Q does not conform to abstract type P". The second benefit is increased performance. In most cases, compile time conformity checks permit us to do assignment and invocation without run time type checking. In some cases a run time check is required; however, the check is made on assignment, and once it succeeds subsequent invocations can execute without further checking. Finally, in many cases compile time type information allows us to generate very efficient invocation code. This is described in more detail in Section 8.

4. Object Creation

As described above, an identifier in Emerald programs has an abstract type, and an object must conform to that abstract type to be named by the identifier. However, Emerald objects do not require a Class object for their creation. In most object-based systems, the programmer first specifies a class object that defines the structure and behavior of all its *instances*. The class object also responds to *new* invocations to make new instances.

In contrast, an Emerald object is created by executing an *object constructor*. An object constructor is an Emerald expression that defines the representation, the operations, and the process of an object. For example, suppose the Emerald program in Figure 4.1 is executed. This results in the creation of a single object. If we wished to create more *oneEntryDirectories* we would embed the object

```
const myDirectory == object oneEntryDirectory
  export Store, Lookup
  monitor
    var name : String
    var AnObject : Any

    operation Store [ n : String, o : Any ]
      name ← n
      AnObject ← o
    end Store

    function Lookup [ n : String ] → [ o : Any ]
      if n = name
        then o ← AnObject
        else o ← nil
      end if
    end Lookup

    initially
      name ← nil
      AnObject ← nil
    end initially

  end monitor
end oneEntryDirectory
```

Figure 4.1: A oneEntryDirectory Object

```
const myDirectoryCreator ==
immutable object oneEntryDirectoryCreator
  export Empty

  const OED == type OED
    operation Store [ String, Any ]
    function Lookup [ String ] → [ Any ]
  end OED

  operation Empty → [ result : OED ]
    result ← object oneEntryDirectory
      export Store, Lookup

      monitor
        var name : String
        var AnObject : Any

        operation Store [ n : String, o : Any ]
          name ← n
          AnObject ← o
        end Store

        function Lookup [ n : String ] → [ o : Any ]
          if n = name
            then o ← AnObject
            else o ← nil
          end if
        end Lookup

        initially
          name ← nil
          AnObject ← nil
        end initially

      end monitor
    end oneEntryDirectory
  end Empty
end oneEntryDirectoryCreator
```

Figure 4.2: A oneEntryDirectory Creator

constructor of Figure 4.1 in a context in which it might be repeatedly executed, such as the body of a loop or operation. This is illustrated in Figure 4.2. In this example, we construct the single object specified by the outermost object constructor. That object exports an operation called *Empty*; invoking the *Empty* operation executes the object constructor on lines 9 to 29, creating a new object that conforms to abstract type OED†. Conceptually, each object so created possesses its own copy of the code for *Store* and *Lookup*, as in SW2 [18]. In practice, there will be at most a single copy of the code on each node, and that copy will be shared.

The notion of object creator can be extended to as many levels as the programmer requires. For example, consider the builtin object *Array*. Array exports an *of* operation that expects an abstract type argument, as in

Array.of[Integer] .

The result of this invocation is an object that exports an operation *Create* of zero arguments. When *Create* is invoked, as in

Array.of[Integer].Create

the result is an array object, i.e., an object that exports operations like *setElement*, *getElement*, and *upperbound*.

In a similar way, one could define a typed OneEntryDirectory creator creator that is parameterized by the type of the directory entry as shown in Figure 4.3.

```
const myTypedDirectoryCreatorCreator ==
immutable object typedDirectoryCreatorCreator
  export of

  function of [ ElementType : AbstractType ] →
    [ result : DirectoryCreatorType]
      where
        OED == type OED
          operation Store [ String, ElementType ]
          function Lookup [ String ] → [ ElementType ]
        end OED
        DirectoryCreatorType == type T
          operation Empty → [ result : OED ]
        end T
      end where

    result ← object typedDirectoryCreator
      export empty

      operation Empty → [ result : OED ]
        result ← object oneEntryDirectory
          export Store, Lookup
            .
            .
            .
        end oneEntryDirectory
      end Empty
    end typedDirectoryCreator
  end of
end typedDirectoryCreatorCreator
```

Figure 4.3: A typed Directory Creator Creator

† In order to declare *Empty*, it is also necessary to declare a new abstract type *OED*. This is avoided in a language like Russell [9] by making the compiler infer the types of operation results.

