The Structure of Cedar

Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

This paper presents an overview of the Cedar programming environment, focusing primarily on its overall structure: the major components of Cedar and the way they are organized. Cedar supports the development of programs written in a single programming language, also called Cedar. We will emphasize the extent to which the Cedar language, with runtime support, has influenced the organization, comprehensibility, and stability of Cedar.

Produced in the Computer Science Laboratory (CSL) at the Xerox Palo Alto Research Center, Cedar is a research environment supporting the development and use of experimental programs, emphasizing office information and personal information management applications. Although it was clear that some unsolved problems would be addressed, the intent was to combine well-understood methods and technologies to exploit a new generation of high-performance personal computers, including the Xerox 1132 (Dorado) and Xerox 1108 (Dandelion).

The primary design objective of Cedar was to improve the productivity of experienced programmers in the production of experimental programs. An early requirements document describes the specific capabilities needed to achieve this objective [12]. Several of the more important requirements concerning the system's structure included:

- *Concurrency.* Although Cedar is a single-user system, it must nonetheless support the execution of concurrent applications. For instance, compilation, text editing, status displays, and background file updates should be able to proceed simultaneously.
- *Industrial strength.* The system must include a large virtual address space, efficient and powerful facilities for the automatic management of storage, and a rich set of program development tools (editors, compilers, symbolic debuggers, version management control). These facilities must be achieved without major performance penalties.
- *Integration.* A prerequisite for the concurrent operation of independent applications is coexistence: the applications must be able to share the lowest-level resources such as memory, files, and the display screen without disturbing each other. But the structure of Cedar should also foster the sharing of higher-level components where possible, including cooperation among applications. (For instance, events to be remembered by an automated appointments calendar might be entered from event announcements received by electronic mail.) Finally, we wanted our user interface experiments to converge to a collection of widely-applicable user interface paradigms, presenting an integrated and consistent user view. A more detailed discussion of Cedar's integration mechanisms from a program developer's viewpoint appears in a companion paper in this proceedings [13].

This paper has two major parts. Section 2 provides a comprehensive (although not exhaustive) overview of the Cedar system, including the Cedar language and the system's components. Based on this description, Section 3 discusses the overall structure of the system: its underlying philosophy, the design decisions that helped create it, and its points of similarity and difference from several other popular programming environments. Finally, we present some examples of how the structure of Cedar facilitates program development.

2. Cedar Overview

The organization of Cedar has benefited from the lessons of several rounds of implementation. Figure 1 summarizes its overall structure, as a set of major divisions each comprising a set of layered components. Each component is built upon abstractions supplied by components at lower layers in the structure. The figure was designed to express the orderings and dependencies among the components; block areas imply neither the relative importance nor the relative sizes of the components they represent.

The four major Cedar divisions are: the Cedar Machine—hardware, microcode, and primitives needed to

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM 0-89791-165-2/85/006/0230 $00.75

execute the language; the Nucleus – the operating system kernel; Life Support – the basic facilities needed for program development; and Applications – packages and tools written by and for the Cedar user community. Underlying the entire system are the facilities of the Cedar language. This section begins with a description of the important features of the Cedar language. It then follows the organization of Figure 1 (from bottom to top) to present an overview of the four Cedar divisions, emphasizing the lower three.

2.1 The Cedar Language

The Cedar language, a descendant of Mesa [14, 23, 27], is a strongly-typed systems implementation language in the Pascal family. Mesa includes facilities for modularization and separate compilation (with full type-checking across module boundaries), lightweight processes and monitors, exception handling, and first-class procedure variables. Cedar extensions retain full type-checking while providing automatic storage management and facilities for delaying the binding of type information until runtime. In addition, Cedar provides immutable strings, as well as Lisp-like lists and atoms. We begin with a reminder of some important characteristics that Cedar shares with Mesa, and then we discuss the facilities unique to Cedar.

Cedar features inherited from Mesa

A Cedar program consists of a set of separately-compiled modules. There are two kinds of modules: *interface* and *program* modules. Interface modules describe a set of types, procedures, and variables that together specify a related set of functions or a data abstraction. Interface modules are compiled into symbol tables that are used to enforce inter-module type checking, both at compile time and when modules are bound together to form a program. Program modules contain actual data declarations and executable statements. A program module that supplies the code for a public procedure or variable P *exports* an instance of P; this module is said to *implement* P. Other program modules that access P must *import* an interface describing P; these modules are *clients* of that interface.

The *configuration* is another concept inherited from Mesa; it is a separate specification of how a set of modules should be combined to form a program. The import list of each component in a configuration must be satisfied by the interfaces exported from other included components, or by a list of interfaces imported by the configuration itself. A configuration can be constructed to export (make available outside the configuration) all or only some of the interfaces that are exported by its components.

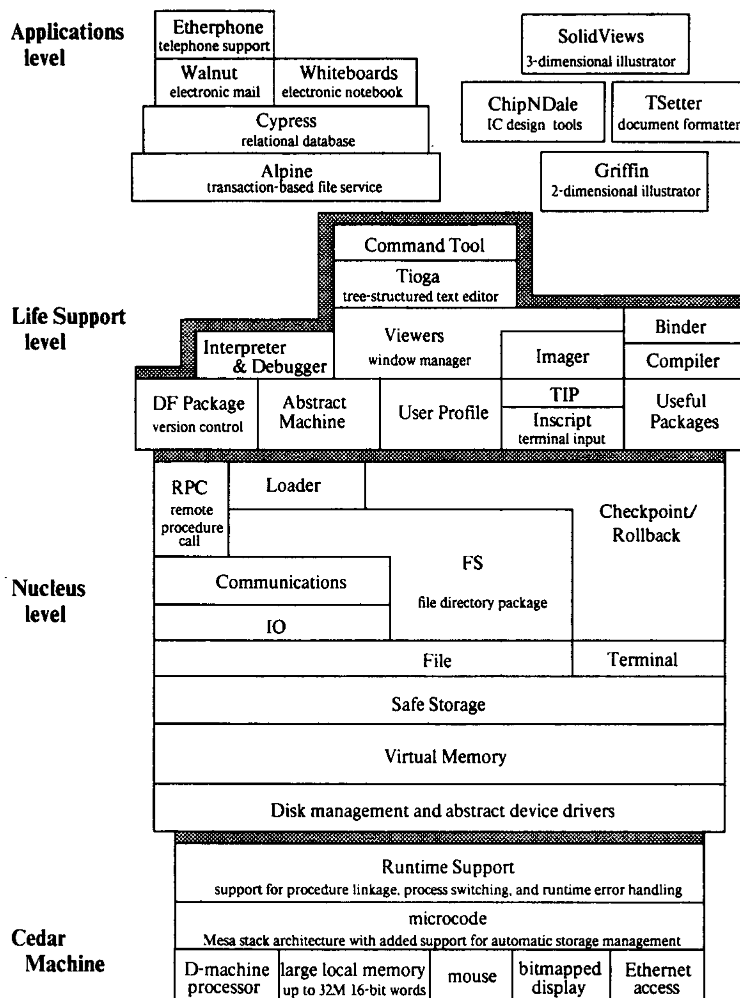

Figure 1. The Structure of Cedar.

Cedar Extensions

The extensions that distinguish the Cedar language from Mesa provide automatic deallocation of dynamic storage (supported by reference counting and a garbage collector), delayed type binding via generic pointers, and a runtime type mechanism.

Automatic storage deallocation. Cedar's storage management extensions provide *safe storage*. These changes eliminate the following two kinds of problems with Mesa's explicitly allocated and deallocated pointers to dynamic storage:

- First, the programmer must deallocate a dynamic object at precisely the right time to avoid dangling references, in which an (invalid) pointer to an object remains after the object has been deallocated, and storage leaks, in which an object becomes inaccessible without being deallocated for re-use.
- Second, invalid pointers can result from failure to initialize a pointer, incorrect pointer arithmetic, or

explicit violations of the type system through improper use of type coercions (LOOPHOLEs). Using an invalid pointer to modify memory can destroy program or system data structures in ways that are difficult to track down.

Automatic storage deallocation solves the first problem, thus making the construction of experimental programs significantly more convenient.

The *safe subset* of the Cedar language, which includes the extensions described here and a carefully-selected subset of the original Mesa language, addresses the second problem. It has been formally demonstrated that even erroneous programs written in the safe subset maintain a set of invariants that ensure the integrity of the memory allocation structures, other system data, and all code [29]. The *unsafe* features that remain outside the safe subset must occasionally be used, most often in the lower levels of the system. The additional syntax required to use them provides ample warning that the programmer is responsible for maintaining the invariants.

