Check for updates

# A Structural View of the Cedar Programming Environment

DANIEL C. SWINEHART, POLLE T. ZELLWEGER, RICHARD J. BEACH, and ROBERT B. HAGMANN
Xerox Palo Alto Research Center

This paper presents an overview of the Cedar programming environment, focusing on its overall structure—that is, 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. Its primary purpose is to increase the productivity of programmers whose activities include experimental programming and the development of prototype software systems for a high-performance personal computer. The paper emphasizes the extent to which the Cedar language, with run-time support, has influenced the organization, flexibility, usefulness, and stability of the Cedar environment. It highlights the novel system features of Cedar, including *automatic storage management* of dynamically allocated typed values, *a run-time type system* that provides run-time access to Cedar data type definitions and allows interpretive manipulation of typed values, and *a powerful device-independent imaging model* that supports the user interface facilities. Using these discussions to set the context, the paper addresses the language and system features and the methodologies used to facilitate the integration of Cedar applications. A comparison of Cedar with other programming environments further identifies areas where Cedar excels and areas where work remains to be done.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Programming Languages**]: Language Constructs; D.4 [**Operating Systems**]: General; D.4.7 [**Operating Systems**]: Organization and Design; H.0[**Information Systems**]: General

General Terms: Design, Languages

Additional Key Words and Phrases: Experimental programming, integrated programming environment, open operating system, strongly typed programming language

## 1. INTRODUCTION

Cedar is an environment for developing and testing experimental computer software systems. The primary focus for Cedar has been to develop a wide range of experimental office information and personal information management applications. Cedar also supports the development and evolution of the Cedar environment itself, as well as research into new technologies such as VLSI design

This paper is a revision and extension of an earlier preliminary version, "The Structure of Cedar," which appeared in *SIGPLAN Not. 20*, 7 (July 1985).
Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.
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.
© 1986 ACM 0164-0925/86/1000-0419 $00.75

tools for high-performance computer systems. People working with Cedar on such projects do so in small groups or as individuals. Cedar is used by several organizations within the Xerox Corporation, but primarily in the Palo Alto Research Center (PARC) where Cedar was created.

This paper presents an overview of the Cedar environment. It examines how the Cedar environment is structured to meet the needs of creating experimental software systems, and describes many of the components that make up the environment. It also reviews the features of the Cedar language, which has had a strong influence on Cedar's design. The paper identifies several important concepts contained in Cedar that contribute to its power and success. We briefly evaluate the success of Cedar in meeting the goals set out in its requirements document and also compare Cedar to other programming environments that attempt to meet similar goals.

This paper does not concentrate on user interface issues or on how to use Cedar, but does occasionally refer to such issues when they relate to the power or structure of Cedar. Readers interested in a Cedar user's view of software development in an earlier version of Cedar should refer to Teitelman's paper, "A Tour Through Cedar" [51].

Over time the name Cedar has come to refer to both the major language in the environment and to the environment itself. Originally, the language was named "Cedar Mesa," to indicate its heritage as an evolution and proper superset of the Mesa language. However, common usage shortened the name of the language to simply Cedar, and left the reader or listener to determine from context whether Cedar referred to the language or the environment. In this paper we consistently use the phrase "Cedar language" to refer to the language; "Cedar" by itself refers to the programming environment.

An earlier and shorter version of this paper was presented at the ACM SIGPLAN/SIGSOFT Symposium on Language Issues in Programming Environments [49].

### 1.1 The Origins and Evolution of Cedar

Cedar was created and continues to evolve in the Computer Science Laboratory at Xerox PARC. Cedar is a research environment supporting the development and use of experimental programs, emphasizing office information and personal information management applications. These programs often feature formatted text, graphics, digitized voice, databases, and distributed computing. Although it was clear at the outset that some unsolved problems would be addressed, the intent was to combine well-understood methods and technologies to create an environment for software research that would exploit a new generation of high-performance personal computers, including the Xerox 1132 (Dorado) [26] and the Xerox 1108 (Dandelion) [25].

Cedar, as an operating system and as a programming environment, is the direct descendant of several earlier Xerox systems: Alto, Smalltalk, Interlisp, Pilot, and the Xerox Development Environment.

