The Eden System: A Technical Review

GUY T. ALMES, MEMBER, IEEE, ANDREW P. BLACK, EDWARD D. LAZOWSKA, AND JERRE D. NOE, SENIOR MEMBER, IEEE

Abstract—The Eden project is a five year experiment in designing, building, and using an "integrated distributed" computing system. We are attempting to combine the benefits of integration and distribution by supporting an object based style of programming on top of a node machine/local network hardware base. Our experimental hypothesis is that such an architecture will provide an environment conducive to building distributed applications.

This technical review is written three years into the project. We begin by summarizing the Eden system: its concepts, history, status, and context. We next discuss the way in which the task of supporting the Eden architecture is divided between the kernel, the programming language, and user-level (library) code; we describe the experiences with paper designs and prototype implementations that led us to this division of labor. We then show how distributed applications make use of various aspects of the Eden architecture. We conclude by providing some preliminary evaluations based on our experiences to date.

The objective of our research is to assess the benefits (in terms of programmability) and the costs (in terms of necessary support) of our system architecture. We feel we have gained insights on a number of questions of relevance well beyond Eden or Eden-like systems.

- How should the job of supporting a system such as Eden be divided between the kernel, the programming language, and user-level (library) code?
- How do distributed applications make use of various aspects of the Eden architecture (location independence, concurrency, etc.)?
- Of the many design and implementation choices we have made, which are apparently good or apparently bad, based on our experience to date?

Index Terms—Capability, Concurrent Euclid, concurrent programming, distributed electronic mail, distributed program, distributed system, Eden, object-oriented system, remote procedure call.

Manuscript received October 26, 1983; revised June 14, 1984. This work was supported in part by the National Science Foundation under Grant MCS-8004111. Computing equipment was provided in part under a cooperative research agreement with Digital Equipment Corporation.

The authors are with the Department of Computer Science, University of Washington, Seattle, WA 98195.

0098-5589/85/0100-0043$01.00

I. An Overview

THE Eden project is an experiment in designing, building, and using an "integrated distributed" computing system.

Eden began with the observation that contemporary computing systems tend towards two extremes: centralized systems providing a high degree of integration but poor support for personal computing, and distributed systems providing good support for personal computing but a low degree of integration. There exists a spectrum of possibilities between these two extremes.

In Eden, we are attempting to combine the benefits of integration and distribution by supporting an object based style of programming on a collection of node machines connected by a local network. Our experimental hypothesis is that such an architecture will provide an environment conducive to building distributed applications: we believe that the modularity afforded by an object based approach will be especially valuable in such an environment.

The objective of our research is to assess the benefits (in terms of programmability) and the costs (in terms of necessary support) of our architecture. With two years of our five year project remaining, a complete assessment is not yet possible. However, we feel that we have gained insights on a number of questions of fairly general interest.

- How should the job of supporting a system such as Eden be divided between the kernel, the programming language, and user-level (library) code? This is the subject of Section II.
- How do distributed applications make use of various aspects of the Eden architecture (location independence, concurrency, etc.)? This is the subject of Section III.
- Of the many design and implementation choices we have made, which are apparently good or apparently bad, based on our experiences to date? This is the subject of Section IV.

In the remainder of this section we quickly review the concepts, history, status, and context of Eden.

A. Concepts

In Eden, a distributed application is built as a collection of *Eden objects*, or *Ejects*. From the point of view of the applications programmer, Ejects have the following characteristics.

- Ejects communicate with one another by means of *invocations*—messages sent from one Eject to another requesting a particular action and a reply.
- Each Eject has a unique identifier. The Eject initiating an invocation must have a *capability* for the Eject that is the target of that invocation. This capability contains both the unique identifier of the target Eject and a set of *access rights*.
- Possession of a capability for an Eject does not imply any knowledge of its location within the Eden system. Ejects are mobile, and invocation is location independent.
- Each Eject has a *concrete Edentype*—a code segment that defines the *invocation procedures* supported by that Eject. For example, if *msg* is a capability for an Eject of Edentype MailMessage, and if Ejects of this Edentype support an invocation procedure named *Deliver*, then another Eject can make the invocation *msg.Deliver*(<*parameter list*>). Edentypes are conceptually similar to the collection of methods that make up a Smalltalk Class [1].
- Each Eject has a *data part* that defines its state, both *long term* (the values maintained between invocations) and *short term* (information such as the local variables and parameters of invocations).
- Ejects are active. Each Eject typically consists of a number of processes which communicate with one another via monitors. When a process in one Eject invokes another Eject, that process blocks until the invocation completes, but other processes within the invoking Eject can continue to run. In addition to processes that respond to invocations and processes that make invocations, an Eject may contain processes that perform internal housekeeping operations. This contrasts with Smalltalk, where sending a message transfers control to the receiver, and replying to a message causes the receiver to quiesce.
- An Eject may perform a *Checkpoint* operation, which creates a *passive representation*—a data structure designed to endure system crashes. The data in a passive representation should be sufficient to enable the Eject to reconstruct its long term state. The *Checkpoint* primitive is the only mechanism provided by the Eden kernel whereby an Eject can access permanent storage.
- In practice, Ejects are not always active, either because they (or their computers) have crashed, or because they have explicitly deactivated themselves in order to economize on the use of system resources. If a passive checkpointed Eject is sent an invocation, it will be *activated* automatically by the Eden kernel. An Eject activated in this way will read its passive representation and put its internal data structures in the state that existed at the time of the *Checkpoint* operation.

Building a distributed application in Eden involves defining appropriate Edentypes, using the *Eden Programming Language (EPL)*. The characteristics of Ejects noted in the preceding paragraphs have the following implications.

- Modularity at the system level allows applications to be built from independent components. While the programmer sees a world of active Ejects, code is written for one Edentype at a time, not for the whole world.
- The internal structure of each Eject is the concern of its programmer alone. For example, choices concerning the number of processes used, the ways in which they interact, and whether the Eject suspends itself awaiting the reply to an invocation or continues with other tasks, are left to the programmer.
- The only way the "outside world" can act upon an Eject is by invocation, so the information released to the outside world is completely at the control of the programmer.
- Eden does not insulate the programmer from crashes, but provides certain primitives (e.g., *Checkpoint*) that can be used to build robust applications.

B. History

The Eden project began in September 1980, as the first grant under the National Science Foundation's Coordinated Experimental Research Program. This paper is written after three years of the project's five year lifetime.

During the first year of the project we refined the notion of Eject, wrote a partial specification of Eden, and designed and implemented prototype node machines based on Intel's iAPX

432 processor and Ethernet. We were attracted to the 432 architecture because of the flexible support that it provided for processes, interprocess communication, and address spaces.

During the next six months of the project we built a single-node prototype of Eden, called Newark, on top of a VAX system running VMS. This experience taught us a number of important technical lessons concerning interfaces: between Eden and existing computing environments, between the Eden kernel and Ejects, and between the Eden programmer and Ejects. It also taught us that, in our choice of the 432 as the eventual host for Eden, we had underestimated the importance of factors such as stability, quality of development environment, and diversity of existing software.

Based on our Newark experiences, we devoted the next six months to revising our specifications, to designing an implementation of distributed Eden on top of a collection of VAX systems running Berkeley UNIX™,[1] and to designing a programming language providing proper support for Eden programmers.

The most recent year of the project has been devoted to the implementation of this version of the kernel and EPL, and to the construction of a small set of applications.

C. Status

Version 1.0 of the Eden kernel is now operational on a collection of VAX/UNIX™ systems interconnected by Ethernet. Eden coexists with UNIX™, in the sense that an individual can make simultaneous use of UNIX™ and Eden services. This coexistence is crucial in minimizing the software effort required to make Eden usable.

The kernel is written in C. On each node it consists of a single UNIX™ process; the code size of this process is approximately 90 kbytes. In addition, a UNIX™ process is associated with each Eject.

The Eden Programming Language gives the programmer the illusion of multiple threads of control within each Eject, and also makes invocation look like a procedure call. EPL is based on the Concurrent Euclid language [2], [3].

We have built several demonstration applications. One of the most interesting is a locally distributed mail system which consists of two principal Edentypes: MailBox and MailMessage. A MailMessage is composed through a mail system interface, and is delivered by invoking (in a location independent manner) the MailBox of the intended recipient. Of particular interest are the simplicity of the design and the ease of implementation afforded by Eden. Other applications include a transput (I/O) facility, a file system, and an appointment calendar system. The first and second of these are examples of facilities that would be part of the "operating system" in a more traditional environment. The second and third are examples in which transactions are supported at the application level, using primitives provided by the Eden kernel. We also have programmed certain numerical algorithms designed to exploit the parallelism possible within the system.

D. Context

Eden is a system for building distributed applications. Through the kernel and EPL, Eden provides high-level support for the sharing of information and processing capacity in a locally distributed system.

It is important to realize that although Eden is presently implemented on top of UNIX™, Eden is not an attempt to build a "distributed UNIX™" (e.g., UCLA's LOCUS project [4]). UNIX™ for us is an implementation vehicle and a source of software utilities.

The objectives of Eden are similar to those of M.I.T.'s Argus project, and Ejects resemble in many ways Guardians in Argus [5]. It probably is fair to say that Argus began as a programming language effort that viewed its kernel as run-time support, while Eden began as an operating systems effort that viewed EPL as important syntactic sugar. Both Ejects and Guardians are large, compared, say, to Smalltalk objects. A directory or a mailbox is a reasonable granularity for an Eject, while a Guardian corresponds most closely to a "logical node machine." The Clouds project at the Georgia Institute of Technology also is closely related to our work, although, like Argus and in contrast to Eden, Clouds provides explicit support for atomic actions [6].