5. Abstract Types as Objects

Abstract types are objects that obey a particular invocation protocol: they export an operation (without arguments) called *getSignature* that returns a *Signature*. In other words, an abstract type is an object that conforms to the following abstract type:

```
immutable type abstractType
  function getSignature → [ Signature ]
end abstractType
```

Conversely, any object that conforms to the above type may be treated as an abstract type. For example, if we add the following function definition to Figure 4.2,

```
function getSignature → [ result : Signature ]
  result ← OED
end getSignature
```

we may use *oneEntryDirectoryCreator* as an abstract type. We may now write

```
var aDir: myDirectoryCreator
aDir ← myDirectoryCreator.Empty
```

rather than

```
var aDir : AbstractDirectory
aDir ← myDirectoryCreator.Empty
```

Given the dual role of *myDirectoryCreator*, we see that it may have been appropriate to give it a less descriptive name. Similarly, we may define the primitive object *Array* such that the object returned by the *of* operation may be used as an abstract type‡. This allows us to write

```
var a: Array.of[Integer]
a ← Array.of[Integer].Create
```

6. Supporting Multiple Implementations

The most important goal of the Emerald design is the support of a uniform object model. The semantics of all objects, whether large or small, local or distributed, should be independent of the implementation technique. This uniformity should hold both for the programmer who builds objects and types, and for the application that invokes them.

The best example of a system with a uniform object model is Smalltalk. One characteristic that makes this possible is that Smalltalk is not distributed. In a distributed environment, the different implementation techniques that must be used for local and remote invocation often lead to the use of different abstractions for local and remote objects. For example, in the MIT Argus system [21], there are two different entities: Argus Guardians [20], which represent the abstraction of a node, and CLU Clusters [19], which represent local objects contained inside Guardians. In the Eden Programming Language [7], used to build applications on the Eden system [4], large network-wide entities are written as Eden objects, while local entities are defined using Concurrent Euclid data structures [17].

The problem with the two model approach is that the programmer must decide which model to use. Because the two models are semantically distinct, once an object is implemented in one style it must be rewritten for use as the other. For example, if we build an Eden tree object in EPL and later need a tree for internal use within another object, we must either design and code a different tree, or suffer the inefficiencies of the more general implementation.

In Emerald, all objects are coded using the single object definition mechanism. At compile time, the Emerald compiler chooses among several implementation styles for the object, picking one that is appropriate to the object's use. Different implementations tradeoff representation efficiency and invocation overhead for generality. Three different implementation styles are used.

1. *Global objects* are those that can be moved within the network and can be invoked by other objects not known at compile time (in other words, references to them can be exported). These objects are heap allocated by the Emerald kernel and are referenced indirectly. An invocation may require a remote procedure call.
2. *Local objects* are local to another object (i.e., a reference to them is never exported from that object). They are heap allocated by compiled code. These objects never move independently of their enclosing object. An invocation may be implemented by a local procedure call or by inline code.
3. *Direct objects* are local objects except that their data area is allocated directly in the representation of the enclosing object. These are used mainly for builtin types, structures of builtin types, records, and other simple objects whose organization can be deduced at compile time.

‡ We previously stated that the abstract type of every identifier in Emerald must be manifest. The expression *Array.of[Integer]* is manifest since the target (*Array*) is immutable, the operation (*of*) is a function, and the argument (*Integer*) is immutable. The expression can therefore be evaluated by the compiler.

Thus, Emerald is similar to EPL and Argus, in that there are several different implementation styles with varying performance characteristics. However, unlike these languages, the implementation differences are hidden from the programmer. The compiler chooses the best implementation based on compile time information. In many cases, the compiler can determine the implementation of local objects and can use this information for further optimizations. If the compiler knows only the abstract type then it must assume the more general object invocation mechanism.

7. Distribution Support

Emerald is designed for the construction of distributed applications. As previously stated, we believe that objects are an excellent way of structuring such programs because they provide the units of processing and distribution. This belief has been confirmed by our experience with distributed applications in Eden [1-3, 6].

The tendency of many distributed systems is to hide distribution from the programmer. For example, in Xerox RPC [5], remote procedure calls were added to Cedar Mesa. In so far as it was possible, remote procedure calls were designed to be semantically identical to local procedure calls. This is obviously a desirable property and is what makes RPC so attractive; programs can be written and debugged on a single node using local procedures and then easily distributed.

Emerald supports the same notion with object invocation. All objects are manipulated through invocation, and all invocations are location independent; it is the responsibility of the run-time system to locate and transfer control to the target object. Remote invocation achieves the same benefits as remote procedure call.

While it is crucial that invocation be location independent, it is not necessary that an object's location be invisible. Many applications may choose to ignore distribution, but others may wish to benefit from location dependence. For example, a replication manager may wish to distribute object replicas on different nodes, or two applications may wish to be co-located during periods of high activity. Applications that are concerned with distribution may wish to discover and modify objects' locations, but they still benefit from location-independent invocation.