The evolution of Cedar began with a simple environment for the Alto personal computer [54]. The most important new feature of the Alto was its bitmapped display, which allowed a new style of user interaction. Software for the Alto was created using the typeless BCPL language. Its structure was based on the notion

of an open[1] operating system [29]. The tools built for the Alto environment were limited primarily by memory, but also lacked integrated functionality and were executed serially, thus monopolizing the Alto during their use. The Alto system exhibited a varied set of user interface paradigms, since this was a time of experimentation and of limited ability to share existing code.

When the strongly typed Mesa language [35] was developed for the Alto, the Mesa implementors produced a faithful rendering of the Alto/BCPL system components, without extending their concepts. The next major system development was the Mesa-based Pilot operating system [40] and its associated Tajo programming environment [47, 55]. Pilot and Tajo were designed for use with a second generation of workstations, such as the Dandelion, that included memory mapping and larger physical memories than were available on the Alto. The Pilot/Tajo programming environment was used within Xerox to create the Star and Viewpoint office systems, and later became the Xerox Development Environment (XDE) product, marketed by the Xerox Information Systems Division.

The Cedar system started with an assessment in 1978 of the goals and requirements for an experimental programming environment. The requirements document [15] outlined over 50 goals aimed at defining an environment for creating moderate-sized programs to be used by moderate numbers of people. The development of the Dorado within PARC at that time promised significantly greater computing resources, including several times more processing speed, larger real and virtual memories, higher memory bandwidth for color and larger black-and-white displays, and larger local disk storage. An assessment of how earlier versions of Cedar met the goals of the requirements document is presented by Teitelman [52].

Appendix B charts the evolution of Cedar over the next five years. The earliest versions of Cedar were built on the Pilot operating system. These early versions adopted ideas from other interactive programming environments, notably Interlisp and Smalltalk. Later versions incorporated changes due to innovations in XDE and those due to observed shortcomings in earlier versions of Cedar. These issues are discussed in Section 3, *Structural Overview of Cedar.*

Cedar has also borrowed from more conventional current operating systems, among them the UNIX® operating system. However, Cedar and systems with which we can most usefully compare it (Interlisp-D, Smalltalk-80, and the UNIX systems) attempt to achieve similar goals through markedly different methods. A comparison of Cedar with these systems appears in Section 10, *Comparisons.*

### 1.2 Benefits to Cedar Users

The Cedar programming environment offers several important and powerful paradigms for software development. Four of the major benefits are improved programmer productivity, software integration, higher quality software, and flexible program development methodologies.

---

[1] Our use of the term *open operating system* differs from the more current notion of an open system. An *open operating system* is a structuring methodology for an operating system (see Section 3.1, *Open Operating Systems*), while an *open system* implies that the interfaces to a system are in the public domain [59].

® UNIX is a registered trademark of AT&T Bell Laboratories.

1.2.1 *Improved Programmer Productivity.* Cedar was designed to improve the productivity of experienced programmers in the development of experimental programs. This resulted in a rich set of program development tools (program editors, compilers, symbolic debuggers, and version managers), automatic storage management to reduce the drudgery of storage allocation, and an extensible system architecture that provides leverage in building new applications through exploitation of existing implementations. Cedar also supports concurrent operation of several applications. A programmer can therefore attend to the most important activity by easily switching his attention among several tools and operations. As an aid to development and testing, Cedar also allows experimental versions of programs to operate concurrently with earlier versions.

1.2.2 *Software Integration.* Considerable power can be achieved by building on the best work of others to incorporate those ideas or functions in new systems. Integration is more than simple techniques for interconnecting programs [16]. Rather, integration applies to designing extensible and customizable software packages, and to building packages that can be used by other programs. Furthermore, integration extends to consistent user interface paradigms and shared components for user interface construction that transfer the user's training from one application to another. Several integrating mechanisms available in Cedar are described in Section 8.

1.2.3 *Higher Quality Software.* We expect to use our experimental programs in real situations to solve real problems. The environment must therefore be robust and must gracefully handle large problems. Larger and faster processors, support for large numbers of concurrent applications, and the availability of shared resources within a distributed computing environment all contribute to improved software quality, as do consistent and carefully designed interfaces to the applications and their implementations.