Eden does not attempt to be a minimal cost IPC or RPC system. Our motivation for exploring an object based approach is our belief that building distributed applications is substantially more complex than building centralized applications, and that the assistance provided by such an approach in mastering this complexity warrants the additional run-time expense.

II. Supporting Eden

This section discusses the implementation of the Eden 1.0 system and the evolution of various aspects of this implementation over the course of the project. We feel that each of these subjects has relevance well beyond Eden or Eden-like systems, and thus will be of interest to the general systems community.

Table I summarizes many of the points that we will discuss. On the left are nine aspects of the Eden architecture. Across the top are three major stages in the evolution of our system. The entries in the table summarize how each aspect of the architecture was supported in each evolutionary stage. We suggest that Table I be reviewed both before and after reading this section.

A. Support for Concurrency within Ejects

In the present implementation of Eden, the underlying system is Berkeley UNIX™ running on VAX's. The interprocess communication mechanism used is the Accent IPC package [7]. The addition of the IPC package, an Ethernet driver, and a simple module to compute timestamps are the only changes we have made to the UNIX™ kernel.

In Eden 1.0, each active Eject executes within a separate UNIX™ process with its own address space. The Eden kernel manages this process using UNIX™ facilities, and communicates with the Eject via the IPC mechanism. This design isolates Ejects from each other during execution, and provides the firewalls necessary to prevent a bug in one Eject from causing a fault in another. Also, it allows the kernel to add new Edentypes and Ejects while the system is running. Since the Eden kernel knows about only one UNIX™ process for each Eject, the kernel design is considerably simplified.

[1] UNIX is a trademark of Bell Laboratories.

TABLE I
THE EVOLUTION OF EDEN

	Initial	*Newark*	*Eden 1.0*
Host Machine	Intel iAPX 432	Digital VAX	Digital VAX
Underlying System	iMAX	VMS	UNIX™
User Language	Extended Ada	Pascal	EPL
Multiple Processes Within an Eject	Kernel / System	Kernel / System	Language
Synchronization Within an Eject	Architecture (Ports)	System IPC (Mailboxes)	Language (Monitors)
Synchronous Invocation	Kernel	Kernel	Library and Language
Allocating Processes to Incoming Invocations	Kernel	Library	Library and User Code
Packing/Unpacking Parameter Lists	not considered	User; Pascal Records	EPL Preprocessor, ESCII
Protection of Capabilities	Architecture / Kernel	none	Kernel
Transmission of State in Checkpointing	not considered	Automatic; Single Segment	Explicit; Multiple Data Items
Implementation of the Kernel	Protected calls to Routines	System IPC to Kernel Process	System IPC to Kernel Process
Interface to Non-Eden Programs	none	VMS Programs Invoke Ejects	Ejects Use UNIX™

Eden 1.0 supports concurrency within Ejects by using the facilities of the Eden Programming Language (EPL) [8]. EPL is derived from Concurrent Euclid, a dialect of Toronto Euclid that provides multiple processes and Hoare monitors. Because they are implemented by the language, EPL's processes and monitors are extremely lightweight tools when compared to the process and IPC mechanisms provided by UNIX™.

B. The EPL Translator's Role in Invocation

Many remote procedure call systems employ language processors to enhance their usability (e.g., [9]). In Eden, EPL provides translator support to ease the task of declaring and using invocation procedures. For example, suppose that the programmer of Edentype Login wants to perform a *Lookup* operation on *RootDirectory*, an Eject of Edentype Directory, in order to obtain a capability for *LoginDirectory*, a particular user's login directory. The following piece of EPL code might be used.

```
const RootDirectory: Capability for Directory
    := Eject "RootDirectory"
var LoginDirectory: Capability for Directory
var Status: EdenStatus
var UserName: String
...
RootDirectory.Lookup(UserName, LoginDirectory, Status)
```

The programmer of Edentype Directory will have declared a special procedure for handling this operation.

```
invocation porcedure Lookup(
    CallersRights: EdenRights,
    SearchKey: String,
    var Result: Capability,
    var Status: EdenStatus) =
begin
...
end Lookup
```

(EPL requires the *CallersRights* parameter in the declaration of this invocation procedure, to allow the invoked Eject to check the invoker's access rights.)

These two code fragments appear in different Ejects; it is the job of EPL to ensure that the *RootDirectory.Lookup* procedure call in the Login Edentype does eventually activate the *Lookup* invocation procedure in *RootDirectory*. There are three essential functions involved: parameter packaging, stub generation for the invoked Eject, and stub generation for the invoking Eject. We discuss these in turn.

- *Parameter Packaging:* The ESCII facility (an acronym for Eden Standard Code for Information Interchange) allows operation names and parameter lists to be packaged into self-describing data structures. For example, the ESCII structure that packages the operation name and value parameters of the call

RootDirectory.Lookup(UserName, LoginDirectory, Status)

enables the target to determine that it contains 1) the string "*Lookup*," 2) the single string argument *UserName*, and 3) an indication that a single capability result is expected in the reply. This allows a great deal of type checking to be done at run-time. In comparison to similar facilities in other systems, ESCII is conservative in the variety of types it includes: a small number of base types and arrays of them are the most important.

- *Stub Generation for the Invoked Eject:* When the Directory Edentype is compiled, the translator builds a version of *CallInvocationProcedure* tailored to the invocation procedures it defines. *CallInvocationProcedure* is usually called directly by the Edentype programmer; its function will be discussed in Section II-C, but one of its tasks is to unpackage incoming ESCII's.
- *Stub Generation for the Invoking Eject:* At the same time, the translator generates a stub for each invocation procedure exported by Directory; these stubs are included in the EPL code of Login, and of any other Edentypes that invoke Directory. The stubs include calls to the ESCII packing routines and also to *SynchInvoke*, a lower-level invocation support routine that assists in handling outgoing invocation messages. *SynchInvoke* will be discussed in Section II-C.

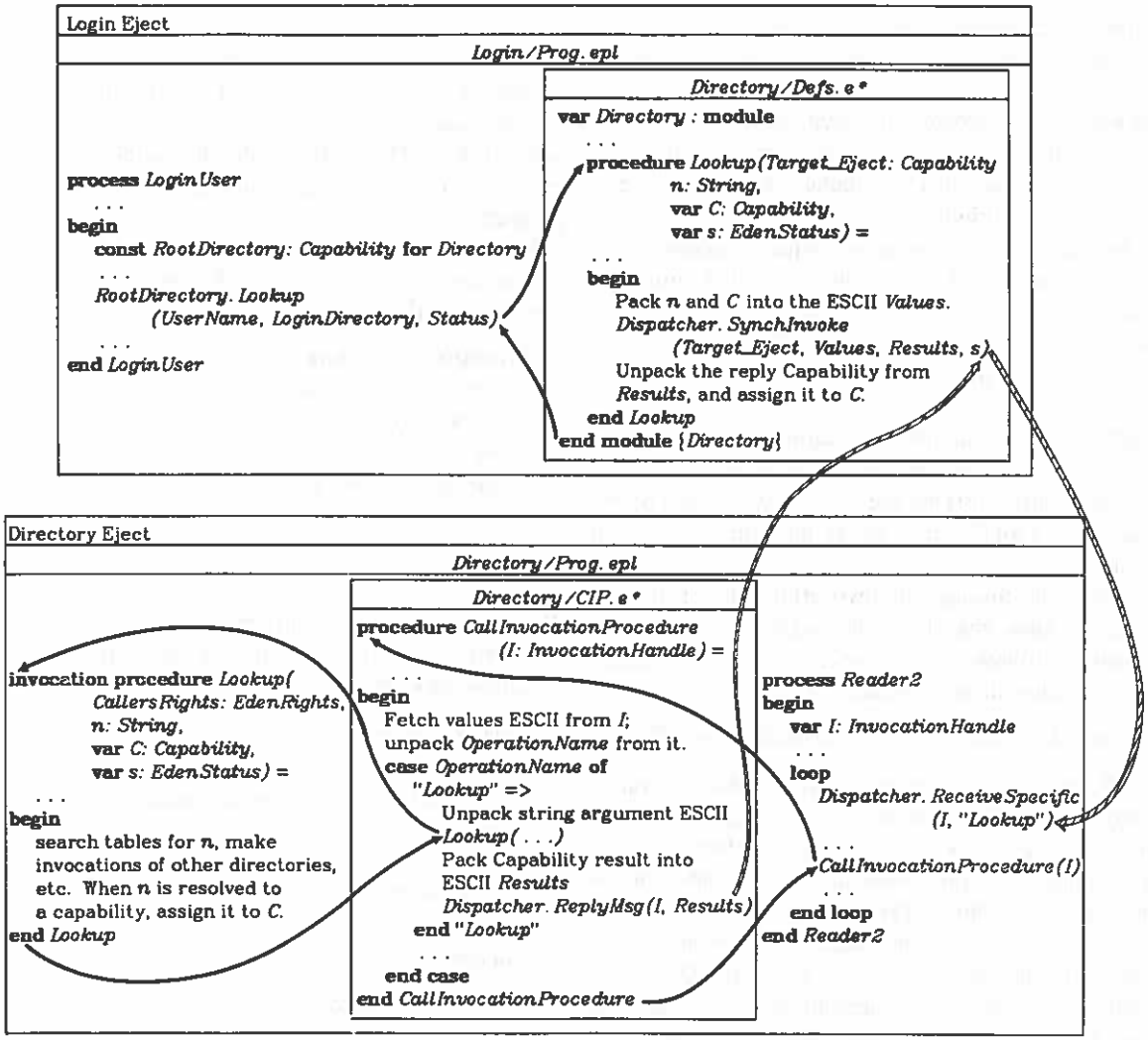
Fig. 1 shows how these pieces of code fit together, in addition to illustrating certain aspects of the lower-level support for invocation that will be discussed in Section II-C.