References, a new class of pointer data types analogous to Pascal's or Mesa's POINTER types, provide the means for safe program access to collectible storage. A reference variable, called a REF, holds the address of a collectible object of a specified data type. The system automatically initializes REFs to NIL. The operator NEW allocates a new collectible object of a specified type, with optional initialization, and returns a reference to the new object. References may be freely replicated and discarded, by assignment or by procedure parameter binding; the system releases a region of collectible storage only when no valid references to it remain. For example, the declarations

```
Node: TYPE = RECORD [leftSon, rightSon: REF Node,
        contents: CHAR];
root: REF Node;
```

declare a new variable *root* to hold nodes of a binary tree of characters, while the statement

```
root ← NEW[Node ← [NIL, NIL, 'A]];
```

allocates a new collectible object of type Node, initializes its value to the leaf "A", and stores a reference to the new object in *root*.

Because unsafe constructs are sometimes needed to write low-level system code, the Cedar language has not eliminated them. However, their use is only permitted within clearly marked procedures or blocks. Unsafe language constructs include the unsafe type escape mechanism LOOPHOLE, the original Mesa POINTERs, and address arithmetic. Furthermore, assignments to REF-containing variant records are not permitted to change the tag (choice of variant) of the record. Because the system deallocates procedure activation records when the procedure returns (i.e., there are no retained frames), nested PROCEDURE values cannot be assigned to collectible storage or to a module's global storage. When necessary, the compiler also generates code to check for other conditions that could cause illegal memory references, including out-of-range assignments to numeric variables and array index bounds violations.

The Cedar compiler verifies the programmer's adherence to the safe subset restrictions. Additional Cedar syntax controls the level of safety checking. Program blocks are specified by the programmer to be one of CHECKED, TRUSTED, and UNCHECKED. Within CHECKED blocks (the system default) only constructs in the safe subset are permitted; as a result, code in a CHECKED block can never be the direct cause of a memory smash. Use of unsafe features is allowed in TRUSTED and UNCHECKED blocks. By labelling a block TRUSTED, a programmer asserts that all uses of unsafe features within that block maintain the invariants; UNCHECKED blocks carry no such assertions. Further, a programmer specifies that a Cedar procedure is either SAFE or an UNSAFE; the body of a SAFE procedure must be either CHECKED or TRUSTED. CHECKED program blocks may call only SAFE procedures.

Delayed type binding. Compile-time type specification, otherwise known as strong typing, can catch many common programming errors during compilation and also allows the compiler to produce efficient code. However, delaying type binding until runtime can provide important program flexibility, particularly during program development. For example, programming tools such as debuggers must be able to manipulate objects of any type. The original Mesa language offers very limited capabilities for delaying type binding: the choice among predeclared alternatives of a variant record may be made at runtime, and the lengths of sequences and descriptor-based strings and arrays may be specified at runtime. Additional type flexibility in Mesa can only be achieved through use of the unsafe type escape mechanism LOOPHOLE. Cedar extensions for delayed type binding include a generic reference type (REF ANY) and a runtime type system.

A variable of type REF ANY can take on, through assignment or parameter passing, a value of type REF T for any type T. However, the actual type of the referenced object must be verified at runtime before the object can be examined or modified. Two runtime functions and some new syntax allow the use of REF ANY variables while retaining full compile-time type checking.

The boolean form ISTYPE[x, T] is defined to return TRUE if and only if the actual type of the object x is equal to the type T.

The type transfer form NARROW[x, T] has type T; it is defined to return x if and only if ISTYPE[x, T] = TRUE; otherwise it raises a runtime type error. The type T can be omitted if it is unambiguously determined by context.

A special form of SELECT statement (similar to Pascal's case statement) has been defined to ease the use of REF ANY variables. The statement

```
WITH v SELECT FROM
    v1: T1 => {<stmtlist1>};
    v2: T2 => {<stmtlist2>};
    ...
    vn: Tn => {<stmtlistn>};
    ENDCASE => {<stmtlistn+1>}; -- assumes only that v has
            type REF ANY
```

is interpreted as if each arm were written as

```
IF ISTYPE[v, Ti] THEN {vi: Ti = NARROW[v]; <stmtlisti>}.
```

(Note: the braces "{}" are equivalent to BEGIN - END

brackets, and the "=" in v_i's declaration makes its value immutable within that arm.) Because the object referenced by v is known to have a specific type in each arm, the arm's statements are permitted to examine or modify its fields.

In addition to generic reference variables, Cedar also has generic procedure types. Procedures can be declared to take values of type ANY as parameters, and/or return values of type ANY as results. Generic procedure values must be narrowed analogously to generic reference types before use.

Generic reference types allow procedures to manipulate objects of prearranged varying types, but do not permit procedures to examine or modify objects of completely unspecified types – an important capability for debuggers and other monitoring tools. To fulfill this need, Cedar provides a runtime type system to manipulate the runtime representations of types; a *type tag* is stored with each collectible object. In the current implementation, these functions are too slow to be a substitute for full polymorphism.

Miscellaneous extensions. Finally, several other flexible data types based on REFs have been introduced into Cedar. Notable among them are variable-length immutable text strings (known as ROPEs), variable-length linked lists, and ATOMs.

Cedar's list-processing facilities are similar to those in Lisp, except that (as with all other Cedar variables) the type of every list variable or expression is a compile-time constant (but may be REF ANY). For example, to declare a list of 32-bit integers and set its value to the first six primes, one could write

```
lprimes: LIST OF INT;
lprimes ← LIST[1, 2, 3, 5, 7, 11].
```

Several operations are defined on lists. For any list l whose elements have type T, the value of l.first is an object of type T (basically Lisp CAR), and the value of l.rest is an object of type LIST OF T (basically Lisp CDR). The function CONS[e, l] constructs a new list lnew with lnew.first = e and lnew.rest = l.

Cedar's ATOMs are uniquely-addressed values much like Lisp atoms. They can be located by their client-assigned names, decorated with property lists, and compared for equality using a simple pointer test.

2.2 The Cedar Machine

All Cedar programming is done using the Cedar language; there are no assembly language routines. The machine hardware, microcode, and low-level runtime support combine to form a virtual machine well-suited to the efficient execution of Cedar programs.

Hardware. The Cedar programming environment runs on the family of Xerox Scientific Workstations, which includes the Dorado and the Dandelion [18]. The Dorado [19] is a high-performance personal workstation with 16-bit words, a cached virtual memory with a large virtual address space (24 bits, word-addressed), and up to 32 megabytes of physical storage (typically two to eight megabytes). The writeable microstore allows customized instruction sets for different languages and environments. Input/output devices include a large (1024 x 808 pixels) high-resolution bitmapped black-and-white display, a keyboard, a mouse pointing device, and an Ethernet interface. A color display can be added.

Cedar workstations operate in the Xerox Research Internet environment that includes database and file servers, shared printers, name authentication servers, and distributed electronic mail services [2, 7, 5, 26].

Microcode. The Cedar microcode implements an extension of the Mesa machine architecture [18], which was designed to execute Algol-like languages efficiently. Two factors combine to produce exceptionally compact representations of programs: a stack machine architecture, which allows zero-address instructions, and variable length byte-coded instructions, whose encodings are based upon an analysis of static instruction frequencies in existing compiled Mesa programs [36]. A compact program representation not only saves storage space, but it also contributes to faster execution, largely due to increased locality and hence fewer cache misses and page faults. The architecture also supports arbitrary control transfer disciplines (such as coroutines): activation records are allocated from a heap rather than a stack. In addition, the architecture allows for concurrent execution of up to one thousand processes. The microcode provides linkages to compiled fault and exception handlers; extensions for Cedar in support of safe storage include reference-counted store instructions and additional exception handlers that intercept invalid storage references.

Runtime Support. Low-level routines and data structure definitions provide a Cedar language interface to the microcoded processor architecture, supporting procedure linkage, process switching, and runtime error handling. Although this component is not written in the safe language, its interfaces are asserted to be safe.

2.3 The Nucleus

The Cedar Nucleus contains the basic operating system facilities needed for memory management, process management, file system management, and communications with the user and the outside world.

Device drivers. Cedar has borrowed from the Xerox Pilot system the notion of abstract device interfaces [30]. Corresponding implementations on each processor for each specific device type extend the virtual machine defined by the microcode to include the peripherals as well. For example, Cedar provides an interface abstractly defining the behavior of disk drives. The Disk component can be programmed in terms of this interface, without detailed knowledge of the peculiarities of each type of drive that the underlying implementations must support.