1.2.4 *Flexible Program Development Methodologies.* The Cedar environment itself serves as a model for the development of software systems. Small applications can often be built quickly by extending and combining existing interfaces. Large applications can be built using the same tools and techniques as those used for building the Cedar environment. For example, a large suite of VLSI design tools is being built with Cedar software development tools.

### 1.3 Novel Aspects of Cedar

As Cedar has evolved, several aspects of this programming environment have proved to be particularly interesting:

—Safe Storage, which provides automatic storage management in a strongly typed language;
—deferred type binding for run-time type discrimination;
—an open operating systems approach to the Cedar system, including its components, tools, and applications;
—the Cedar Abstract Machine, which provides program access to program structures, types, and data;
—local, remote, and multimachine symbolic debugging in context;
—Tioga, a programmer's text editor, which is extensible and integrated into the environment;

—the Imager, a device-independent graphics package for high-quality two-dimensional images;
—several methodologies for implementing and managing large system development.

These aspects of Cedar will be introduced and discussed when relevant in the remainder of the paper.

### 1.4 Outline of the Paper

Cedar is a large complex software system. This paper describing Cedar is also large and complex. Some of the technical concepts we present profoundly influence the design of many Cedar components. Consequently, the reader may find it difficult to appreciate the implications of some concepts until a thorough reading has exposed their influences and consequences. The authors certainly found it difficult to present these concepts in a linear fashion. Unfortunately, we can only publish a linear document, and we chose the following outline.

Understanding the Cedar environment begins with an understanding of the Cedar language presented in Section 2, *The Cedar Language.*

The layered architecture of the Cedar components is introduced in Section 3, *Structural Overview of Cedar*, which discusses the major design philosophies used in Cedar. The four major layers are presented in four separate sections: 4. *Cedar Machine,* 5. *Nucleus*, 6. *Life Support*, and 7. *Applications.* These largely descriptive sections concentrate on the components of each layer, together with some structural aspects. Material that has not been published previously—for example, the Abstract Machine component of the Nucleus—receives greater attention.

The focus changes from a structural overview to a methodological discussion in Section 8, *Methodologies*, which describes the methods developed to use the components of Cedar to greatest advantage. Two case studies in Section 9, *Case Studies*, illustrate how Cedar was used to develop the Cedar Imager and to build an integrated voice-annotated electronic mail system. The next section, 10, *Comparisons*, examines other programming environments that have attempted to achieve similar goals and requirements. Concluding arguments about the strengths and weaknesses of the current Cedar system are presented in Section 11, *Conclusions.* Appendix A contains the glossary of Cedar terminology, which defines Cedar terms and indexes their use in the paper. Appendix B chronicles the release history of Cedar.

The authors suggest that readers who are unfamiliar with Cedar should read the language section, 2, in some detail, briefly skim the structural overview, Sections 3–7, and then read the methodology, Section 8, case studies, Section 9, and comparisons, Section 10. Using the glossary on first reading will help define unfamiliar terms and locate sections that discuss terms in more detail. A second reading of the language and structural overview sections will permit greater understanding of the conclusions drawn in Section 11.

## 2. THE CEDAR LANGUAGE

The Cedar language, an extension of Mesa [19, 27, 31, 35], is a strongly typed systems implementation language in the ALGOL family. Mesa includes facilities for modularization and separate compilation (with full type-checking across

module boundaries), lightweight processes and monitors, exception handling, and procedure variables. Cedar language extensions retain full type-checking while providing automatic storage management and facilities for delaying the binding of type information until run-time. In addition, the Cedar language provides immutable[2] strings, as well as Lisp-like lists and atoms.

This section presents selected features of the Cedar language. It reviews those features inherited from Mesa that strongly influence the structure of the Cedar system and describes the features unique to the Cedar language. Throughout this paper we use the term "Mesa/Cedar" to refer to the common portion of the two languages. The features described here cover the major differences between Cedar and Pascal. Other Cedar language details are omitted for brevity.