The design of EPL is very conservative: it represents a minimal set of extensions to Concurrent Euclid. This conservatism works to EPL's advantage: it enabled the implementation to be constructed within our limited resources, and it allows any programmer familiar with Pascal and the concept of modularity to absorb EPL with little difficulty.

C. Lower-Level Support for Invocation

EPL provides the highest of three levels of support for invocation. In this section we describe the other two. The arrangement of functions among these levels had to satisfy the following criteria.

- Each of the EPL processes within an Eject must be able to do (synchronous) invocations without blocking other processes.
- Similarly, the Eject needs to be able to wait for invocation requests and, when they arrive, to allocate them to a process for service.



This figure illustrates the *RootDirectory. Lookup* invocation referred to in Sections II-B and II-C. The upper part of the figure represents the Login Eject that initiates the invocation. The call *RootDirectory. Lookup(UserName,...)* within Login is shorthand for *Directory. Lookup(RootDirectory, UserName, ...)*, i.e., is a call on the stub procedure *Lookup* in the module *Directory*.

The lower box represents the Directory Eject named by Login's *RootDirectory* capability. It contains the process *Reader2* which receives the invocation request and calls the EPL-generated procedure *CallInvocationProcedure*. This in turn calls the invocation procedure *Lookup*, which finally accomplishes the task at hand.

The boxes marked * represent code that is generated automatically when the directory Edentype is compiled. The solid lines represent procedure calls/returns within Ejects. The hashed lines represent invocations/replies between Ejects.

Fig. 1. Invoking a Directory Eject.

- The programmer of an Edentype must be able to allocate these processes in an intelligent way so that deadlock can be avoided.
- The sending and receiving of invocations must be easy for the EPL programmer.
- The kernel should be as simple as possible.
- Invocation should be as efficient as possible.

Immediately below the EPL level is the synchronous invocation level, provided by a procedure library within each Eject. The essential contribution of this level is to use the lower asynchronous invocation level, together with EPL processes and monitors, to provide invocation facilities that can block a single process at the EPL level. There are three major procedures.

- *SynchInvoke* is used within the stubs generated by EPL. It invokes an Eject denoted by a "target" capability, and waits for the reply to arrive. Only the calling EPL process blocks during the call.
- *ReceiveOperation* is used directly by the programmer. It is called by an EPL process in order to wait for an incoming invocation request. Only this process blocks during the call. *ReceiveOperation* takes a set of operations as an argument; it returns only when a request for one of these operations is present. *ReceiveAny* and *ReceiveSpecific* operations also are provided for use when universal or singleton sets are desired.
- *CallInvocationProcedure* also is used directly by the programmer. It is called by an EPL process that has received an incoming request and wishes to call the appropriate invocation procedure. *CallInvocationProcedure* unpacks the parameters from the invocation request message, checks them for compat-

ibility with the invocation procedure, makes the call, and finally executes a *ReplyMsg* (supported by the asynchronous invocation level).

The Eden kernel itself provides the asynchronous invocation level. Note that there are two "sending" primitives (one for making requests and the other for replies) but that there is exactly one "receiving" primitive.

- *AsynchInvoke* sends an invocation request message to the target Eject. It does not wait for a reply, but rather returns an *InvocationHandle* to the caller so that the caller can recognize the reply when it comes via a future call on *ReceiveMsg*.
- *ReplyMsg* sends an invocation reply message back to the invoking Eject.
- *ReceiveMsg* receives the next invocation request or reply message. If no message is present, the call returns immediately. UNIX™ software interrupts are used to notify the user of the arrival of messages. (An EPL process can block until the arrival of such an interrupt.)

We now will walk through an invocation, first from the invoker's point of view, then from the target's. Reference to Fig. 1 will assist in following the presentation. Consider again the *Lookup* invocation shown previously.

RootDirectory.Lookup(UserName, LoginDirectory, Status)

When the EPL program is translated, the capability variable *RootDirectory* will be identified as one that refers to an Eject that supports the operations of the Directory Edentype. This allows EPL to locate the appropriate procedure header for the *Lookup* invocation procedure. (There is no implication, however, that a translation- or run-time check is made that *RootDirectory* refers to an Eject of concrete Edentype Directory; neither capability variables nor capabilities are typed. We elaborate on this point in Section III-G.) EPL identifies *RootDirectory* as the target of the invocation and the string *UserName* as the single argument value. The operation name *Lookup* and the argument are packaged into an ESCII and the call

SynchInvoke(RootDirectory, ValueESCII, ResultESCII, Status)

is made. This call on the Eden library results in two actions. First, the call

AsynchInvoke(RootDirectory, ValueESCII, OutHandle, Status)

is made to send the invocation request message to the target. The *OutHandle* parameter is provided by the kernel to allow the invoker to identify the reply to this *AsynchInvoke* when it comes. Second, the calling EPL process waits on a condition variable.

At some later time, the *Dispatcher*, a separate EPL process provided by the Eden library, receives the invocation reply message via

ReceiveMsg(WasPresent, NewHandle, ResultESCII, NewStatus)

Noticing that *WasPresent* is true and that *NewHandle* matches the *OutHandle* of an EPL process currently blocked within *SynchInvoke*, the *Dispatcher* process calls a monitor entry to store *ResultESCII* and signal the condition for which the user process is waiting. The user process, still executing within library code, now completes its *SynchInvoke* call, unpackages its *LoginDirectory* result parameter from *ResultESCII*, completes the original *Lookup* call, and returns to the user's program.

We now will walk through the same call, this time from the perspective of the target. Recall that within the target's EPL program was the declaration

```
Invocation procedure Lookup(
    CallersRights: EdenRights,
    SearchKey: String,
    var Result: Capability,
    var Status: EdenStatus) =
begin
. . .
end Lookup
```

The user must also explicitly provide an EPL process to execute the invocation. It does this by using two facilities of the synchronous level.

```
process Reader2
begin
    var MyHandle: InvocationHandle
    loop
        ReceiveSpecific(MyHandle, "Lookup")
        CallInvocationProcedure(MyHandle)
    end loop
end Reader2
```

The call to *ReceiveSpecific* enters a monitor to record the operation that the process is willing to serve, and then awaits on a condition variable until a request for this operation (*Lookup*) arrives. Recall that within each Eject there is a special EPL process, the *Dispatcher*, which executes

ReceiveMsg(WasPresent, NewHandle, ArgumentESCII, NewStatus)

Noticing this time that *WasPresent* is again true, but that *NewHandle* is an invocation request (handles are tagged with the function of the message), the *Dispatcher* enters the monitor to store the *NewHandle* and *ArgumentESCII*. Finding *Reader2* blocked waiting for a *Lookup*, the *Dispatcher* unblocks this process, which returns from *ReceiveSpecific* and calls *CallInvocationProcedure*. This procedure is especially constructed by the EPL translator to retrieve *ArgumentESCII*, unpackage the operation name and value parameters from it, check them for agreement with the invocation procedure header, and make an ordinary call on the invocation procedure *Lookup*. Upon return from *Lookup*, *CallInvocationProcedure* packages *LoginDirectory* into an ESCII, say *ResultESCII*, and executes

ReplyMsg(MyHandle, ResultESCII, Status)

This completes the call to *CallInvocationProcedure; Reader2* now is ready to receive another invocation request.

This division of labor between kernel, library, and translator support has achieved our goals. The key ideas are to use a clean

asynchronous facility provided by the kernel together with the process and monitor facilities that EPL inherited from Concurrent Euclid to provide synchronous facilities for use by EPL processes. This is only practical, however, because translator support reduces tedium.

D. History of Process and Invocation Support

During the course of the Eden project, several of our implementation ideas have changed. In this section we focus on the progression of these ideas for multiple processes within Ejects and for synchronous invocation. This progression has had two driving forces.

- Our early use of Intel's iAPX 432 and iMAX as our underlying system gave way to the use of the more conventional VAX/UNIX™ combination. This change of context rendered many of our initial implementation ideas inappropriate.
- Equally important was a shift in our attitude toward the nature of our kernel. Initially, the kernel provided both the interface to be used by Edentype programmers and the interface to be enforced at run-time. We have found that splitting these two allowed for a smaller kernel and more flexibility for programmers outside the kernel.

Our initial plans called for multiple processes within Ejects, and for these processes to use synchronous invocation [10]. We felt that it was unreasonable for a system of the 1980's to restrict its programmers to a sequential programming language, and that the most straightforward way to achieve concurrency was to use explicit multiple processes, rather than a single process and asynchronous invocation. Any implementation ideas were very sketchy.

At a very early point in the project, we decided to build Eden on an underlying system consisting of Intel iAPX 432's and iMAX. The 432 is an unusual system in many respects. Among them is that there is great coherence between the architecture, language, and system notions of process, synchronization, and address space. Thus, in providing multiple processes within an Eject, there was no issue of whether they would be language- or system-level processes—these were the same things. Our implementation plans at this stage of the project emphasized direct kernel support for the functionality seen by the Edentype programmer. Thus, for example, the function of allocating processes to incoming invocation requests was to be provided directly by the kernel. There was some disagreement among us about whether the kernel would provide an asynchronous invocation mechanism at all, since it would be easy to provide synchronous invocation directly.