Disk. The Disk component provides a uniform interface to the attached disk drives. It provides low-level facilities for investigating the state and configuration of each drive, and for performing page-level input/output operations between specified disk addresses and virtual memory locations. Clients of the Disk component must ensure that virtual memory buffers have physical memory allocated to them.

Virtual Memory. The Cedar virtual memory (VM) differs in philosophy from its most recent ancestor, Pilot [30]. Pilot was designed for processors that had relatively small physical

memories and disk capacities. This required a space-efficient but complex implementation based on mapping regions of virtual memory to named disk files. Cedar, intended for larger machines, has been able to abandon this approach in favor of a simpler, more time-efficient scheme. Cedar represents virtual memory as a single backing file, employing a resident page map. VM permits higher-level clients to ensure temporarily that a region of virtual memory has physical memory allocated to it, so that components at levels lower than VM can deal with memory through virtual addresses without incurring page faults.

File input/output must be accomplished by explicit operations, rather than as VM mapping actions. The performance improvements, both for code swapping and for file access, have been significant. Perhaps more importantly, this design permits the virtual memory implementation to occupy a position quite low in the Cedar level structure; only the Cedar machine implementation and the VM implementation itself need to deal with physical memory addresses. Thus the majority of the Cedar system can operate in the virtual memory environment.

Safe Storage. The safe storage extensions to the Cedar language are supported by the runtime type system mentioned in §2.1, a storage allocator (implementing the NEW operator), and a combination of garbage collection techniques. The allocator stores a runtime *type tag* in each new object; these tags index runtime data structures that the garbage collectors use to locate embedded references. The garbage collection algorithms were derived from earlier designs by Deutsch and Bobrow [11]. A full description of the revised algorithms appears in Rovner's recent paper [32].

An *incremental* garbage collector runs at frequent intervals, triggered by specified elapsed-time or memory utilization criteria. It operates as a background process. The incremental collector is able to reclaim most of the storage objects that are no longer referenced, using information obtained from reference counts and examination of current activation records. An optimization called the *conservative scan* reduces the execution time of the incremental garbage collector, but it can cause a few collectible objects to be retained.

The incremental collector cannot detect cyclic data structures, such as those generated by two-way linked lists or certain queue implementations. Programs can explicitly break cycles when they determine that such data structures are no longer needed. In addition, a conventional, preemptive *trace-and-sweep* garbage collection algorithm has been included to reclaim such structures. The trace-and-sweep collector reclaims virtually all unreferenced storage (it also uses the conservative scan), but monopolizes the machine for twenty seconds to several minutes during the process. Servers or other programs that need to remain available for long periods of time without danger of storage leakage can invoke it directly. Users may also invoke the trace-and-sweep collector manually.

A package that creates objects of a given type can also specify *finalization code* to be executed when an object of that type becomes inaccessible outside the package. The finalization code is free to examine the object and perform any final operations such as removing the object from a cache, releasing a virtual memory buffer associated with the object, or breaking the circularity of a data structure to permit additional reclamations by the incremental collector.

File. The local file system underlying Cedar is straightforward. It manages the configuration of one or more physical disk volumes and their subdivision into logical volumes. Within logical volumes, it manages the page-level allocation, deletion, reading, and writing of disk files. The file structuring methods borrow heavily from earlier Xerox systems [22, 30]. In particular, redundant information stored with each file page permits recovery if portions of files or directories are damaged. Only primitive locking facilities are provided, based on many-reader, one-writer write locks.

The File component does not include a directory implementation, leaving that up to higher levels in the hierarchy. Instead, the file-creation procedures return unique identifiers that clients can use to locate the files later. Different clients may choose their own directory organizations for their files, but most choose to use the standard directory implementation.

FS. The Cedar workstation file management and directory package supports the appearance of a uniform file naming space, spanning the user's local disk and the set of shared file servers available through the attached communications network. File names can represent two kinds of files: *local* files, where the only copy of the file resides on the workstation's disk; and *attached* files, where the file name is a symbolic path name to a remote file. Read-only copies of remote files are retrieved and cached as needed on the local disk. FS provides these facilities by maintaining two logical directories describing the contents of the local disk: the *local file name directory* and the *remote file cache directory.*

The local file name directory provides a local, hierarchical name space for files. Arbitrary nested directory structures can be expressed as subdirectories of the single *root* directory. Entries in the local name directory may be either local files or attached files. Thus, a local name space that describes a complete system or set of related tools can be created out of local and remote files.

The remote file cache directory organizes the set of remote files for which local copies exist. Files may be referenced via local file name directory attachments or by using a full symbolic path name. Because files are only copied to the cache when they are needed, often only a small subset of the files indicated by attachments will actually be cached. Disk space is managed automatically by flushing least-recently-used remote file copies from the cache when additional space is needed. Cache entries refer to specific versions of remote files, by name and creation time.

Our current file servers have limitations that prevent reading and writing of remote files from being treated entirely symmetrically. FS will not accept a request to open a remotely-named file for writing. Therefore the file must first be written locally, entering its name in the local directory. A special FS copy routine may then be invoked to create a new remote copy and replace the local directory reference with an attachment to the remote file.

IO. The IO interface defines generic procedures for treating and using streams of characters or words, including useful input scanning and output formatting routines. The Cedar IO package contains over a dozen specific implementations of streams supporting several sources and destinations, among them disk files, the keyboard and display, the Ethernet, and pipes (objects that provide the buffering and synchronization needed to connect an output stream from one process directly to the input stream of another [31]).

It is easy to define specialized streams for specific applications. Programs that read and write streams can be coded without explicit knowledge of the source or destination medium.

Communications. Network communications require substantial software support beyond the low-level device drivers. Cedar includes a complete implementation of the experimental "Pup" internetwork protocols described by Boggs et al in [5]. Lower levels of the Pup package provide a basic datagram (packet-level) service. Higher levels implement asynchronous terminal emulation, a file transfer protocol, a remote procedure call facility, and a range of information utilities, such as time and name lookup services.

Of the higher-level protocols, the most important for new Cedar applications is the communications support for remote procedure calls (RPC). Ordinary calls to procedures through specified interfaces execute on remote machines, returning any results to the caller as usual. The implementation is based on *stub* routines that field the client's calls locally. A stub routine composes procedure parameters into data packets, handles the reliable communication of requests to the remote site, then removes any result values from incoming packets for return to the caller. Corresponding stub routines at the remote site reconstruct the parameters, complete the linkage to the actual procedure implementations, and compose the results into packets. The Cedar RPC package, described by Birrell and Nelson in [3], performs two functions: it automatically constructs both sets of stub routines from the interface definitions, and it provides the underlying algorithms that complete the calls reliably, efficiently, and securely (using optional DES encryption techniques). Cedar RPC builds its protocols directly on the datagram-level of the Pup package.

To date, we have produced three major Cedar systems that use RPC for all their communications: a transaction-based file server, an experimental telephone service, and a "Compute Server". All three are described further in §2.5. Furthermore, implementations of RPC for other languages and programming environments are beginning to extend the range of services that Cedar applications can provide or use.

Terminal. Most Cedar applications are content with the higher-level display-management and user input facilities supplied by *Viewers* and *TIP* (§2.4). However, more radical applications may need to use the display screen or input devices in a conflicting way—to try out a new window package, for example. The *Terminal* interface provides a clean abstraction to the display, keyboard, and mouse. There may be several instances of Terminal, each with its own full-screen bitmap and optional color frame display memory. Operations are available to switch the use of the physical hardware (and thus the entire contents of the screen) among the Terminal instances. The standard Cedar screen is obtained through the use of just one Terminal instance; another is employed to drive a much simpler user interface while the system is being loaded.

Running programs. At the "top" of the Nucleus are two final components. The *Loader* provides the capability to load additional components into a running Cedar environment. (The Nucleus is loaded and initialized using booting methods outside the scope of this paper.) The *Checkpoint/Rollback* component permits the user to save the present Cedar environment (that is, the contents of virtual memory) in a checkpoint file, as well as to restore ("roll back") a machine to the state represented by such a file. It takes several minutes longer to initialize a Cedar system "from scratch" than to roll back to a configuration into which the user has loaded a selected set of development tools and commonly-used applications. The Rollback has become the conventional way to restart Cedar.

2.4 Life Support

The Life Support division provides standard user interaction facilities such as a screen manager, a text editor, command and expression interpreters, and program development and management tools. Many of the Life Support components are quite large, providing functions directly to Cedar users or applications programmers; in this sense they resemble user applications or packages more than operating system components. They are given a division of their own because their functions are vital to providing a complete working environment for users, and because the standard Cedar initialization procedure automatically includes them. Components above the Life Support level are selected and included in the system by individual users.