For these reasons, the Emerald language includes a small number of location primitives. Basic to these primitives are node objects, which are the logical location entities in the system. A node is an abstraction of the concept of a physical machine, but it is possible for several node objects to exist on a single machine. (In fact, in our current implementation, a node is really an address space in which objects are contained.) An object can:

1. *Locate* another object, i.e., determine on what node it resides.
2. *Fix* another object at a particular node.
3. *Unfix* an object, i.e., make it movable following a *fix*.
4. *Move* an object to another location.

In all cases, location is specified through a reference to a target object; the location thus described is the node on which the target currently exists. The target can be an explicit node object, or any other object.

The choice of parameter passing semantics is crucial to both remote procedure call and object invocation. In an object-based system, the obvious choice is call-by-object-reference. Since the value of a variable is a reference to an object, it is that reference (the object name) that is passed in an invocation. This presents a potentially serious performance problem on distributed systems; any invocation by a remotely invoked object of its parameters is likely to cause another remote invocation. For this reason, systems such as Argus have required that parameters to remote calls be passed by value, not by reference [15].

Because Emerald objects are mobile, it may be possible to avoid many remote references by moving parameter objects to the site of the callee. Whether or not this is worthwhile depends on the size of the parameter object, the number of active invocations, and the number of invocations to be issued by the called object. We expect that parameter objects will be moved in two cases. First, based on compile-time information, the Emerald compiler may decide to move an object along with an invocation. For example, small immutable objects may be copied cheaply and are obvious candidates. Second, the programmer may decide that an object should be moved based on knowledge about the application. To make this possible, Emerald supports a parameter passing mode that we call *call-by-move*. A call-by-move parameter is passed by reference, as is any other parameter, but at the time of the call it is relocated to the destination site. Following the

call it is returned, unless it is a copy of an immutable object, in which case it is garbage collected.

Call-by-move is a convenience and a performance optimization. The move could be done explicitly with the *move* primitive, but that would require more explicit code and would not allow packaging of parameter objects in the same message as the invocation. While call-by-move co-locates the parameter with the target object, it increases the cost of the call and may cause extra remote references from the call's initiator. One of our goals is therefore to experiment with various policies for using this parameter passing mechanism.

8. Implementation Aspects

The Emerald system is being prototyped on a small network of DEC MicroVAX II workstations connected by a ten Megabit/s Ethernet. The system runs on top of Berkeley Unix. Using Unix has some performance consequences, particularly in inter-machine communication; however, it has little impact on performance within a node.

Emerald is implemented in two closely related parts, the Emerald compiler and the Emerald kernel. An Emerald node is a single Unix address space in which the kernel and all objects located on that node execute. Processes within objects are lightweight for fast context switching and invocation. Processes and monitors are implemented as in Concurrent Pascal [11]. Protection is provided by the compiler, as in Xerox Mesa/Pilot [23].

As previously described, objects can be implemented in several ways. Direct objects are supported directly ("inline") within other objects and are invisible to the kernel. Other objects are created by kernel calls and supported by kernel data structures.

To support remote referencing and mobility, object references must be location independent. Since direct objects are compiled inline or allocated directly in invocation records, they can be referenced by offset within the object or data structure. All other objects are referenced by the address of a node-local *object descriptor*. The object descriptor contains the object's unique ID, a location hint if the object is remote, and a pointer to its data area, process, and code if the object is locally resident. An object descriptor must exist on a node as long as any references to the corresponding object remain on that node. Object descriptors are heap allocated by the kernel and garbage collected.

Each node also has an *object table* that contains an entry for every remotely referable object on that node. The object table is used to determine if an object exists on a node, and if so to provide the address of its object descriptor.

Because an object reference is the address of an object descriptor, references are machine-dependent and must be translated when an object moves. When the kernel moves an object, it sends along a mapping of object descriptor addresses to object IDs. On the receiving node, new object descriptors are allocated as needed and the object references are modified to point to them. On the sending node, the object descriptor for a moved object is modified to indicate the object's new location. The location is treated as a hint; we are using a location protocol based on forwarding addresses [13] supplemented by a reliable broadcast that is used when forwarding addresses are lost (due to crashes).

To help the kernel in finding references that need to be translated, the compiler generates *templates* that describe the structure of each object. Code and templates are stored in kernel structures called *concrete types*. One concrete type exists for each object constructor. They are immutable, and copies of them may exist on many nodes. When an object is moved to another node, the concrete type is not sent along; it is requested by the target node only if needed.

Locating the code for an invoked operation is simplified by the Emerald type system. The abstract type of a variable specifies the operations that can be performed on the object it names. At run time, the variable references an object with a specific concrete type. Even though the object may have more operations than the abstract type, the additional operations cannot be invoked.