With the Newark system, built on a VAX/VMS substrate, and with the current Eden 1.0 system, built on a VAX/UNIX™ substrate, we had to face many new technical issues. We knew that any process or interprocess communication facility supported by the VAX architecture would be more cumbersome than those possible with the 432. There was considerable temptation, in fact, to drop our ideas of multiple processes and synchronous invocation. The decision in Newark, however, was to provide each Eject with multiple processes supported directly by VMS, and to use the VMS notion of Mailboxes for interprocess communication both between Ejects and among processes within an Eject. However, while synchronous invocation was supported directly by the kernel, the allocation of processes to invocation requests was performed by libraries of user-level code. Performing the dispatcher function at this level simplified the design of the Newark kernel, and also allowed us to experiment with several different dispatching techniques. Edentypes were programmed in Pascal and programmers had to build invocation messages "by hand" using Pascal records, which proved extremely tedious. Our limited experience using Newark convinced us that multiple processes and synchronous invocation were worth retaining, but that we would have to find better implementations for them.

As we evaluated the Newark experiment in preparation for the design of Eden 1.0, we became aware of the Concurrent Euclid language and decided to try to use both CE processes and UNIX™ processes in our new design. Our experience in performing the dispatcher function at the user level in Newark made this possibility a natural one to consider, and the obvious advantages in reduced overhead vis-a-vis the uniform use of VMS processes in Newark were clear. The essential outline of the techniques described earlier came quickly. We were determined to make the invocation mechanism easier to use than it had been in Newark, so the EPL effort was given equal importance with the implementation of the kernel. The Eden 1.0 kernel and EPL translator both were available for use during the spring of 1983. The simplified kernel design made possible by providing only asynchronous support at that level eased the kernel implementation effort; also, since the work needed to implement synchronous invocation was spread between the kernel processes and the kernel library, we were able to approach these tasks separately. The presence of EPL greatly eased the task of programming Edentypes.

In summary, we were driven in the same direction by the technical implications of using a conventional substrate and by a shift in attitude towards having the kernel provide only necessary primitives rather than the complete user-visible interface. We believe that the current design is sound in our current VAX/UNIX™ environment and that it would work well in a variety of other conventional environments.

E. Support for Saving and Restoring State

Ideally, an Eject is an active entity with permanent state. In reality, an Eject has two manifestations: an active form (with its system-level process) and a passive form (consisting primarily of a disk file). All invocations are implemented by the active form, but only the passive form can survive a crash. This section discusses how the use of a passive form allows the Eject to approach the ideal mentioned above. Fig. 2 shows how an Eject acquires and loses active and passive forms.

When an Eject is created, only an active form exists. It can therefore execute and engage in invocations, but has no state on permanent store. If it were to Deactivate, or if it (or its computer) were to crash, it would vanish, and could never be invoked again. If another Eject with a capability for this vanished Eject were subsequently to attempt to invoke it, the invocation would fail. (However, the kernel does not make any attempt to remove such "dangling" capabilities.)

An active Eject can execute a checkpoint sequence, in which

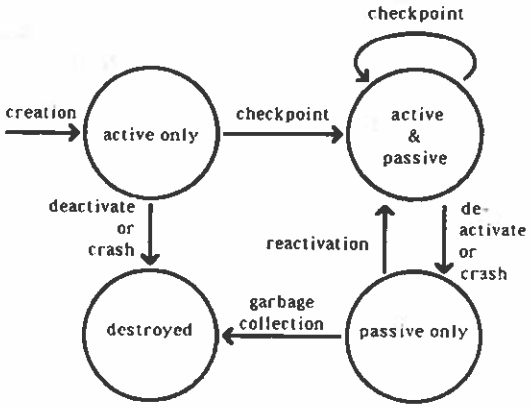


Fig. 2. Transitions involving active and passive forms.

it opens a passive form, writes its state in a series of *PutData* calls, and then completes the passive form with a *Checkpoint* call. Once the Eject has written its passive form, it has identity and state on permanent store. If it subsequently *Deactivates* or crashes, its active form will vanish, but the passive form will remain.

If an Eject has a passive form but no active form, and another Eject invokes it, the Eden kernel will reactivate it, i.e., will construct a new active form which will receive the invocation. This new active form can query the kernel to discover why it was activated. On finding that it is a reincarnation, it can reinitialize its state by executing a restore sequence, in which it opens its passive form for reading, reads its state in a series of *GetData* calls, and then closes the passive form.

Once an Eject has a passive form, it will be deallocated only by an Eject garbage collector, which runs periodically to discover Ejects that are not reachable by any path. Currently our garbage collection algorithm is sequential and runs only when no Ejects are active. Future work will consider distributed garbage collection algorithms that can execute concurrently with active Ejects.

The Eden kernel attempts to implement *Checkpoint* atomically using the UNIX™ *mv* call to indivisibly change an entry in the UNIX™ directory of passive forms. Unfortunately, the presence of the UNIX™ disk cache makes this inefficient and, in the final analysis, impossible.

F. Kernel Implementation

The Eden kernel on each machine is implemented as a UNIX™ process which communicates with Ejects via IPC. The kernel process has several functions.

- It creates and manages the UNIX™ processes used to implement Eject active forms.
- It maintains certain parts of each Eject's state that must be protected against misuse or error by the Edentype programmer. (An important example is the table of capabilities owned by the Eject.)
- It also maintains a cache of the locations of remote Ejects and of the status of other Eden machines to aid in the mapping of capabilities into physical locations.
- It implements a set of operations used by Ejects, including the asynchronous invocation facilities, Eject creation, and checkpointing.

In addition to the kernel process there is a kernel library—a set of (unprotected) procedures that is linked within each Eject. This library makes kernel operations available via a procedural interface. It also performs many operations directly, when their implementation does not require access to the tables maintained in the kernel process. For example, the *CapaEqual* operation compares two capabilities for equality, and does not require that these capabilities be validated by the kernel process. Operations implemented within this library obviously can be accessed with much less overhead than those implemented within the kernel process, but this comes at the cost of making the kernel library quite large, thus swelling the size of Edentype executable images.

One novel aspect of the Eden kernel is its technique for storing and protecting capabilities. Most object oriented systems store C-lists (vectors of capabilities) in the kernel's address space and allow the user to denote capabilities only by indexes into the C-list [11]. We considered this approach, and rejected it for several reasons.

- It would require the overhead of communication with the kernel to examine the fields of the capability (e.g., its access right field).
- It creates confusion about the notion of a value of type Capability in EPL. Would two such values be equal only if they were the same C-list index, or also if they refer to equal capabilities in different indexes?
- Similarly, when making a copy of an EPL variable of type Capability, should the programmer copy the capability into a different index, or simply copy the index value?

Our approach is to let each Eject keep capabilities in its own address space, but to have the kernel maintain a (protected) table containing copies of these capabilities. Entries are added, for example, when a new capability arrives as an invocation parameter or when the Eject creates a new Eject. Any capabilities used by the Eject (for example, in an invocation) are checked for existence in the table. To simplify matters, only one entry is kept for each capability within the Eject's state (no matter how many copies the Eject may have), and the union of the access rights of known capabilities is stored with this entry. The problem of ever deallocating entries in the table remains; this is solved by allowing the Eject to occasionally enumerate the capabilities known to it. This enumeration could in principle be automated by the marking phase of an EPL garbage collector. Even in the absence of such a collector, however, we find this enumeration less of a burden than keeping track of C-list indexes would be.

The high overhead of communication between Ejects and the kernel process severely limits Eden's performance. This clearly is one of the disadvantages of building on top of an existing system. On the other hand, implementing the kernel by modifying the UNIX™ kernel would have been both more difficult and less portable.

The Eden kernel processes on the various machines communicate via a simple datagram protocol implemented on the Ethernet. Higher level TCP/IP protocols are used only to

move executable Edentype image files and passive representations from machine to machine.

G. Interfacing Eden to the Outside World

So far we have described how Ejects are invoked by other Ejects, but have said nothing about how a user sitting at a terminal can make an invocation or create an Eject. These facilities initially were provided by escaping to the underlying operating system, and have been incorporated into Eden only recently. This evolution illustrates the way in which we have taken advantage of the presence of UNIX™ to limit the amount of Eden software which had to exist before Eden could begin to be used.

Since Ejects are implemented as UNIX™ processes, they can use UNIX™ services to communicate with non-Eden files and programs. We take no steps to prevent this in the current prototype. Each of our early applications contains one or more "bridge Ejects" which communicate with the user's terminal using UNIX™ I/O and an application-specific command language, and which communicate with the application proper using invocation. This still leaves the problem of arranging for an Eject to communicate with the standard input and output files available in the UNIX™ environment. This facility is provided by *becomej*, a UNIX™ program that creates an Eject as its own child process rather than as a child process of the Eden kernel. Some examples of bridge Ejects will be found in Section III-A. It should be emphasized that very few of our Ejects actually use UNIX™ in this way.

More recently, two Ejects have been constructed that encapsulate the functions of these bridge Ejects. The first is a TerminalHandler Edentype, which should eventually be the only Eject run with *becomej*. The TerminalHandler allows Ejects to open windows on the screen and to use them to fetch input from and to perform output to the user; all communication between the client Ejects and the TerminalHandler is via invocation. Using the terminal handler, a number of Ejects can share the resources of a single terminal in a way that is very natural in an object oriented environment.

The second Eject is the Eshell, which implements a command language for Eden based on a subset of the Eden Programming Language. The Eshell runs as one of the TerminalHandler's clients and allows a user sitting at a terminal to make arbitrary invocations of any Eject for which a capability is possessed. The generality of the Eshell makes it invaluable for testing, debugging, and other systems work, but inappropriate for interfacing with applications that need a user-friendly interface. It therefore seems likely that particular applications may continue to have special-purpose interfaces that will be started from the Eshell and will communicate with the user via the TerminalHandler.

III. Building Applications in Eden

The previous sections of this paper described the facilities that Eden provides for the applications programmer. This section illustrates the utility of these facilities by showing how they help in the construction of distributed applications. We will survey four applications and then use them to provide insight into why some of our design choices seem to have been good, and why others may need to be reconsidered.