### 2.1 Strong Typing, Interfaces, and Modules

The Mesa/Cedar language is strongly typed. That is, the type of every value can be determined via static analysis. The compiler performs this analysis to ensure the type-correctness of all programs. Strong typing allows the compiler to catch many common programming errors and to produce efficient code.

A Mesa/Cedar program consists of a set of separately compiled modules. They are of two kinds: *interface* and *implementation* modules. An interface module acts as a specification for a related set of functions or a data abstraction. It describes public data types, procedures, and variables. It can also include definitions of opaque types, whose structure and behavior are hidden from the external world. An implementation module contains executable statements, provides storage for variables, and supplies concrete representations for opaque types.

For example, suppose that interface module A defines procedure P (that is, A specifies the names and types of P's parameters and results). An implementation module that supplies the code for P *exports* an implementation of P to A. Other modules that access A.P must *import* interface A; these modules are *clients* of the interface A.

Associated with each running implementation module is a *global frame* that contains storage for its global variables. These global variables act as **own** variables for the procedures defined in a module. It is possible to allocate multiple instances of a global frame for a single implementation module.

The clear separation of interfaces and implementations serves several valuable purposes. First, it provides selective information hiding between implementors and clients. This permits independent program development and allows multiple implementations of the same interface. For example, an implementation can be modified to fix bugs or to improve performance without requiring client recompilation, provided that the modified implementation continues to conform to the original interface. Second, it provides the mechanism for intermodule type-checking. Interface modules are compiled into symbol tables that are consulted when modules are compiled, when they are bound together to form a program, and when they are loaded into the system. The enforcement of strong

[2] Throughout this paper, when we refer to immutable values we mean values that, once created, may not be changed. Operations on such values include deleting them entirely (usually through garbage collection), examining them, or producing new values by copying all or parts of them. References to immutable strings may be passed freely among programs without danger that the values observed by the original owners will change.

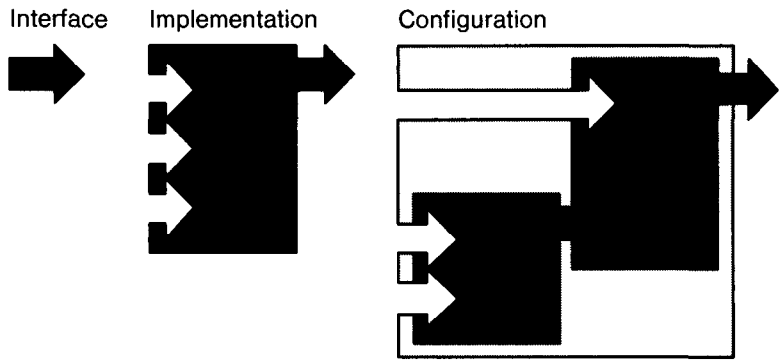


Fig. 1. The dependencies among interfaces, implementations, and configurations are crucial to modular structure in the Mesa family of languages. An interface is depicted as an arrow representing an abstraction. An implementation is depicted as a block of code that may export one or more interfaces and that may import several interfaces to supply procedures or variables upon which the implementation depends. A configuration may also export and import interfaces depending on the combination of implementations within the configuration. Many more arrangements of imported and exported interfaces are possible than depicted in these simple examples. For an example, see the Walnut Voice configurations in Figure 7.

typing across module boundaries permits Mesa/Cedar programmers to make extensive changes to large systems with confidence. For example, if the number or types of a procedure's parameters are modified and a module imports an old version of the procedure, then the compiler, the binder, or the loader will report a type error.

A separate *configuration description* specifies how to resolve the imports and exports of a collection of implementation modules. Configurations allow flexibility in creating a program from alternative implementations of the same interface. A configuration can be hierarchical, including within it subconfigurations. Configurations, like modules, may also import and export interfaces, as shown in Figure 1. This provides name scoping for interfaces. The process of combining modules based on a configuration description is called *binding*. Interfaces imported by a top-level configuration will be resolved during loading. The binding mechanisms supported by Cedar are static. Dynamic binding is achieved by using procedure variables and a variety of conventions that will be described throughout the paper.