The data structure used to locate operations is called an Abstract-Concrete vector. We associate with each variable a vector with one entry for each operation defined by its abstract type. The contents of the entry is the address of the corresponding procedure entry point in the concrete type. On invocation, a simple indexing operation produces the address of the procedure to call.

When an assignment is made, the vector may have to be changed if the new object is implemented by a different concrete type. The compiler generates code to perform this change if it cannot tell the concrete type of the object

to be assigned. Note that an Abstract-Concrete vector must exist for every pair <abstract type, concrete type>, but these vectors can be shared by all variables (on the same node) that have the same abstract/concrete binding.

9. Conclusions

The goal of Emerald is to support the construction of object-based distributed programs while providing excellent performance for local and private objects. Emerald's novel features include its single object model used for both small private objects and large mobile objects, its abstract type system that permits static type checking while allowing multiple implementations, and its explicit notion of location.

Languages like Smalltalk rely heavily on the concept of *Class*. However, Classes have at least three functions: they generate instances, they act as a repository for the code of those instances, and (through the inheritance hierarchy) they provide a classification scheme for instances. Emerald allocates these functions to separate mechanisms: objects are created by explicit constructors, code sharing is managed by the kernel, and abstract types provide a classification scheme that is independent of an object's implementation.

We have been designing Emerald for over a year and are now building a prototype implementation. We currently have a primitive single-node kernel and a compiler capable of compiling simple Emerald objects into VAX machine code. Early performance tests indicate that we can execute a local invocation in approximately the same time as that required by a comparable calling sequence using the VAX *CallS* instruction, and a process context switch in about seven times the *CallS* time.

References

[1] G. Almes and C. Holman, "Edmas: An Object Oriented Locally Distributed Mail System", Technical Report no. 84-08-03, December 13, 1984.

[2] G. T. Almes, A. P. Black, C. Bunje and D. Wiebe, "Edmas: A Locally Distributed Mail System", *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984.

[3] G. Almes and C. Holman, "The Eden Shared Calendar System", Technical Report 85-05-02, Department of Computer Science, University of Washington, June 22, 1985.

[4] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering SE-11*, 1 (January 1985), 43-59.

[5] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59. Presented at the Ninth ACM Symposium on Operating System Principles October, 1983.

[6] A. P. Black, "Supporting Distributed Applications: Experience with Eden", *Proceedings of the Tenth ACM Symposium on Operating System Principles*, Orcas Island, Washington, December 1985, 181-93.

[7] A. P. Black, "The Eden Programming Language", Technical Report 85-09-01, Dept. of Computer Science, University of Washington, Seattle, Washington, September 1985.

[8] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "Distribution and Abstract Types in Emerald", Technical Report 86-02-04, Dept. of Computer Science, University of Washington, Seattle, Washington, February 1986. To appear in *IEEE Transactions on Software Engineering*.

[9] H. Boehm, A. Demers and J. Donahue, "An Informal Description of Russell", Technical Report 80-430, Dept. of Computer Science, Cornell University, Ithaca, New York, October 1980.

[10] P. Brinch Hansen, "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering 2* (June 1975), 199-205.

[11] P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice Hall, Inc., 1977.

[12] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", *IEEE Software 1*, 2 (April 1984), 19-42.

[13] R. J. Fowler, "Decentralized Object Finding Using Forwarding Addresses", Ph.D. Dissertation, Technical Report 85-12-1, Dept. of Computer Science, University of Washington, December 1985.

[14] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

[15] M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types", *Trans. Prog. Lang and Systems 4* (October 1982), 527-51.

[16] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Comm. ACM 17*, 10 (October 1974), 549-57.

[17] R. C. Holt, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, 1983.

[18] M. R. Laff and B. Hailpern, "SW2 - An Object-base Programming Environment", *SIGPLAN Notices 20*, 7 (July 1985). In Proceedings of the ACM

SIGPLAN 85 Symposium on Language Issues in Programming Environments.

[19] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM 20*, 8 (August 1977), 564-576.

[20] B. Liskov and R. Scheiffer, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *9th ACM Symp. on Prin. of Prog. Lang.*, 1982.

[21] B. Liskov, "Overview of the Argus Language and System", Programming Methodology Group Memo 40, M.I.T., Laboratory for Computer Science, February 1984.

[22] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating Systems Kernel", *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, October 1981, 64-75.

[23] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell, "Pilot: An Operating System for a Personal Computer", *Comm. ACM 23*, 2 (February 1982), 81-92.

[24] C. Schaffert, T. Cooper and C. Wilpolt, *Owl Reference Manual*, Eastern Research Lab, Digital Equipment Corporation, Hudson, Massachusetts, February 7, 1985.