A. Four Distributed Applications

Edmas—The Eden Distributed Mail System: Edmas was one of our first applications and thus is one of the most mature. It is hardly surprising, therefore, that many of our early opinions of Eden are based on experience with Edmas. The description below is deliberately superficial; a more complete one appears in [12].

Edmas is constructed from two principal Edentypes, the MailBox and the MailMessage. The user interface is implemented using two further Edentypes and some UNIX™ utilities. One pre-existing Edentype, the Directory, was reused. Subsequent work involved adding the DistributionList Edentype.

Imagine that a user wishes to send a message using Edmas. First, the text of the message is composed using the Emacs editor. When the user is satisfied that the message is complete, the *send* key is pressed on the terminal, causing the message text to be passed to a MailSendInterface Eject. This bridge Eject is started by *becomej;* it communicates with Emacs by using UNIX™ I/O facilities, and with Edmas by invocation. Another function of the MailSendInterface is to translate the addresses typed by the user at the keyboard (character strings) into capabilities for addressees' MailBoxes; this is accomplished by performing *Lookup* invocations on a Directory Eject. The MailSendInterface then creates a new MailMessage Eject, and by a series of invocations initializes the various fields of the MailMessage. Among these is the *From* field, which contains a capability for the sender's own MailBox and is used by the *Reply* facility. When the MailMessage is completely composed, the MailSendInterface makes a *Deliver* invocation on the MailMessage. This has two effects: the MailMessage becomes *frozen*, meaning that it will reject subsequent modifying operations, and the MailMessage delivers itself to the addressees. The work of the MailSendInterface for this message now is complete; it may be deactivated or used to send further messages.

Mail delivery is accomplished by the MailMessage Eject performing a *Deliver* invocation on each addressee, i.e., on each capability on its *To* list. The capability for the MailMessage itself is passed as a parameter to the *Deliver* invocation. If the addressee is actually a MailBox Eject, the effect of this invocation is simply to enqueue the capability of the MailMessage in the MailBox. The owner of the MailBox can interrogate it for unread mail using Emacs and a bridge Eject of Edentype MailReceiveInterface.

The addressee also may be a DistributionList Eject. In this case the effect of the MailMessage's *Deliver* invocation is to cause the DistributionList to deliver the message to each of the addresses on the list, which may be either MailBoxes or other DistributionLists. If they happen to be DistributionLists on different nodes, then there is true parallelism in the delivery. Since delivery to a large list may take some time, the DistributionList replies to the initial *Deliver* invocation before starting to deliver the message.

The Eden Transput System: One of the interesting aspects of

Eden is that facilities that would have to be "built in" to conventional systems need not be part of the Eden kernel, but can be added as applications. Two interrelated examples of such facilities are the Transput (input and output) System and the Eden File System (EFS).

In a modern operating system, an application will wish to perform transput both to physical devices, such as printers and terminals, and to other applications, such as printer spoolers and screen managers. In fact, when an application is written, it is often not known whether it will be used with a terminal or screen manager; to achieve device independence, both kinds of transput should be done in the same way. In UNIX™, for example, output is performed using the *write* system call. This is true both when the data are sent to another program and when they are sent to a file; only the program that makes the initial connections has to know which case obtains. It is the job of the operating system nucleus to either write the file, or to buffer the output until the receiving program does a *read*.

Although Ejects may use all the richness of invocation for their communication, at least in some circumstances a simple byte-stream transput system is required; reading from hardware devices is an obvious example. In Eden such devices will be represented by standard Ejects, and the Eden kernel is not required to be a party to their use. It would be pleasant if all devices accepted the same set of invocations, as this would allow device independent transput. In practice, the characteristics of devices vary, but we feel that the interface presented by all devices should include a basic byte-stream protocol whenever this is reasonable.

This idea is naturally extended to Ejects that are not device drivers, but which nevertheless produce output, accept input, or do both. It is then possible to construct pipelines of Ejects that act as filters, using the terms in the UNIX™ sense.

One naive approach to constructing a transput protocol would be to define both *Read* and *Write* invocations. The difficulty is that this provides no simple way for an Eject performing a *Read* invocation to obtain data from another Eject performing a *Write* invocation. It is necessary to interpose an Eject between them whose function is to accept *Write* invocations, store the data in a buffer, and service *Read* invocations using the buffer. These buffer Ejects have a cost, but serve no real function. One way of eliminating them is to use *Read* invocations but no *Write* invocations: an Eject wishing to perform output simply waits for another Eject to perform a *Read* on it.

This idea is discussed in more detail in [13]. We have constructed a transput library for EPL which provides the user with conventional *Get* and *Put* primitives for basic data types, while maintaining the "read only" protocol between Ejects. We also have written two Edentypes, UnixFile and UnixFileS, which together constitute a "device driver" for UNIX™ files, i.e., they allow other Ejects to access UNIX™ files by means of invocation. The TerminalHandler Edentype mentioned in Section II-G provides a similar interface to a user's display.

The Eden File System: Conventional sequential files are not part of the Eden kernel; a sequential file is simply an Eject that allows its contents to be read and written using the Eden Transput System. The *Checkpoint* facility allows such an Eject to maintain its contents on stable storage.

However, in the same way that stream transput is not the only communication mechanism within Eden, sequential files are not the only form of long-term storage. All Ejects, once *Checkpointed*, are equally permanent. A means is required for keeping capabilities for them. The Edentype Directory does just this. A capability for any Eject can be associated with a mnemonic string and stored in a Directory, and can later be retrieved by performing a *Lookup* invocation on the Directory with the string as argument. Since Directories can be entered into (other) Directories, an arbitrary directed graph of Directories can be built up from a single root.

The design of the Eden File System (EFS) is described more fully in [14]. Two facilities that it provides for the user of the sequential file are versions and transactions. If a user wishes to keep several versions of a document, rather then entering several files in a directory, he enters a single VersionManager Eject therein. When a client Eject opens an output stream to the VersionManager, a new sequential file is created that reads from the client Eject. When the file is completed it becomes the latest version known to the VersionManager. Opening a VersionManager for reading will normally provide a stream from the latest version of the file.

If the client wishes to update several files atomically, it first creates a TransactionManager Eject. It then asks the TransactionManager to open each of the VersionManagers that are to participate in the transaction. When the new files are closed, they are not immediately installed as new versions; instead their capabilities are passed to the TransactionManager. If the client asks the TransactionManager to abort the transaction, it simply *Deactivates* itself: nothing in the file system has yet been changed. If the transaction is to be committed, the TransactionManager performs a two-phase commit with the VersionManagers. In effect, the TransactionManager acts as the commit record for the transaction.

An initial use of Eden sequential files is to provide a location independent file system for users of UNIX™. A special-purpose interface permits the contents of Eden files to be moved to UNIX™ files and vice versa.

The Eden Appointment Calendar System: Prompted by our experiences with Edmas, we have begun work on a distributed appointment calendar. One of the facilities offered by the calendar is to find a time when several people are free, and to tentatively schedule a meeting among them. The meeting is confirmed only when all participants have agreed to the suggested time.

The scheduling of a meeting may be viewed as a transaction, which may commit or abort depending on the responses of the people involved. These transactions may potentially be long-lived, and locking the whole of each calendar until a tentative meeting is confirmed or canceled is unacceptable. Instead the Calendar System uses one Eject to represent each event, in addition to an Eject for each calendar. The Calendar Edentype is basically a list of capabilities for Event Edentypes, together with hints as to the times of the Events and whether they are tentative or definite. Considering the scheduling of an Event as a transaction, the Event Eject acts as its own commit record.

B. Ejects as an Abstraction Tool

Having sketched four early applications built in Eden, we now begin our discussion of how these and other applications make use of the Eden architecture.

Because of the object oriented nature of Eden, the fundamental abstractions of the application can be represented directly in the implementation. For example, we felt that the natural abstractions to use when designing Edmas were mail boxes and mail messages, and we were able to implement these abstractions as Ejects; in more conventional systems we would have had to simulate them using files or segments of files. There is also a clear separation between the interface part of Edmas and the transport and storage part; again, this is realized by using separate Ejects to encapsulate the interfaces.

One of the questions that worried us when designing Eden was the overhead inherent in our notion of Eject. Although both Ejects and the abstract data types of a programming language can be used for encapsulation, Ejects would clearly be orders of magnitude more expensive. Would this difference in cost mitigate against their use? It seemed clear that, unlike Smalltalk objects, Ejects would not subsume programming language data types, and there would therefore be an alternative abstraction facility in Eden. However, we felt that when data were inherently shared, or had to be protected against failures, Ejects would be invaluable. These feelings have been confirmed by our experiences so far. MailMessages are not large, but representing them as Ejects allows them to be accessed in a location independent manner and to be shared among their addressees. Calendar Events are even smaller—just the size of a typical EPL record—but because they are inherently shared, it is still convenient for them to be separate Ejects. There is a world of difference between a group of people attending a meeting, and each person attending separate meetings that happen to occur at the same time; the sharing of Events captures the difference very nicely.

C. The Programmability of Eden

One of the most remarkable things about Edmas is the rapidity with which it was constructed, even though the debugging of the first Eden kernel proceeded in parallel with the testing of Edmas. The system was designed and specified in a week, and demonstrated to an external visitor after six weeks of work by two graduate students who were also taking a full class load. We attribute this speed to two factors. The obvious one is that the most complicated part of conventional mail systems, the transport layer, does not exist in Edmas as far as the applications programmer is concerned. Because the facilities of the Eden kernel allow MailMessages and MailBoxes to be addressed in a location independent way, most of the distribution aspects simply disappear. (In this respect a mail system may be an unfairly favorable application, although this only occurred to us in retrospect.) The second factor is that even before the Eden kernel was running, a prototype implementation of the Eden Programming Language (EPL) was available.