## 2.2 Lightweight Processes and Monitors

Mesa/Cedar provides *lightweight processes* and language support for managing them. Lightweight processes are multiple concurrent threads of control that share the same address space [12, 27, 30]. A lightweight process is entirely represented by its current execution state. This state is a chain of procedure activation records, each containing the local variables and program counter for a procedure invocation that has not yet completed. By contrast, a "heavyweight" process also includes management of major resources, such as virtual memory, open files, and devices [41].

The efficiency of Mesa/Cedar's processes makes it natural to structure programs to reflect their inherent concurrency. A process switch is fast (it takes about twice as long as a procedure call–return pair, or approximately 8 microseconds on a Dorado).

*Monitors* [8, 24] are a unified mechanism for providing synchronization among multiple processes and for protecting shared data. Only one process can execute within any procedure of the monitor at a time. If a process discovers that it needs additional resources while inside a monitor, it can wait on a *condition variable.* This suspends the process and places it on a waiting list associated with that condition variable, after which other processes can enter the monitor. The suspended process runs again when the condition is *notified* or when a timeout occurs.

In situations where a single monitor lock is too restrictive, monitored objects allow additional concurrency. By associating a separate monitor lock with each object, this alternative permits one process to execute inside the monitor for each object instance.

### 2.3 Exception Handling

The exception-handling mechanism provided by the Mesa/Cedar language allows the implementation of an abstraction to check preconditions and ensure internal consistency efficiently. In addition, its distinguished syntax alerts a reader of the program source that an exceptional condition can arise.

When an exception is raised during the execution of a statement, normal execution is suspended and a handler (a special kind of procedure) is invoked. The correct handler for a given condition is determined by a combination of dynamic and static scoping. The handler is selected by searching backward through the call stack of the process for the innermost caller that has supplied a handler for the condition. A handler can specify termination or resumption of the statement that raised the exception. In the termination case, execution continues in the procedure containing the handler, and the later procedures are aborted. To give procedures that are about to be aborted in this way an opportunity to restore a consistent state, the special exception UNWIND is generated in each procedure. For example, a monitor entry procedure typically provides an unwind handler to reestablish its monitor invariant. All exceptions that are not caught by explicit handlers are caught by the Cedar debugger, leaving the suspended execution state intact for examination.

### 2.4 Procedure Variables and Support for Object-Style Programming

Mesa/Cedar procedures can be treated as values. They can be passed as arguments and saved in variables for later invocation. Procedure variables have a variety of important uses, several of which are discussed in more detail in later sections. They are also essential to the implementation of object-style programming, which we define as the representation of program behavior through dynamically created objects that specify both the data corresponding to some abstraction and the operations that can be performed on or by the objects.

In the Mesa/Cedar language, an object's data and the values of procedures that implement its operations can be specified together in the same record. As an

optimization of this format, objects that have many object instances and many operations can be represented by creating an explicit record of procedures for each kind of object. A pointer to this procedure record is then included with the object instance data. The indirection inherent in this structure permits a form of object "classes": each class shares the same procedure record. It is possible to create objects with modified behavior by copying the procedure record and replacing a subset of the procedures. However, the Mesa/Cedar language provides no formal class inheritance mechanism.

For a longer discussion of this topic and an instructive example, see Serlet's discussion of object-oriented programming in Cedar [45].

### 2.5 Automatic Storage Management and the Safe Language

Extensions to the Cedar language provide the basis for automatic storage management [42]. These extensions eliminate the following two problems with Mesa's explicitly allocated and deallocated pointers to dynamic storage:

—The programmer must deallocate a dynamic object at 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 its storage being deallocated for reuse.

—Invalid pointers can result from failure to initialize a pointer, from incorrect pointer arithmetic, or from explicit violations of the type system through improper use of type coercions such as the LOOPHOLE construct. Using an invalid pointer to modify memory can destroy program or system data structures in ways that are difficult to track down.

The first problem is solved by automatic storage deallocation, in which garbage collection algorithms built into the system take on the responsibility of deallocating dynamic objects when they are no longer being used. This makes the construction of experimental programs significantly less tedious for programmers. Furthermore, the structure of the resulting programs is often simpler. Less handshaking is required between a client and an abstraction regarding who will deallocate a dynamic object, and exception handlers need not be certain to deallocate the storage owned by a routine that is being aborted.