From this level on, it is relatively easy to experiment with alternative components, either by replacing existing components with variants, or simply by including the alternatives in private configurations and ignoring the system-provided components. A more complete discussion of these techniques appears in Sections 3.1 and 3.4.

Useful Packages. During the implementation of Cedar, many generally useful packages have been produced. Examples include packages for sorting arbitrary values, for maintaining symbol tables, and for managing queues of user-invoked commands. These packages can be thought of as extensions of the basic "Cedar machine." As their numbers increase, they will make programming in Cedar increasingly convenient.

Inscript and TIP Tables. The user communicates with Cedar by typing, by moving the mouse, and by clicking mouse buttons. The *Inscript* package buffers time-stamped versions of these input events. If an application has special high-performance user input requirements, such as the need to react in real time to the trajectory of the mouse-driven cursor, it can use the Inscript package directly and independently to extract the input events from the buffered stream. This works better than direct sampling of the hardware by individual applications, because the Inscript package collects and time-stamps the events using clocked

interrupts; it is therefore less likely that events will be missed or that confusion about the timing of events will occur. Each client of Inscript must determine which of the input events are intended for it and what their semantics are, ignoring those intended for other clients.

Although the Inscript package can be used directly, most applications are satisfied to allow user actions to be interpreted by the *Terminal Input Processor*, or *TIP*. TIP interprets Inscript input events based on easy-to-write specifications called TIP tables. For each event or each event sequence (such as clicking a mouse button twice in succession or depressing a key for a long time), a TIP table entry specifies a procedure to carry out the semantics of the event. Standard rules determine the choice of which TIP table to invoke for each event, as well as which screen region to include as a parameter to the procedure.

A high-priority process called the *notifier* interprets input events according to the current set of TIP tables. A typical TIP procedure creates a new process to carry out the desired action, then returns immediately so that the notifier can react to the next event. In this way, the user can initiate or control many concurrent applications; furthermore, programs can be written in a way that does not preempt the user's ability to choose from moment to moment which application to talk to.

Default TIP tables define standard behavior for the basic Cedar user interfaces. Specialized TIP tables support the special input needs of advanced applications (such as drawing programs). User-specified TIP tables give the user some ability to custom-tailor any existing application.

Abstract Machine. An original goal for Cedar was to combine a compiled, strongly-typed language with the interpretive symbolic power of Interlisp or Smalltalk. The Abstract Machine is a step in this direction. Its facilities are all ultimately based on the symbol tables and program graphs that the compiler, binder, and program loaders produce. Its primary use at present is in support of ordinary Cedar applications that serve as interactive interpreters, debugging tools, performance monitoring, and other tools for presenting program data in a form sensible to users.

The Abstract Machine (AM) implementation is based on the following concepts:

- *Runtime types.* The unique *type tags* that label allocated objects are also used by all the abstract machine interfaces as runtime type values.
- *Program control.* The *AMEvents* interface provides a set of low-level operations for setting breakpoints and for tracing program flow.
- *Type information.* The *AMTypes* interface provides procedural access to the names and structure of data types, including a complete set of operations for analyzing the internal structure of composite types.
- *Value manipulation.* Other *AMTypes* procedures permit examination and modification of runtime values. The association between the referents of REF variables and their type tags can be made safely and automatically by the system; for other values, the associations are based on TRUSTED program assertions. These operations support interpretive programs that can operate on arbitrary data structures; they are always significantly more expensive than the corresponding compiled Cedar statements operating directly on the same objects.
- *Program and process structure information.* The *AMModel* interface provides similar facilities for investigating program structure: the makeup of procedures in terms of their embedded blocks, of program modules in terms of their procedures, and of configurations in terms of their program modules and subconfigurations. A description of the loaded configurations and their associated global information within a running Cedar system is also available through AMModel. Using the *AMProcess* interface, one can enumerate the active processes, suspend or resume the operation of selected processes, and locate the top activation record for a given process.
- *Multiple virtual memory access.* AM uses the *WorldVM* interface for all references to runtime values and to runtime program and process structures. WorldVM supports symbolic access to the *local* address space, to a *worldswap* environment (a restartable memory image saved on disk) or to a *remote* environment (accessed using network communications). The arms-length methods are infrequently used, but they are invaluable when the local methods fail (see §2.4).

Imager. Cedar applications rely on the power and flexibility of high-resolution bitmapped display terminals. In earlier Xerox systems, system support for interactive graphics was limited to low-level bitmap operations, such as the RasterOp (BitBlt) function described by Newman and Sproull in [28]. While it is possible in Cedar to manipulate bitmaps directly, most applications instead use the *Imager* package, which provides support for the presentation of such graphical images as multiple-font text, lines, curves, closed outlines, and sampled images. These images can be scaled, rotated, translated, and clipped to arbitrary rectangular boundaries by providing the package with simple specifications. Programs can render images in a device-independent fashion on color or black and white display devices, or on a variety of laser printers.

Viewers. Most applications are intended to be used in a cooperative fashion, sharing the display real estate with concurrent applications; they do this using *viewers.* Viewers are Cedar's display windows: rectangular regions whose positions and overall sizes are managed by the Viewers package, but whose contents are the business of the applications that create them. The Viewers package redisplays the contents of each viewer, based on client-supplied specifications, whenever its contents, size, or location changes. Viewers can also be "closed"; they then appear at the bottom of the screen as icons (small evocative pictures). It uses TIP tables to provide the connections between the user's input actions and the application-specific functions, serializing these actions when the user types faster than the actions can be performed.

In actuality, there is a hierarchy of viewers. Within the top-level viewers we have been discussing here, one may nest subviewers—perhaps to provide a subwindow whose contents must be scrolled separately, a subwindow

whose contents is provided by some other application, or an area which must otherwise be managed differently from other information displayed in the viewer. Subviewers may be quite small. For example, the menu buttons that appear in each top-level viewer are represented as small subviewers.

Top-level Cedar viewers never overlap, but instead occupy two adjacent columns, each sharing the available height with other viewers assigned to the same column. (If an auxiliary color display is available, a third column of viewers can appear on it.) Viewers can either allow their height to vary to share the available space equitably, or can insist on some fixed or minimum size that they must occupy. The user can override the assigned widths of the columns and the heights of individual viewers. This tiled design was implemented as an experiment whose objectives were to minimize user window scaling and positioning commands and to achieve high-performance screen updating; the underlying graphics facilities would also support the more common overlapping-window model.

The Viewers package also serves as a point of integration for Cedar applications. Viewer instances are assigned to *viewer classes.* A viewer's class determines its display and user interface behavior. Programmers can create viewers as members of standard system classes, or can define their own viewer classes. A viewer can also be associated with a custom TIP table, and with other attachments that customize its operation.

Tioga. Tioga is the tree-structured text editor used to create Cedar programs and formatted documents. Nodes, corresponding approximately to paragraphs, and their text content can be decorated with user-specified style and font information controlling their displayed and/or printed appearance. Tioga is a galley editor; it does not provide automatic support for page makeup.

Tioga displays its files in text viewers, making extensive use of TIP tables to simplify the specification of the user interface. Tioga implements a simple postfix language in which its operations are expressed. This language specifies the meanings of the interactive editing operations, command abbreviations, and other prerecorded sequences of editing actions.

Apart from its value for editing documents, Tioga is an important Cedar resource, since it can be used in any text viewer. This means that applications like command language interpreters and specialized display tools can employ Tioga's well-understood user interface and text-manipulation features. It also means that text and attributes can be freely copied among viewers. For example, one can record the results of a command in a file, or invoke a command by copying it from a "recipe-book" document, using only the mouse-driven text-editing operations of Tioga.

Although Tioga does not understand Cedar syntax, we find that using Tioga as a program editor has several important benefits. First, viewing programs as formatted documents with common stylistic conventions makes them easier to read and share. Furthermore, Tioga's flexible search commands, combined with a small number of connections to the Cedar Abstract Machine, allow it to approach the usefulness of many special-purpose program development tools found in other current programming environments:

- Simple pattern-matching allows Tioga's abbreviation expansion command to construct easily-filled-in templates for language constructs and procedure call parameters. Tioga's node structure and its level hierarchy allow the suppression of detail for a larger contextual view and the manipulation of entire constructs as units. These capabilities provide many of the advantages of modern syntax-directed editors [17, 37, 10].
- Tioga also performs the use-to-definition portion of the Masterscope functions in Interlisp [39]. A selection of the form *interface.item* may be used to request a new viewer displaying the file that defines (implements) the item, scrolled to the item's definition. (If an implementation of *interface* has been loaded, *AMModel* functions are used to locate the implementation's file name; otherwise, Tioga makes a guess based on program naming conventions.) Unfortunately, mapping from an item's definition to its uses is beyond Tioga's capabilities; it would require the capabilities of *Cedar system modelling,* a partially-implemented extension to the DF Package (see §3.1).
- The Blit debugger [8] constructs menus of currently-visible procedure, variable, and field names to ease user input. Tioga's client interface permits the Cedar debugger to show a breakpoint or error location as a highlighted region in a source file viewer; the user can thus see legal procedure and variable names in context. By using the ability to copy text freely among viewers, the user can copy desired names to the debugger area.