The significance of EPL is not that it extends the state of the art of language design, but that there is a careful matching between the concepts of Eden and the structures of EPL. Eden invocation is a simple concept; the challenge was to make its realization in EPL equally simple. On the invoking side, the programmer sees an ordinary procedure call with an additional *Status* parameter. On the invoked side, the programmer writes a procedure body, again quite ordinary except that it is designated as an invocation procedure and that certain parameters must be present. For example, the following code appears in the MailBox Edentype, associated with the invocation that gives a MailBox a printable name.

```
invocation procedure SetName(
        CallersRights: EdenRights,
        Appellation: String,
        var Status: EdenStatus) =
    Imports ({modules}     StatusDefs,
             {monitors}    var PrintName,
             {procedures}  PerformCheckpoint)
begin
    if (AdministratorRts not in CallersRights) then
        Status := StatusDefs.EPLInsufficientRights
    else
        PrintName.SetName(Appellation)
        PerformCheckpoint
    end if
end SetName
```

Its action is fairly obvious. Notice that access rights are checked by the Eject itself, not by the Eden kernel. *PrintName* is a monitor which protects the appropriate field of the MailBox from conflicting processes. Once the name has been set, it is necessary to *Checkpoint* the MailBox, so that the new name survives crashes.

That the automatic packaging and type-checking of invocation parameters is a significant aid becomes apparent only when the facilities of EPL are inadequate, as they were in one place within Edmas. The Eden kernel defines a *Timestamp* data type to represent moments in history. When a MailMessage is asked (via invocation) for the date and time of its composition, the reply to the invocation naturally contains a *Timestamp*. Unfortunately, when the mail system was first written EPL did not permit *Timestamps* as parameters, so programmers were forced to convert them into raw bytes and back again. The inconvenience of this, together with the risk of error, stood in sharp contrast to the ease with which parameters of the supported types were passed and type-checked.

D. Multiple Processes in Edmas

Another feature of Eden that paid dividends within Edmas is the provision within an Edentype of multiple processes and synchronous invocation rather than a single process and asynchronous invocation. The latter would have been easier to implement, but we felt that the most straightforward way of achieving parallelism was to use explicit multiple processes. Additionally, we felt that it was unreasonable for a system of the 1980's to restrict its programmers to a sequential programming language.

Typically, even the simplest Eject (such as a MailMessage) has three processes: one that handles incoming invocations, one that *Deactivates* the Eject after it has been idle for a certain amount of time, and the *Dispatcher* process described in Section II-C. Even in this simple case, it sometimes is possible to increase throughput simply by providing two (identical) invocation handling processes.

DistributionList Ejects use a more complicated process structure. Since DistributionLists may refer, directly or indirectly, to themselves, some care is necessary to ensure termination and avoid deadlock. The former problem is solved by the use of "hop counts," a well-known notion in graph

traversal [15], and is optimized by the use of a cache. Deadlock can occur if in handling a particular invocation an Eject makes another invocation, possibly of itself. The programmer must then ensure that there is a process available to deal with the new invocation, even though the invoking process is suspended. Sometimes this can be achieved by partitioning the set of invocations into two classes, those whose handling requires external invocations and those whose handling does not, and using a separate process for each class. Sometimes this particular partitioning is inadequate. For example, the code for handling Deliver invocations within a DistributionList may itself make Deliver invocations on DistributionLists. If one of these invocations is (directly or indirectly) recursive, having a single process deal with all Deliver invocations will lead to deadlock. The solution adopted in this case is to divide the work between two processes. One services an (unbounded) queue of outgoing *Deliver* invocations, while the other handles incoming *Deliver* invocations by placing requests on the queue. When proving freedom from deadlock, the existence of several process classes is crucial, although the number of processes in each class is not (provided that there is at least one).

E. Checkpointing and Recovery

The *Checkpoint* mechanism, while simple, is quite satisfactory for Edmas. MailMessages *Checkpoint* once: when their composition is complete, but before they begin to deliver themselves to their addressees. MailBoxes *Checkpoint* whenever a message is delivered to them and whenever a message is removed. Since the state of a MailBox is just a set of capabilities, the data that must be transferred to disk during these *Checkpoint* operations is small—a few hundred bytes for a MailBox with 20 unread messages. If a MailBox chances to crash after the MailReceiveInterface has picked up a message, but before that message is removed, the user will see the message twice. We did not think this a serious shortcoming; had we, it could have been overcome using atomicity techniques (discussed in a subsequent paragraph).

An inherent limitation of *Checkpoint* becomes evident when an Eject with a lot of state needs to make a small change. For example, suppose that the editor that is used to compose the text of a MailMessage were an Eject rather than a UNIX™ program. To prevent loss of data in a crash, the editor ought to *Checkpoint* its buffers frequently. Using the current primitive, this would involve writing *all* of the editor's state to disk at frequent intervals, which would be prohibitively expensive. What is needed is a facility that inexpensively allows small changes in the Eject's state to be made permanent. There are various ways in which this might be achieved, including multiple *Checkpoint* images for each Eject, a logging facility, or the recording of the invocation history [16], [17]. We aim to design such a mechanism in the future. At present, the only way to contain the cost of *Checkpoint* is to use more than one Eject. For example, one could introduce a separate Eject to represent each of the buffers in an editor. While this might be an appropriate way to structure an editor, it is unsatisfactory to be forced into such an arrangement just because of the inadequacy of *Checkpoint*.

Another problem with *Checkpoint* as a crash recovery primitive is exposed when one considers a pair of Ejects that must be synchronized. As an illustration, consider an Eject representing a sequential file, and a client Eject that has opened a stream to *Read* from the File. Within the File Eject there must be a counter indicating how far the client has progressed. Consider what happens if this counter is not checkpointed on each *Read*. If the File Eject crashes but the client does not, the next time the client performs a *Read* invocation the File will be reactivated but the counter will be wrong. Not only is checkpointing on each *Read* expensive (because it involves writing the whole of the file to disk, as well as the counters), it is by itself insufficient to maintain synchronization. This is because a reply message can be lost, and the client thereby persuaded to repeat the invocation. (This problem can be overcome by augmenting the invocation protocol between file and client so that *Read* requests contain sequence numbers, or so that *Read* invocations alternate with *Acknowledgment* invocations.)

A significant difference between Eden and some comtemporary designs is that Eden does not build "atomic actions" into the system kernel. The synchronization problem mentioned above can be solved in both Argus [5] and Clouds [6] by use of system-provided atomicity. In Eden, we have taken the view that applications requiring atomic transactions must build them out of Ejects and invocation. EFS and the transactions associated with Calendar System Events are examples of such applications. (Note that they have greatly differing transaction granularity.) We plan to investigate this area more. At the very least, Eden seems to be a good vehicle for experimenting with different techniques for implementing transactions. By embedding atomicity in the kernel we would have stifled this experimentation.

F. Storage Management in Eden

The problem of storage management appears in Eden in two different contexts. The EPL run-time system manages a heap used by the EPL data structures that are allocated with the *TypeName.New* primitive. The Eden kernel manages a heap used by Ejects themselves when they are created using the *Kernel.Create* primitive. Currently, the storage is reclaimed from the first heap by means of explicit deallocation, and from the second heap via garbage collection.

The conflict between these two management techniques is an old one. Garbage collection is more convenient for the programmer; explicit deallocation for the implementor. It is also argued that garbage collection is unnecessarily expensive because the garbage collector must determine by exhaustive search information that the programmer knew: which objects may be freed.

It is our contention that any object oriented system must provide garbage collection if it is to support the construction of a wide range of applications. In Eden there is no notion of ownership of Ejects, and it is hard to see how one can be introduced. Without such a notion, who should perform the explicit deallocation? A MailMessage, for example, is created by the MailSendInterface, and delivered to the (several) MailBoxes of the addressees. When the addressees read the mail they may wish to file it away in some folder, send a reply

message that refers to the original message, or forget about the message completely. Who should be responsible for deallocating the MailMessage? Who can know when the last reference to it has been lost? The complexity of trying to keep reference counts correctly even for this single application is an indication of why we consider garbage collection to be essential. In effect, Edmas uses Ejects as a system-wide database of messages, and relies on the system to return a message corresponding to a given capability. Without Eject garbage collection, the designers of Edmas probably would have had to build their own database, and to invent their own message identifiers. Not only would this have required substantial effort, but it would have severely limited the flexibility with which other Ejects could refer to MailMessages.

Even within the more restricted world of a single Eject, the lack of garbage collection within EPL is proving to be a problem. Often the programmer does not know when data structures are no longer needed. One of the reasons for this is the heavy use of layering within the code of an Edentype. As an example, consider the *CallInvocationProcedure* routine described in Sections II-B and C. If one of the arguments to the invocation is of a data type represented on the heap (e.g., a string), *CallInvocationProcedure* must allocate a new structure and initialize it with the data from the invocation message. Unfortunately, *CallInvocationProcedure* cannot deallocate this structure because the code of the invocation procedure, written by the Edentype programmer, may have linked it into some global data structure. Of course, one could make rules that forbid the Edentype programmer from doing such a thing, or require that all (and only) those structures that are not referred to from a more global context be explicitly deallocated by the invocation procedure. But doing this would negate many of the advantages of information hiding and data abstraction. The Edentype programmer would immediately have to know which data types are represented on the heap, which assignments cause copying and which sharing, and a host of other such details that are (or should be) secrets of the implementation.

