

Distribution and Abstract Types in Emerald

A. Black
N. Hutchinson
E. Jul
H. Levy
L. Carter



Distribution and Abstract Types in Emerald

ANDREW BLACK, NORMAN HUTCHINSON, ERIC JUL, HENRY LEVY, AND LARRY CARTER

Abstract—Emerald is an object-based language for programming distributed subsystems and applications. Its novel features include 1) a single object model that is used both for programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. This paper outlines the goals of Emerald, relates Emerald to previous work, and describes its type system and distribution support. We are currently constructing a prototype implementation of Emerald.

Index Terms—Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.

I. INTRODUCTION

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed

environment. Second, it must provide for both intra- and internode communication in an efficient manner. The semantics of communication and computation should be consistent in the local and remote cases. Third, it must allow the programmer to exploit the inherent parallelism and availability of a distributed system. Fourth, since shutting down and recompiling an entire distributed system in order to modify some component is unacceptable, the language must permit system extensibility without recompilation; existing programs must continue to work in collaboration with new programs.

Our research focuses on simplifying the programming of distributed subsystems and applications by providing language support for distribution. We have designed an object-based language, called *Emerald*, and a distributed run-time system for Emerald that facilitate the construction of distributed programs for a local area network of independent nodes (workstations). The novel features of Emerald include: 1) a single object model that is used for both programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. The goal of our research is to demonstrate the feasibility of using one simple semantic model for programming both sequential, single-node applications, and concurrent, potentially distributed applications. We currently have a prototype Emerald compiler and run-time system running on a local area network of VAX® workstations.

The next sections present a discussion of previous work in distributed programming languages and an overview of Emerald. Following sections describe the type system and the support for distribution.

II. REVIEW OF PREVIOUS SYSTEMS

To date, languages have supported distribution in several different ways. In the Xerox Cedar System [33], a remote procedure call facility allows programs to access remote servers through standard Cedar/Mesa procedure calls [2]. The advantage of this approach is that it requires no change to the semantics of the language. Automatically generated stub routines on the client and server machines are responsible for packing and unpacking parameters and transmitting and receiving messages. Programmers access a remote service in the same way that they would access a local service, except that they must explicitly locate and connect to the service before it can be used.

®VAX is a registered trademark of Digital Equipment Corporation.

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

A. Black, N. Hutchinson, E. Jul, and H. Levy are with the Department of Computer Science, University of Washington, Seattle, WA 98195.

L. Carter is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. This work was performed while he was a visitor at the University of Washington.

IEEE Log Number 8611363.

TABLE I
EMERALD LANGUAGE FEATURES

	Emerald	Argus		Xerox RPC	Eden		Smalltalk
		CLU	Guardians		EPL	Objects	
Uniform Object Model	✓	✓					✓
Distribution	✓		✓	✓		✓	
Mobile Objects	✓					✓	
Direct Objects	✓	✓		✓	✓		✓
Concurrency Control	✓		✓	✓	✓	✓	
Type Checking	✓	✓	✓	✓	✓		
Abstract Types	✓			✓			

At the University of Washington, the Eden Programming Language (EPL) [3] has been developed for writing distributed applications on the Eden system [1]. EPL is an extension of Concurrent Euclid [19] that provides location-independent invocation of Eden's objects in an integrated distributed system. Because the location of an Eden object is (conceptually) evaluated at each invocation, objects are free to move at any time; one need not locate or connect to an object before invoking it.

The M.I.T. Argus system [24], [25] is an ambitious distributed language project that extends the CLU language [23] to support atomic transactions in a distributed environment. An Argus *Guardian* encapsulates the notion of a physical machine. Inside a guardian are data objects and processes. One Guardian communicates with another by calling a handler in the target Guardian, to which data are passed by value.

In general, object-based systems and languages have viewed their objects in two ways: as large, long-lived resources (e.g., files) as in operating systems such as Hydra [35] and StarOS [20], or as small resources (e.g., records and integers), as in languages such as CLU [23] and Smalltalk [15]. In a distributed environment, both views seem to have their place; the Argus and EPL languages each support two kinds of objects. Argus has Guardians, which are network-wide objects, and CLU objects, which are local to a Guardian; EPL has network-wide Eden objects that contain local EPL variables, monitors, and modules. The reason for this dichotomy is one of locality and performance; local objects communicate through shared store, while network objects communicate through message passing, which requires more communications overhead. Unfortunately, this requires the programmer to use two different object abstraction mechanisms, to code in two different styles, and to foresee all possible uses to which an object will be put. For example, while programming a Collaborative Editing System in Argus, Greif *et al.* [16] have observed that a designer can be forced to use a Guardian where a cluster might be more appropriate.

Emerald has drawn on the experience of all of these systems. The most important difference between Emerald and these systems is Emerald's uniform model of computation. Like Smalltalk, all entities in Emerald are ob-

jects, and a single semantic model suffices to define them. Unlike Smalltalk, however, Emerald is a distributed programming language; its object model is sufficient to describe both local data objects and potentially remote objects containing independent processes. Table I enumerates the principal features of Emerald and compares them to those of Argus, Xerox RPC, EPL, and Smalltalk. The following section provides a brief overview of the Emerald programming language, focusing on its uniform object model, and following sections deal with two important aspects of Emerald: its type system which is based on the concept of abstract types, and its support for distribution.

III. INTRODUCTION TO EMERALD

Emerald is object-based and all information is encapsulated in objects. An object model is appropriate for a distributed system because it implicitly defines 1) the units of distribution and movement, and 2) the entities that communicate. All objects in Emerald are coded using the same object definition mechanism, regardless of the way in which they will be used. The Emerald object model is appropriate for defining *small* objects such as integers, characters, and Booleans as well as *large* objects such as directories and compilers. While different objects may be represented by the system in different ways, all objects exhibit the same semantics. Each Emerald object exports a set of operations; an object can be manipulated only by invocation of one of those operations. Furthermore, Emerald objects are *mobile*. Objects can move at any time, and can be invoked without knowledge of their location.

Each Emerald object has four components:

- 1) An *identity*, which distinguishes the object from all others 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 defines 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