Teitelman's Tour through Cedar [38] includes many examples of the various uses of Tioga and Viewers. Those interested in an expanded treatment of the Imager, Viewers, and Tioga are referred to [1].

Compiler, Binder, and Loader. The Cedar *compiler* verifies the correct use of data types both within modules and across module boundaries. In addition to machine code for each module, the compiler produces symbol tables and statement maps for use by the Abstract Machine. The *binder,* also a separate batch application, produces larger configurations of modules from individually-compiled modules and previously-bound configurations. It extends the compiler's strong type checking by ensuring that the names and time stamps of exported interfaces match those specified by the components that import them. Some of the binder's capabilities reappear in the Cedar *loader* program, which loads modules and bound configurations into a running system, resolving the remaining imported references.

Command Tool. Cedar Life Support includes a conventional command interpreter in the form of a text viewer into which the user types commands and the system responds with results. The command syntax, an amalgam derived both from UNIX [6] and from earlier Xerox systems, includes provisions for redirecting command output to another destination (usually a file or a pipe to a process executing a concurrent command), and for accepting command input

from another source (also usually a file or a pipe). [UNIX is a registered trademark of AT&T Bell Laboratories.]

The Command Tool provides a small number of built-in commands, primarily for running programs and for examining and manipulating local and remote file directories through the services of FS (list, delete, copy, and the like). As applications are started, they may register additional commands with the Command Tool, supplying procedures that extend the set of available operations. Commands are usually executed sequentially, or in a tightly-coupled fashion using pipes, but it is also possible to invoke a command such that it runs concurrently, using a separate viewer for its input and output activities. The user may also create more than one Command Tool, then issue commands in each that may run concurrently.

Source-Level Debugging. The Abstract Machine and the Tioga editor form the basis for several standard tools that collectively provide interactive source-level debugging: an interpreter for expressions in the Cedar language, which can be called from a program or driven directly by the user (for instance, in response to a breakpoint); a tool for exhibiting the state of all the running processes; commands for setting and clearing breakpoints in the compiled code; and so on. In addition, specialized diagnostic routines can be built for specific purposes, calling on the facilities of the Abstract Machine. Debugging can be performed in any address space that the WorldVM interface can reach (local, worldswap, or remote Cedar systems). The non-local access methods can be used to debug a memory environment that has been too severely damaged to respond to debugging commands, or to debug Nucleus components.

Version Management. The *DF Package* plays an important role in managing the thousands of files comprising the Cedar system, as well as managing personal files. A DF file describes a package or program by listing the file names of its components, fully qualified with their network locations and create dates. The DF Package operates on DF files to *retrieve* (establish attachments in the local file name directory) the files listed in a DF file from their remote file servers; to store changed versions of the files on remote file servers and update the DF file to refer to the new files; and to verify that a DF file specifies all of the files (with correct versions) that are needed to construct the package. In addition to the list of files comprising a package, a DF file may specify files to be *imported* from other DF files. These files, while not part of the package, are required by it; the DF Package will retrieve them as well. Thus a DF file can specify all of the files needed to compile, bind, and test a package it describes. DF files are also suitable for describing versions of any item consisting of a collection of files, such as the sections and figures of a paper.

The concepts underlying DF files have been extended to serve as a full description of a running program. These *system models* can form the basis for recompilation, runtime module replacement, and answers to queries about a program's structure (similar to Lisp's Masterscope) [33, 21]. A variant of the DF Package has also been adapted for use in the Xerox Development Environment (XDE) [35].

2.5 Applications

By now it should be clear that any distinction between "the system" and "the applications" is a matter of convenience, as is the assignment of components to particular levels. Components that are originally developed as applications are often evaluated, modified, and incorporated into lower levels, usually into the Life Support division. Others are more clearly user programs providing explicit functions supporting specialized needs. Space would not permit the complete enumeration of the Cedar applications produced to date, even if we knew what they were. Here we catalog a set of applications that are representative of the range of activities Cedar can support.

Cedar includes a number of database-related applications. *Alpine* is a transaction-based network file service, written in Cedar, and running on a dedicated Dorado [7]. *Cypress* is an entity-relationship database package that runs in a user's workstation but stores its database on Alpine servers [9]. *Walnut* is an electronic mail system that operates in conjunction with the *Grapevine* message transport mechanism [2], using Cypress and Alpine to manage each user's messages. *Whiteboards* turns a viewer into an electronic "blackboard", where subviewers of various kinds (text, iconic viewers, graphic viewers) can be arranged by the user.

Cedar applications in the area of computer graphics include a program for producing full-page color illustrations (*Griffin*), a system for manipulating three-dimensional synthesized graphical objects (*SolidViews*), programs for processing scanned images, and programs for driving experimental printers [1].

In the communications area, the *Etherphone* system includes a server and individual workstation programs supporting an experimental telephone and voice recording system that uses Ethernet communications to transmit voice [34]. The *Compute Server* is a framework, built upon RPC, that coordinates the assignment and execution of computing tasks to processors with available compute cycles [16].

An assortment of other Cedar-based applications exist. Hardware designers have produced a suite of VLSI design, simulation, and analysis tools in Cedar. The *Spy* (a descendant of the Mesa Spy [25]) is a tool that monitors CPU usage, memory allocation, or page fault performance. *Celtics* is an interactive execution-trace tool.

The sources for the current version of Cedar (Cedar 6.0) occupy more than 17 million bytes of disk storage. There are over 1500 program source files, more than 400,000 lines of source code, approximately 150 DF files, and over 100 separate configurations. This enumeration includes Life Support and the most common applications. Although Cedar continues to grow, the tools for managing its size and complexity seem to be keeping up.

3. Discussion

Cedar, as an operating system and as a programming environment, is the direct descendant of earlier Xerox systems. The progression began with a simple system for the Alto, using the BCPL language and basing its structure

on the notion of an open architecture [22]. When the Mesa language was developed for the Alto, its implementors also produced a faithful rendering of the Alto/BCPL system components, without extending its concepts. The next major development was the Mesa-based Pilot operating system [30] and its associated Tajo programming environment [35, 40], designed for use with a second generation of workstations that included memory mapping and larger physical memories.

Early versions of Cedar were built on a Pilot base, adopting a number of important ideas from more traditionally interactive language systems, notably Interlisp and Smalltalk, in order to achieve some of the Cedar objectives. Later revisions have benefited not only from observations of shortcomings in the earlier attempts, but also from approaches found in the Xerox Development Environment (XDE). XDE is a product, marketed by the Xerox Office Systems Division, that has been developed as an extension to the Pilot/Tajo system.

Cedar has also borrowed from more conventional current operating systems, among them UNIX. But there are also some significant differences, leading to markedly different methods for achieving some desirable properties. In fact, neither Cedar nor any of the systems with which we can most usefully compare it (Interlisp-D, Smalltalk-80, UNIX) achieve all these properties equally well.

In the overview we described the major components of Cedar, choosing an order that progressed from the low-level "virtual machine" capabilities to the more important user applications. Here we concentrate on the overall structure of the system: what it is, why it is that way, and what facilities have been provided in the language and in the environment to support the development of programs. We will address this issue through a discussion of the following topics:

- *Structuring Methods:* The approach that was used to structure the components of Cedar—language attributes, memory management components, and structuring philosophies that were used to achieve the system objectives.
- *Structural Choices:* A discussion of the careful design decisions that led to the ordering of components within Cedar.
- *Comparisons:* Areas of similarity and difference between the architectures of Cedar and selected other programming environments, identifying valuable features that should be considered for inclusion in future Cedar systems.
- *Program Development Methods:* A brief discussion of the effects of Cedar's structure upon program development.

3.1 Structuring Methods

The influence of Alto/BCPL: an open system approach

Much of the design philosophy of Cedar can be traced back to the BCPL-based system designed in the mid-1970's for the Alto personal computer. The designers of the Alto/BCPL system called it an *open* system, contrasting it with conventional multi-user operating systems, which they termed *closed* [22].