The simplest solution from the point of view of the programmer is to implement a full garbage collector for EPL. At present we simply allow Ejects to grow slowly in size, knowing that eventually they will *Deactivate* themselves and the kernel will reclaim all of their address space. This arrangement is tolerable since garbage does not find its way into an Eject's *Checkpoint* image.

G. Types and Type Checking in Eden

There are several notions of type in Eden. Within an Eject there is the notion of type supported by EPL: both values and variables are typed, type equivalence is required for most operations, and new encapsulated types can be created using the module construct. Two types are equivalent only if they can be traced to the same definition in a common module: mere textual identity is insufficient. This rather conventional notion of type raises difficulties when one considers the problem of passing typed values from one Eject to another, i.e., from one EPL program to another. One would like to require that both the sender and receiver of a value agree on its type. But the existing concept of type cannot be applied because the same module cannot exist in different Edentypes. Of course, a copy of the module can appear in each Eject, but that does not imply equivalence of the EPL types they export.

The alternatives are either to define a new notion of abstract type equivalence at the system level, perhaps by using the system-wide name space of Eden, or to use a weaker notion of type-checking for invocations, such as structural or textual equivalence. So far we have avoided this problem by permitting only a small set of system-defined types to be passed across invocations.

At the inter-Eject level, broader notions of type exist. The piece of EPL code that is executed by a particular Eject is called its *concrete Edentype*, or often just its Edentype. Many Ejects of the same Edentype may exist at one time; for example, there may be many MailBoxes, all of which have the same concrete Edentype (indeed, all the MailBoxes on a particular node machine share the same code). From the outside, i.e., to some invoking Eject, the concrete Edentype of another Eject is irrelevant. It is the *behavior* of an Eject that is important to its users. Each Eject may be thought of as an abstract machine. The type-code of the Eject defines the transitions of the machine; the inputs are the invocations it receives, and the outputs are the replies to those invocations. Since this pattern of invocation and reply is all that other entities can observe about the Eject, all Ejects with equivalent state machines provide the same functionality. Because many pieces of EPL code can define the same transitions, it is quite possible for several distinct concrete Edentypes to behave in the same way. In such a case the Edentypes provide alternative implementations of the same abstract machine. If they accept the same set of operations and react in the same way, they are said to be of the same *abstract Edentype*. So, if some type programmer were to write a piece of code that defines the same behavior as the existing MailBox code, but is more efficient in some circumstances, one might like to have both the new and old concrete MailBox Edentypes coexisting in the system, and providing two distinct implementations of the same abstract Edentype. One important application of this idea is the provision of a version to the type with enhanced debugging or monitoring facilities.

Multiple implementations can exist in Eden because there is no checking of concrete Edentypes. Capabilities are not typed at compile time, and it is not possible for one Eject to enquire of the concrete Edentype of another (when performing an invocation, or at any other time). In contrast, the Hydra system [11] did check types at the corresponding places, and thus prohibited multiple implementations of this kind. In Eden, if the invoked Eject does not support the requested operation, or supports an operation with the same name but with a different set of parameters, the invoker receives a reply (generated by EPL) describing what has happened. In other cases, the invocation proceeds.

So far this approach has not hurt us, and has permitted the notion of behavioral compatibility to be further extended. If a client Eject M assumes that some server Eject behaves as the abstract Edentype E, then not only will M be satisfied by any implementation of E, but also by any implementation of E', where E' is a superset of E. In other words, provided that E' contains all the operations of E and that their semantics are

the same, it does not matter to M that E' contains other operations as well.

One place where we took specific advantage of this fact was in adding DistributionLists to Edmas. When a MailMessage *Delivers* itself, there is no check that the destination is actually a MailBox. So, provided that the destination accepts a *Deliver* invocation and "does the right thing" with the message, all should be well. The fact that *Delivery* to a DistributionList triggers a complicated sequence of further *Deliveries* is of no concern to the MailMessage, or indeed to any part of the system other than DistributionLists themselves. Both MailBoxes and DistributionLists implement a superset of the abstract MailSink Edentype, for which there is no corresponding concrete Edentype at all.

A similar thing happens within the Eden Transput System. The Transput System relies on the abstract Stream Edentype. However, there is no concrete Edentype that is just a Stream; there are streams from sequential files, and streams from terminals, and streams from filter Ejects, all of which conform to the Stream abstraction but which provide other services in addition.

Having several Edentypes provide the same operations is reminiscent of the type hierarchy of Simula or Smalltalk. In fact, some of our applications use multiple type inheritance in the sense of [18]. However, whereas Simula and Smalltalk automate inheritance in the language processor or run-time system, Eden implements it by convention; the association of abstract with concrete Edentypes is not directly supported by EPL or the Eden kernel.

It remains to be seen whether these conventions will prove inadequate as the number and variety of Edentypes and Edentype programmers grow. If they do, there are various schemes that would enable a programmer to enquire about the abstract Edentype of another Eject. One such scheme that can be implemented entirely outside of the Eden kernel is to create a "specification Eject" for each abstract Edentype, and to have each concrete Edentype tell its invokers its abstract Edentype. The specification Ejects can be connected in a directed graph, the edges of which indicate that one abstract Edentype is a subset of another. An invocation would be type-correct if the abstract Edentype expected by the invoker could be reached by following the edges of the Edentype graph starting at the abstract Edentype of the invokee.

We have also given some thought to automating the Edentype inheritance scheme in a way that is similar to that described by Borning and Ingalls for Smalltalk. This would enable code from one Edentype to be automatically inherited by another, so that changes in the first Edentype would be reflected in the beneficiary. Whereas Smalltalk has to deal with inheritance of procedures and data structures only, in Eden the problem is compounded by the need to inherit processes as well. As the above discussion of deadlock avoidance in the mail system should make clear, allocation of invocations among processes is sometimes a nontrivial task, and it is difficult to see how an automatic inheritance scheme would cope with it. Even adding a new process to an Edentype must be done with care because data structures that previously were accessed by only one process may need to be protected by monitors.

IV. Evaluation

In earlier sections we summarized the Eden architecture, described how this architecture is supported, and illustrated the use of this architecture by certain applications programs.

We noted in Section I that the scientific questions we must explore involve assessing the benefits (in terms of programmability) and the costs (in terms of necessary support) of our architecture. This assessment has just begun. In this final section we state our preliminary results.

The presentation is divided into five parts. In the first two parts we consider the global questions just stated: benefits and costs. (Clearly we are not yet in a position to fully answer either of these questions.) In the final three parts we evaluate various specific choices that we have made. First we consider choices that appear to have been good ones, then choices that appear to have been bad ones, and finally areas in which we have too little experience as yet to hazard a guess. Each of the five parts is brief since supporting material appears in earlier sections.

A. Benefits

We are pleased with the programmability afforded by Eden. Applications such as Edmas have been brought up remarkably quickly.

The object based approach has conveyed to Eden the same benefits it brought to earlier centralized systems such as Hydra. In addition to ease in constructing applications from scratch, the reusability of Edentypes has already been demonstrated; for example, in the use by Edmas of a Directory Edentype initially developed for EFS.

The style of location independence supported by Eden also has shown itself to be a boon in various applications, most particularly in Edmas.

The construction of additional applications will allow us to test the benefits of our architecture further.

B. Costs

Invocation is slow in Eden 1.0. At the highest level—a synchronous invocation made from EPL using procedure call syntax—nearly 100 ms are required on a VAX-11/750 for the execution of a statement such as

RootDirectory.Lookup(UserName, LoginDirectory, Status)

Clearly, no system whose kernel consists of several UNIX™ processes communicating via IPC is going to be fast. Our objective for Eden 1.0 was a prototype that we could construct fairly quickly and that would be sufficiently performant to allow us to conduct experiments using it. The system meets this objective.

A key question that must be answered is the extent to which performance problems are inherent in our architecture, rather than being artifacts of our implementation. Assessing the "true cost" of supporting an architecture such as Eden will allow the benefits to be placed in perspective. This assessment is a major effort that we are just beginning.

C. Apparently Good Choices

- *The Provision of Concurrency Within Ejects:* Many programming problems are solved most naturally using cooperating

sequential processes. Multiple processes are used in even the simplest Ejects. Their existence allows us to make invocation appear as a synchronous operation, a simple and familiar interface. In more complex Ejects, multiple processes allow deadlock to be avoided even when invocations may be recursive.

- *The Support of This Concurrency by Means of EPL:* Because the processes and monitors seen by the EPL programmer are implemented by the language rather than by the operating system, the overhead is very low, and the programmer feels free to use processes generously.

There is an interesting contrast between the world within an Eject and the world of Eden at large, where processes (i.e., Ejects) enjoy the protection and overhead of separate address spaces, and communication is by explicit message passing (i.e., invocation). We might hypothesize that the very kind of independence that is essential for Ejects (in order to deal with, for example, crashing of only part of the system), and the kind of communication that must be used when the hardware connection is a network, are inappropriate for use "in the small."

- *The Support for Invocation Provided by EPL:* One of the principal lessons of Newark was the need for syntactic support for invocation. In Eden 1.0 there is a careful match between the concepts of Eden and the structures of EPL. The role of EPL in making invocations as easy to perform as procedure calls and in automatically packaging and type checking invocation parameters is a significant contribution to the programmability of Eden.
- *The Typing of Invocation Parameters But Not of Capabilities:* We have noted that from the outside, i.e., to some invoking Eject, the concrete Edentype of an Eject is irrelevant. It is the behavior of an Eject—the invocations it accepts and the responses it generates—that is important to its users. Thus, although the parameters of an invocation are typed (via ESCII), capabilities are not. This approach has allowed us, for example, to add DistributionLists to Edmas in a straightforward manner.