A closed system, as defined in the Alto/BCPL report, has hardware memory protection, generally in the form of hardware support for separate address spaces for the operating system routines and for each user application. The operating system provides user programs with special methods for invoking a fixed set of operations. The routines used to provide these operations, unless they are also explicitly exported as system operations, are not available directly to client programs.

The Cedar *open* operating system is essentially a collection of program modules (containing sets of related procedures) sharing the machine's single address space. The important aspects of this open approach are:

- Operating system routines can be called as ordinary Cedar procedures. There is no sharp boundary between client programs and system routines.
- The components of Cedar are carefully arranged into layers. Higher-level layers are built on the capabilities of lower-level ones.
- The components in one layer may only call procedures located in the same or lower layers. This restriction is unfortunately enforced only by convention, although violations often result in system errors. (In the Alto system, it was possible to free the memory occupied by unneeded higher-level layers for other uses; inadvertent upward calls had disastrous results. In Cedar, disabling failures can occur due to the order in which components are loaded or initialized.)
- This structure differs from the *virtual machine* concept, in which each level of a system is implemented entirely in terms of the abstractions provided by the next-lower one. The difference is that, in open systems such as Cedar, the lower-level modules remain directly available to clients at all higher levels. An application can generally choose to use components at any level or to replace them with custom-built components (which can still use the standard lower-level components).

The influence of Mesa: strong typing and interfaces

Alto/BCPL was a useful open system, but it had many shortcomings. BCPL is a typeless language that provides many opportunities for errors that the type systems of Mesa (and thus Cedar) would prevent. Mesa's strong type checking has demonstrably improved the reliability and the ease of development of programs produced for Xerox processors [14].

Mesa's *interfaces* are very useful in describing and delimiting the capabilities supplied by a particular system component. Further, *configurations* provide a concrete way to describe components within the language and to identify the interfaces that each component implements. With configurations, one can also use private copies of standard system components, possibly binding them to different versions of the interfaces they import, without fear of the name conflicts or undetected binding errors that made this kind of thing risky in the Alto/BCPL world. As we will see in §3.4, tools such as Tioga and Viewers can even be

used to develop their successors, by judicious use of the configuration language.

Mesa's interfaces and configurations do not provide a complete descriptive tool for the structure of Cedar. Export lists identify the public and private interfaces of a component, but there is no provision for enforcing the restriction against upward calls. The prototype system modelling language of Lampson and Schmidt [21] is a more powerful specification tool for defining system structure than is the existing configuration language, but it would also need to be extended in order to make the layered structure and its corresponding constraints explicit. This is a topic for additional research.

Mesa processes, protected by monitors [20], may preempt each other at any time, permitting rapid service for high-priority processes and for time-slice scheduling algorithms. These lightweight preemptive processes account in large measure for the success of the multi-tasking capabilities of both XDE and Cedar.

The influence of Smalltalk and Interlisp: safe storage

Cedar's primary contribution to the evolution of this family of open systems is *safe storage.* None of its predecessors are immune to the catastrophic damage or eventually-fatal storage leaks that result from improper pointer management—the kinds of unrecoverable mishaps that traditional "closed" systems were designed to protect against. Where traditional systems confine such damage to the process or job that causes it, Cedar's aim is to prevent the damage entirely, through its combination of compile-time and runtime tests—a technique that is known to work well in Interlisp and Smalltalk implementations. Admittedly, storage leaks, while infrequent, can still occur in the safe subset of Cedar between invocations of the trace-and-sweep garbage collector (due to the inability of the incremental garbage collector to reclaim cyclic structures), and occasionally because of the conservative scan optimization.

Many applications have now been developed using only Cedar's safe subset. These programs required far less diligence and attention to the details of memory management than their earlier counterparts did. Furthermore, algorithms that make heavy use of storage allocation tend to be significantly shorter and easier to read.

In addition to the direct protection benefits of safe storage, we have been pleased by some additional flexibilities that automatic storage management permits. In systems without garbage collection one must deal with the ownership of objects, especially parameters to procedures. For example, a routine that prints text ROPEs might be supplied either with a fixed value, whose storage should not be released since it will be used repeatedly, or with a constructed value, whose lifetime need not extend beyond the completion of the printing routine. The client must either surround the call with allocation-management statements, or must somehow charge the printing routine with the responsibility for managing the disposition of the parameter's storage; either method is clumsy.

Cedar ROPEs are arbitrary-length but immutable text strings whose convenient operations and efficiency have led to their widespread use at all levels of the system. ROPEs could not have been implemented without automatic storage management.

One way for a high-level *client* procedure to thwart the policy forbidding direct upward calls is to supply a procedure value as the parameter to a lower-level *service* procedure. If the *supplied* procedure is to be called during the execution of the service procedure (perhaps defining an action to be performed for every element produced by a generic enumeration procedure), it is known as a *call-back* procedure. Often it is useful to nest the call-back procedure definition within the client procedure, so that it may examine or alter the state of the original client. If the service procedure stores away the supplied procedure for later invocation when specified conditions arise, the supplied procedure is known as a *registered* procedure (these cannot be nested, since the client may return before they are invoked). Since the client supplies the procedure, there is a reasonable guarantee that the higher-level component exists and is initialized. Good examples of registered procedures are the routines that extend the set of operations available to the Command Tool.

It is not immediately obvious, but automatically-managed storage increases the value and safety of call-back and registered procedures, because it provides additional flexibility in the kinds of values that can be exchanged through these procedures. In systems without safe storage, concern over the lifetime of explicitly-managed storage objects has led to restrictions on the use of procedure variables in system calls. In closed systems, difficulties in establishing the proper memory environment generally prohibit the use of either registration or call-back procedures.

In Cedar, the storage management operations are atomic with respect to all but the highest-priority processes (which are not permitted to invoke these operations). Thus, the powerful preemptive-process capabilities of Mesa have been preserved in Cedar without threatening the safety guarantees.

3.2 Structural Choices

Every major revision of Cedar has included careful attention to the layered structure of its components. Each time, new attempts were made to produce a clean, sensible organization satisfying a number of potentially-conflicting objectives:

- The components located lowest in the structure should have the fewest dependencies on other components, so that there need not be violations of the policies prohibiting calls to higher levels.
- For the same reason, there should be no "loops" (mutual dependencies) among components.
- The components located lowest in the structure should provide the most important and widely-used system functions.
- Subject to the above objectives, components should occupy positions as high in the structure as possible. This makes them easier to develop and maintain, and allows them to use more of the system's capabilities.

Ideally, then, the components with the fewest dependencies

must also be the most widely-needed ones in order to avoid conflicts in meeting these goals. In recent versions of Cedar (beginning with Cedar 5.0), these objectives appear to have been met particularly well. The main reason for this is that Cedar 5.0 included a rewrite of the virtual memory, disk, file, and directory packages that eliminated many of the undesirable dependencies. Cedar 5.0 and its successors also make heavier use of *registered* procedures, which permit upward calls to higher-level components when necessary. Non-critical parts of programs can then reside at a higher level.

Additionally, Cedar programmers have been encouraged first to construct packages with well-defined Cedar interfaces describing their functionality, then if appropriate to produce user interface programs (viewer-based tools, Command Tool commands, often both) that call on the packages. While a package may have to be located fairly low in the structure, its user interfaces (which must depend on large numbers of other system resources) can be moved much higher. A good example is the Abstract Machine (located in Life Support) and the myriad debugging applications that depend on it.

Components that do not have access to the basic memory management facilities – in Cedar, VM and Safe Storage – are at a significant disadvantage. They must be very carefully written, and they are often very difficult to understand or change. These components should therefore be located as low in the structure as possible. From Cedar 5.0 onward, the only program above the Cedar Machine that does not use virtual memory is the VM implementation itself. Even device drivers and the Disk package, which are located below VM, can use virtual memory locations, based on methods described in §2.2. VM is so low in the structure that it cannot even find the disk file used to back up memory: when the File Package initializes, it calls VM to inform it of the backing file location. The simple design of VM makes this possible, since file directories or even file concepts are not required to get VM to work.

Safe Storage resides just above VM, having been moved much lower in the structure than was possible in the earlier Pilot-based versions of Cedar. Because of this, nearly all of the system components are written in the safe subset of the Cedar language, resulting in increased reliability and convenience. The location of Safe Storage also enables most programs to use the Cedar data types that depend on collectible storage, including ROPE, ATOM, LIST, and STREAM.

In earlier Cedar systems, parts of the IO package had to be located above the Abstract Machine, because some of its advanced features, such as printing a REF ANY, needed AMTypes functions. This was unfortunate, since the simpler features of IO were widely used. In Cedar 5.0, IO was moved to its present position in the Nucleus, by arranging for the Abstract Machine implementation to supply the procedures needed for the advanced features as registered procedures. Components between IO and the Abstract Machine must merely avoid the advanced features, at least until the Abstract Machine has been initialized.

The placement of other components in the Nucleus and Life Support divisions follow similar reasoning based on the structure objectives stated above. Facilities such as Tioga appear within Life Support at a level that might seem surprisingly low, until one realizes their central importance in the implementation of most Cedar user interfaces.

At the higher levels, the applications are not as tightly interrelated, and the precise layering is not as important. The main problem at these levels is finding an acceptable initialization order for interrelated programs, or in connecting them in such a way that the initialization order does not matter.

There are problems in moving programs to lower positions. One of these is that the debugging and error handling tools depend upon much of the system (including at least the Abstract Machine, FS, File, Safe Storage, Imager, Viewers and Tioga). Local debugging for these packages is delicate, so the worldswap debugger or a remote debugger running on another machine must often be used when working in this region.

3.3 Comparisons

To put Cedar in perspective, we will compare its structure with those of a small number of programming environments that were not in Cedar's direct evolutionary chain, looking at both the similarities and the differences in their designs. Some of the differences are inherent, while others provide insights that could lead to future developments in Cedar. We will look at the two systems from which Cedar has borrowed most heavily: Interlisp-D and Smalltalk-80. We also include a discussion of UNIX, a traditional system whose ideas have influenced Cedar significantly.

There are a number of important programming environment features that we are not considering in this paper: programs as data, fast turnaround for program changes during system development, and the specifics of the user interface. We concentrate instead on structural aspects.

Interlisp-D

Interlisp is a dialect of Lisp, initially an application program running in the Tenex operating system [4]. Since Interlisp provides a single global name space, and since virtually all of the system except the lowest-level primitives and the access to operating system facilities are written in Interlisp, the design is inherently an open one. However, the input/output facilities and wholesale memory management facilities were limited to whatever the Tenex system provided.

More recently, Interlisp has been transported to Xerox personal workstations, including the Dorado and Dandelion. It has been enhanced with a powerful display and window management package (based on earlier prototype work using Tenex Interlisp with Altos as terminals), reappearing as Interlisp-D [17]. Interlisp-D should be classed as an open system, in the sense that all of the components comprising the system are available to client programs.

All Lisp dialects rely centrally on automatic management of their list structures; the clear success of Lisp garbage-collection methods led us to add them to Cedar. When programs use only the basic functional primitives of Lisp, they are inherently safe. To handle concurrent processing, Interlisp-D includes a simple non-preemptive

process scheduler with no semaphore or monitoring facilities. Errors in process synchronization cannot interfere with proper memory management, but one must exercise care to avoid races and deadlocks.

A running Lisp system has no identifiable component structure or explicit layering, but rather contains a vast collection of individual procedures. Of course, the user documentation does present the system in an orderly fashion, clustering groups of related procedures according to their purpose.

Smalltalk-80

Smalltalk systems, from Smalltalk-72 through the present Smalltalk-80, have also evolved towards a greater degree of openness. As with Interlisp, the parts of the systems written in Smalltalk are universally available, since Smalltalk operates in a global name space. And like Interlisp, the amount of the system written in Smalltalk has increased as the implementation became more efficient. Now virtually any aspect of Smalltalk is available to programmers except a very small kernel.

Smalltalk systems have always required automatic memory management, dealing with allocated objects more complex than those of Interlisp. Objects are represented as variable-sized records containing embedded object references. These implementations provided a partial existence-proof for the kind of memory management Cedar needed. The overall safety of Smalltalk-80 is thus similar to that of Cedar and Interlisp-D. The process-management facilities are quite similar to those in Interlisp-D.

The object-oriented approach exemplified by Smalltalk-80 was also a goal of Cedar, a goal so far only partly met. The present Mesa and Cedar languages now include some simple syntactic constructs that allow the programmer to invoke a set of procedures associated with a particular data type using an object-oriented notation. Many Cedar facilities use this syntax, but the construction and management of such objects are the responsibility of each programmer. Moreover, neither the Cedar language nor the system provides any support for the important Smalltalk-80 notion of class inheritance: specific object classes specified as extensions to the specifications of more general ones. Class inheritance is an orthogonal structuring approach to the explicit layering of Cedar components; it deals with the relationships between implementations of related object types rather than the relationships between callers and callees. Classes and class inheritance are important concepts that might benefit strongly-typed languages like Cedar.

Although the Smalltalk-80 implementation does not exhibit an explicit layering of components, it does have effective means for clustering the operations belonging to each component – as collections of operations implemented by a particular class. In fact, the Smalltalk-80 system supports further organization of operations within a class, encouraging the programmer to group these operations into more specifically-defined categories. This is also an idea that could be used to advantage in Cedar.

UNIX

We have chosen UNIX as an example of what we have called a closed operating system, which relies on hardware memory protection to partition the code and data used by the system for its operation from those of the user processes, and similarly to protect user processes from each other. The closed approach has disadvantages which led to the development of open systems like Cedar, but it also has important advantages.

Disadvantages:

- The clear boundary between the application and the system is apparent in the application programs, usually appearing explicitly as a system call of some kind. Subcomponents of the system facilities are often not directly available to applications.
- Applications that run as parts of an integrated system often benefit from the ability to share memory. In particular, the management of the shared screen-view within systems like Cedar are heavily dependent on shared memory. System performance and programming convenience suffer when applications are forced to take a more arms-length approach to information-sharing.
- Changing the operating system to provide new or different functions is not as straightforward as it is in Cedar. (However, we should point out that since UNIX sources are generally available and comprehensible, it is possible to customize a UNIX system.)

Advantages:

- A user process cannot readily interfere with the operation of the system or another process, whatever the inherent safety of the programs running in the process.
- User applications can be terminated and their memory and other resources entirely reclaimed as easily as they can be loaded and started.
- Multiple address spaces make it easier to support more than one programming language or environment on the same machine; detailed memory-management decisions (which are the primary difficulties in getting languages to coexist) are left to the individual processes in their individual address spaces.
- Debuggers can run in protected processes, using system-provided facilities for accessing the target memory and other runtime state, which can be completely frozen during the debugging activity. Cedar's local debugging can break down due to process deadlock or failure in the safety mechanisms; one must then resort to remote debugging or worldswap debugging, both fairly clumsy methods (although perhaps less clumsy than the methods available for debugging the UNIX kernel when troubles arise there).

We believe that the advantages of closed systems are important. Combining the advantages of both approaches to programming environment design, beginning with either base, is an important topic for future research.

3.4 Program Development Methods

One of the goals of Cedar was to provide for fast turnaround from small program changes. In general, this would have required methods for directly replacing an object module with a new version, reestablishing the bindings to its imported components and to its clients. This capability does not yet exist within Cedar.

Instead, Cedar programmers have employed two main techniques for developing new versions of program modules or configurations. The simplest method is *replacement*: producing a new instance of the system using the new module version instead of the old one. It is usually more convenient, when possible, to *augment* the system, adding new instances of a module so that multiple versions exist concurrently. In the latter instance, it might be necessary to hide the new module within a configuration that does not export all of the module's interfaces, to avoid name conflicts or improper binding.

A Cedar system is constructed by making a boot file from the Cedar language components of the Nucleus and Cedar Machine levels. When this file is booted, it reads a *boot configuration file* that contains a list of programs that will comprise the Life Support division. There is a default configuration file, but the user may supply a substitute. Similarly, each user supplies a file (the user profile) that specifies which Applications level programs to load once the Life Support components have been initialized. During normal system operation, the user can load additional programs, usually by issuing requests to the Command Tool.

When replacement is necessary, the level of the module determines how hard it is to replace. If it is part of the Cedar Machine or in the Nucleus, a new boot file must be constructed and the workstation rebooted (five minutes to build and boot on a Dorado). If the module is in Life Support, the boot configuration file must be altered to include the new version (two minutes to boot). If the software is at the Applications level, one need only perform a Rollback operation to produce a version of the system that does not contain the module before running the new version (one minute).

It is possible to augment the system with a new module version whenever neither version will interfere with the other's proper operation. If the new module exports a new version of any existing interfaces, it must be hidden in (bound into) configuration that does not export them. One loads the new version of the module or the hiding configuration, reloads any higher-level modules that depend on it, then tests the addition. Since old versions are not being removed, an occasional Rollback operation must be performed to produce a "clean" version of the system. Most programs are developed this way.