D. Aparently Bad Choices

- *The Semantics of Checkpoint:* The Checkpoint primitive in Eden replaces, rather than selectively updating, the passive representation. Checkpoint shows itself to be inadequate when a large Eject needs to make permanent a small change. There are various ways in which this might be handled efficiently; we aim to design and implement one in the future.

E. Open Questions

- *One Eject, One Capability:* Eden adopts the simple notion that each Eject has one capability to which invocations can be addressed. An alternative would have been to provide each Eject with a set of capabilities. Each option can be simulated using the other; simulating multiple capabilities in Eden involves passing an additional integer argument to each invocation procedure.

Unfortunately, once an invocation is specified and implemented by several concrete Edentypes, adding an extra parameter is very inconvenient. To counter this, some of our invocation protocols require the extra integer even when it is not used. In the Transput System, for example, an Eject which is acting as a data source requires that it be told which "channel" is being read in every request for data, even if it is in fact supporting a single channel. The only way around this problem is to split every source that may service n channels into n separate Ejects. In general, the "one Eject, one capability" rule prevents us from coalescing two Ejects into one without also changing their interfaces. It may thus prove to be a barrier to the smooth evolution of applications.

- *The Active Nature of Ejects:* The protection requirements of Eden suggested that Ejects should enjoy the invulnerability of separate address spaces. The idea that each Eject should also have a separate and independent process followed because most computer architectures combine the notions of process and address space. However, there are alternatives. One is to share a single system process among all Ejects of each concrete Edentype. Another is to combine the notion of process with that of invocation, so that a single thread of control could pass through the protection domains of all of the Ejects involved in a particular task.

We cannot experiment with these alternatives in Eden; our particular choice is built into the kernel. However, we can learn something about the advantages and disadvantages of our approach. So far, we can mention that it is sometimes very convenient for Ejects to have "housekeeping processes" (processes that exist for the internal gratification of the Eject itself, rather than to service invocations). Even the simplest Ejects use housekeeping processes to observe how long has elapsed since the Eject was last invoked, and to *Deactivate* the Eject when that time reaches a certain limit. Another example of the use of such processes is in the Edmas DistributionList, which delivers mail to members of the list after the initial *Deliver* invocation has returned.

- *The Utility of Eject Mobility:* The fact that Ejects are mobile allows us to study approaches to problems such as load balancing and availability. Some of these investigations have begun.
- *Providing Location Independence in the Kernel:* Eden's experience with location independence has been positive. In fact, the ability of Ejects to perform/service invocations while being ignorant of the location of their target/invoker is central to the success of the applications cited in Section III. On the other hand, the ability of Ejects to implement robust services, or to exploit mobility for performance reasons, has suffered from the inability of an Eject to control its own location. We have developed ways of allowing Ejects, when necessary, to be explicit about the location of both active and passive forms (for example, upon creation of a new Eject). We believe that doing so, while retaining strict location independence for both partners in an invocation, will achieve the proper balance.

F. A Final Comment

Since Eden differs in substantial ways from most other systems, we have not always been able to draw directly on prior experience—our own or others'—in making architectural and implementational decisions.

On the one hand, decisions cannot be deferred indefinitely. On the other hand, decisions made in the absence of experience

often come to be regretted. Two (retrospectively obvious) lessons that we have learned concerning experimental systems work are the importance of building "throwaway prototypes" and the value of using program libraries and conventions rather than kernel code in areas of uncertainty.

For example, had we attempted to automate type inheritance early on, we might have chosen something similar to Hydra type-checking or Simula Classes. Having deferred automation, we find ourselves writing applications that use a more general notion of type inheritance.

Similarly, we did not impart "exactly once" semantics to invocation, because we were not certain of the ways in which invocation would be used. We did not build transactions into the kernel because we did not know what they would be used for. Unfortunately, we *did* build in a single state saving primitive, and this has proven to be a mistake.

Acknowledgment

Many faculty, staff, and students have contributed to Eden during its three year evolution: J. Bennett, M. Fischer, R. Fowler, T. Knight, H. Levy, and S. Vestal to the initial specification; S. Cady, B. McCord, L. Nielsen, J. Sanislo, and S. Vestal to Newark; S. Banawan, J. Bennett, S. Cady, C. Holman, D. Jacobson, E. Jul, G. Mager, L. Nielsen, J. Rees, J. Sanislo, and H. Venkateswaren to the Eden 1.0 kernel; A. Borning, J. Brower, N. Hutchinson, and B. McCord to EPL; C. Bunje, C. Holman, and D. Wiebe to Edmas; J. Brower to the Transput System; J.-L. Baer, D. Jacobson, W. Jessop, A. Proudfoot, and C. Pu to EFS; C. Binding, I. Domenech, and P. Jensen to the Eshell and TerminalHandler.

J. Sanislo deserves special mention for managing the development of Newark and of Eden 1.0. J. Browne, P. Hibbard, J. Morris, and J. Saltzer have given generously of their time and experience, participating in three technical reviews of the project. We wish we had followed more of their advice.

Numerous people offered helpful comments on drafts of this paper.

References

[1] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Reading, MA: Addision-Wesley, 1983.

[2] R. C. Holt, "A short introduction to concurrent Euclid," *SIGPLAN Notices*, vol. 17, pp. 60–79, May 1982.

[3] ——, *Concurrent Euclid, the Unix System, and Tunis.* Reading, MA: Addison-Wesley, 1983.

[4] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proc. 8th Symp. Operating Systems Principles*, Dec. 1981, pp. 169–177.

[5] B. Liskov, "On linguistic support for distributed programs," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 203–210, May 1982.

[6] J. E. Allchin and M. S. McKendry, "Synchronization and recovery of actions," in *Proc. 2nd Annu. ACM Symp. Principles of Distributed Computing*, Aug. 1983, pp. 31–44.

[7] R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," in *Proc. 8th Symp. Operating Systems Principles*, Dec. 1981, pp. 64–75.

[8] B. C. McCord and A. P. Black, "EPL programmer's guide," Eden Project, Univ. Washington, Seattle, 1983.

[9] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 36–59, Feb. 1984.

[10] E. D. Lazowska, H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal, "The architecture of the Eden system," in *Proc. 8th Symp. Operating Systems Principles*, Dec. 1981, pp. 148–159.

[11] E. Cohen and D. Jefferson, "Protection in the Hydra operating system," in *Proc 5th Symp. Operating Systems Principles*, Nov. 1975, pp. 141–160.

[12] G. Almes, A. Black, C. Bunje, and D. Wiebe, "Edmas: A locally distributed mail system," in *Proc. 7th Int. Conf. Software Engineering*, March 1984, pp. 56–66.

[13] A. P. Black, "An asymmetric stream communication system," in *Proc. 9th Symp. Operating Systems Principles*, Oct. 1983, pp. 4–10.

[14] W. H. Jessop, D. M. Jacobson, J. D. Noe, J.-L. Baer, and C. Pu, "The Eden transaction based file system," in *Proc. 2nd Symp. Reliability in Distributed Software and Database Systems*, July 1982, pp. 163–169.

[15] P. M. Merlin and P. J. Schweitzer, "Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock," *IEEE Trans. Commun.*, vol. COM-28, pp. 345–360, Mar. 1980.

[16] A. Borg, J. Baumbach, and S. Glazer, "A message system support-in fault tolerance," in *Proc. 9th Symp. Operating Systems Principles*, Oct. 1983, pp. 90–99.

[17] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proc. 9th Symp. Operating Systems Principles*, Oct. 1983, pp. 100–109.

[18] A. H. Borning and D. H. H. Ingalls, "Multiple inheritance in Smalltalk-80," in *Proc. Nat. Conf. Artificial Intelligence*, Aug. 1982, pp. 234–237.

Guy T. Almes (S'69-M'77) received the B.A. degree from Rice University, Houston, TX, in 1972 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980.

He joined the faculty of the Department of Computer Science at the University of Washington, Seattle, in 1979. His research interests include the design of distributed systems. He has participated in the Eden Project at Washington since its inception in 1980.

Andrew P. Black was born in London, England, in 1956. He received the B.Sc. (Hons.) in computing studies from the University of East Anglia, East Anglia, England, and the D.Phil. degree from the Programming Research Group of the University of Oxford, Oxford, England, in programming languages and software engineering.

He has been on the faculty of the Department of Computer Science of the University of Washington, Seattle, since 1981. His current research interests are in the area of distributed systems, programming language design, and the mechanics of program and system construction.

Dr. Black is a member of the Association for Computing Machinery, the British Computer Society, and the Union of Concerned Scientists, and an affiliate of the IEEE Computer Society.

Edward D. Lazowska received the A.B. degree from Brown University, Providence, RI, in 1972 and the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1977.

He has been on the faculty of the Department of Computer Science at the University of Washington, Seattle, since that time. His research interests fall within the general area of computer systems: modeling and analysis, design and implementation, and distributed systems.

Jerre D. Noe (S'43-M'49-SM'52) received the B.S. degree in electrical engineering from the University of California, Berkeley, in 1943 and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1948. His experience between these dates included work at the Radio Research Lab at Harvard, the American British Lab in England, and Hewlett-Packard Company.

Currently Professor of Computer Science at the University of Washington, Seattle, he has been in the computer field since 1950. At SRI International, he directed the ERMA project that developed the first computer system for the banking industry in the 1950's and served as Executive Director of the Information Science and Engineering division from 1961 to 1968. He then served eight years as Chairman of Computer Science at the University of Washington. While at SRI he was a lecturer at Stanford, and during 1976-1977 he was a Visiting Professor at the Vrije Universiteit in Amsterdam.

Dr. Noe is a member of Eta Kappa Nu, Tau Beta Pi, Sigma Xi, and the Association for Computing Machinery.