An instructive example of augmentation involved a recent revision of the Imager; the new one supports improved device independent graphics, but it is incompatible with the old one. During testing, its developers wanted to use tools based on the released versions of Viewers and Tioga to debug the new version. They constructed a configuration that contained new versions of the Imager, Viewers, Tioga, and TIP, but which exported none of their interfaces to the system at large. They had to include a few additional programs (including Inscript) that were not sufficiently reentrant to be shared with the existing tools. When the test configuration was started from the CommandTool, it obtained a new "virtual terminal" from the Terminal Package. They could switch the real terminal between the virtual terminals by typing special function keys, providing access to both the old world and the new one. They were able to use the standard system viewers and debuggers to examine and debug the new packages from the normal display, switching the virtual terminal to the new world in order to interact with it and view the effects.

These methods do not eliminate the need for a more general module replacement facility, but they have proven remarkably effective.

4. Summary

In this paper, we have described the major parts of the Cedar programming environment. We have shown how strong typing and explicitly-specified interfaces help support the layered architecture of Cedar. We have stressed the system layering, which is designed to reduce compilation dependencies and to make important system components available to the largest possible number of clients.

Throughout, we have emphasized the contribution of safe storage (incremental garbage collection, runtime type discrimination, generic references, and runtime symbolic access) to the cleanliness of Cedar's structure, as well as to its convenience and reliability.

Experience with Cedar's predecessors, with earlier versions of Cedar, and with other open systems have contributed to its architecture, as have important features derived other environments, including Smalltalk-80, Interlisp-D, and UNIX. In Cedar, we have attempted to integrate these traditions into one programming environment.

Acknowledgments

Cedar has been a massive undertaking, so far, consuming well over fifty person-years of design and implementation effort. There is not space to name them all, but we felt it important to acknowledge the major contributors, and even that is a long list! The following people were primarily responsible for the conceptual development, implementation, and project management of Cedar: Russ Atkinson, Andrew Birrell, Mark Brown, Bob Hagmann, Butler Lampson, Roy Levin, Scott McGregor, Jim Morris, Bill Paxton, Michael Plass, Paul Rovner, Ed Satterthwaite, Eric Schmidt, Mike Schroeder, Larry Stewart, Ed Taft, Bob Taylor, Warren Teitelman, John Warnock, and Doug Wyatt. Upwards of fifty people have made significant contributions, either officially or as creative users. To those contributors who were not mentioned explicitly, please accept our apologies and appreciation.

We would like to thank Rick Beach, Luis Felipe Cabrera, Pavel Curtis, Carl Hauser, Michael Plass, and Larry Stewart for their careful reading and helpful comments

during the preparation of this report, and Subhana Menis for her usual professional editing and formatting assistance.

References

1. R. Beach. "Experience with the Cedar Programming Environment for Computer Graphics Research," *Graphics Interface 84.*
2. A. Birrell, R. Levin, R. Needham, and M. Schroeder. "Grapevine: An Exercise in Distributed Computing," *CACM* **25**, *4*, Apr 82.
3. A. Birrell and B. Nelson. "Implementing Remote Procedure Call," *ACM TOCS* **2**, *1*, Feb 84.
4. D. Bobrow, J. Burchfiel, D. Murphy, and R. Tomlinson. "TENEX: a Paged Time Sharing System for the PDP-10," *CACM* **15**, *3*, Mar 72, 135-143.
5. D. Boggs, J. Shoch, E. Taft, and R. Metcalfe. "Pup: An Internetwork Architecture," *IEEE Transactions on Communications* **28**, *4*, April 80, 612-624.
6. S. Bourne. "The UNIX Shell," *Bell System Technical Journal* **57**, *6*, Part 2, July-Aug 78, 1971-90.
7. M. Brown, K. Kolling, and E. Taft. *The Alpine File System*, Xerox PARC Report CSL-84-4, 1984.
8. T. Cargill. "Debugging C Programs with the Blit," *AT&T Bell Laboratories Technical Journal* **63**, *8*, Part 2, 1984, 1633-48.
9. R.G.G. Cattell. *Design and implementation of a relationship-entity-datum data model*, Xerox PARC Report CSL-83-4, May 83.
10. N. Delisle, D. Menicosy, and M. Schwartz. "Viewing a Programming Environment as a Single Tool," *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Apr 84, 49-56.
11. P. Deutsch and D. Bobrow. "An Efficient, Incremental, Automatic Garbage Collector," *CACM* **19**, *7*, July 76.
12. P. Deutsch and E. Taft. *Requirements for an Experimental Programming Environment*, Xerox PARC Report CSL-80-10, 1980.
13. J. Donahue. "Integration Mechanisms in Cedar," *Proc. of ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments*, June 85.
14. C. Geschke, J. Morris, and E. Satterthwaite. "Early Experience with Mesa," *CACM* **20**, *8*, Aug 77.
15. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*, McGraw-Hill, 1983.
16. R. Hagmann. *Process Server: Sharing Processing Power in a Workstation Environment*, in preparation.
17. *Interlisp Reference Manual*, Xerox Corporation, Oct 83.
18. R. Johnsson and J. Wick. "An Overview of the Mesa Processor Architecture," *Proc. of Symposium on Architectural Support for Programming Languages and Operation Systems, Apr 82 (SIGPLAN Notices* **17**, *4*, Mar 82).
19. B. Lampson and K. Pier; B. Lampson, G. McDaniel, and S. Ornstein; D. Clark, B. Lampson, and K. Pier. *The Dorado: A High Performance Personal Computer, Three Papers*, Xerox PARC Report CSL-81-1, Jan 81.
20. B. Lampson and D. Redell. "Experience with Processes and Monitors in Mesa," *CACM* **23**, *2*, Feb 80, 105-117.
21. B. Lampson and E. Schmidt. "Organizing Software in a Distributed Environment," *Proc. of SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, San Francisco, June 83.
22. B. Lampson and R. Sproull. "An Open System for a Single-User Machine," *Proc. of the Seventh Symposium on Operation Systems Principles*, Dec 79, 98-105.
23. H. Lauer and E. Satterthwaite. "The Impact of Mesa on System Design," *Proc. of the 4th Int'l Conference on Software Engineering*, Munich, Sept 79.
24. E. McCreight. "The Dragon Computer System: An Early Overview," in *Proc. of the NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, Urbino, Italy, July 84.
25. G. McDaniel. "The Mesa Spy: An Interactive Tool for Performance Debugging," *Proc. of 1982 ACM SIGMETRICS Conference on Measurement and Modleing of Computer Systems*, Aug 82.
26. R. Metcalfe and D. Boggs; R. Crane and E. Taft; J. Shoch and J. Hupp. *The Ethernet Local Network: Three Reports*, Xerox PARC Report CSL-80-2, Feb 80.
27. J. Mitchell. *Mesa Language Manual*, Xerox PARC Report CSL-79-3, 1979.
28. W. Newman and R. Sproull. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, 1979.
29. S. Owicki. "Making the World Safe for Garbage Collection," *POPL 8*, Jan 81.
30. D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. "Pilot: An Operating System for a Personal Computer," *CACM* **23**, *2*, Feb 80.
31. D. Ritchie and K. Thompson. "The UNIX Time-Sharing System," *Bell System Technical Journal* **57**, *6*, Part 2, July-Aug 78, 1905-30.
32. P. Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Xerox PARC Report CSL-84-7.
33. E. Schmidt. *Controlling Large Software Development in a Distributed Environment*, PhD Thesis, U.C. Berkeley 1982; also available as Xerox PARC Report CSL-82-7, 1982.
34. L. Stewart, D. Swinehart, and S. Ornstein. "Adding Voice to an Office Computer Network," *Proc. of GlobeCom 83, IEEE Communications Society Conference*, Nov 83.
35. R. Sweet. "The Mesa Programming Environment," *Proc. of ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments*, June 85.
36. R. Sweet, J. Sandman, Jr. "Empirical Analysis of the Mesa Instruction Set," *Proc. of Symposium on Architectural Support for Programming Languages and Operation Systems, Apr 82 (SIGPLAN Notices* **17**, *4*, Mar 82).
37. T. Teitelbaum and T. Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *CACM* **24**, *9*, Sept 81, 563-573.
38. W. Teitelman. "A Tour Through Cedar," *IEEE Software*, April 84.
39. W. Teitelman and L. Masinter. "The Interlisp Programming Environment," Computer **14**, *4*, 1981, 25-33.
40. D. Wallace. *Tajo Functional Specification, Version 6.0*, Xerox internal document (Oct 80).