The *safe subset* of the Cedar language addresses the second problem. The safe subset includes a carefully selected subset of the original Mesa language as well as the extensions described here. It has been formally demonstrated that even erroneous programs written in the safe subset maintain a set of *storage invariants* that ensure the integrity of the storage allocation structures, other system data, and all code [38]. 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 described below provides ample warning that the programmer is responsible for maintaining the system's storage invariants.

*Reference* types, a new class of pointer types analogous to Pascal's or Mesa's POINTER types, allow safe access to collectible storage. A reference variable, called a REF, holds the address of a collectible object of a specified data type. To help ensure that a reference's value is always valid, the system automatically initializes it to NIL. The operator NEW allocates a new collectible object of a specified type,

with optional initialization of its contents, and returns a reference to the new object. References may be freely copied (by assignment or by procedure parameter binding) and discarded (by assigning NIL or by exiting a scope). The system makes a region of collectible storage available for further allocation only when no references to it remain.

For example, the declarations

```
Node: TYPE = RECORD[left, right: 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 contents field to the letter "A", and stores a reference to the new object in root.

Procedures are either SAFE or UNSAFE, depending on whether they guarantee to maintain the storage invariants. The programmer can let the compiler check that a safe procedure uses only safe constructs (a CHECKED block), or may instead assert its safety (a TRUSTED block). Code in a CHECKED block cannot be the direct cause of a memory smash. In addition to ensuring that only constructs in the safe subset are used, the compiler generates tests for illegal memory references, such as out-of-range assignments to numeric variables and array index bounds' violations.

The use of unsafe language constructs is permitted in TRUSTED and UNCHECKED blocks. These constructs include the type-escape mechanism LOOPHOLE, the original Mesa POINTERs, and address arithmetic. Situations that require their use arise most often in low-level system code, such as the low-level implementation of data structures and safe storage itself, low-level I/O, and unpacking network communication packets. By declaring a block to be TRUSTED, the programmer asserts that all uses of unsafe features within the block maintain the storage invariants. UNCHECKED blocks carry no programmer warrantees.

Our use of the term *automatic storage management* throughout this paper denotes both the notational convenience and protection offered by the safe subset of the Cedar language, and the set of storage management capabilities that are enabled by the allocation and garbage collection methods of the Cedar Safe Storage facilities (See Section 5.4).

### 2.6 Delayed Type Binding

Delaying type binding until run-time can provide important program flexibility. 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 run-time, and the lengths of sequences and descriptor-based strings and arrays may be specified then. Additional type flexibility in Mesa can only be achieved through use of the unsafe type-escape mechanism LOOPHOLE. Cedar language extensions for delayed type binding include a generic reference type (REF ANY) and a run-time type system.

A variable of type REF ANY can take on a value of type REF T for any type T. However, the actual type of the referenced object must be verified at run-time before the object can be examined or modified. Two run-time 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 (with type T) if and only if ISTYPE[X, T] = TRUE. Otherwise it raises a run-time 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 v # NIL AND ISTYPE[v, Ti]
   THEN {vi: Ti ← NARROW[v]; ⟨stmtlisti⟩}.
```

Because the object referenced by v is known to have a specific type in each arm, the arm's statements are permitted to examine and/or modify the values of its fields.

In addition to generic reference variables, the Cedar language also has generic procedure types. A procedure type may use ANY as the type of its formal parameter record type and/or result record type. This allows flexibility in specifying a procedure parameter or procedure variable. Procedure values with specific domains and ranges may be widened to these dynamic types, and later tested and narrowed analogously to REF ANY. They must be narrowed before being applied. The inefficiency of the current implementation of generic procedure values prohibits their use for time-critical programming tasks.

Generic reference types allow procedures to store or pass as parameters objects of any reference type and to examine or modify objects of prearranged varying types. However, they 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 run-time type system to manipulate the run-time representations of types. (The type of each statically allocated object can be discovered by consulting the appropriate symbol tables, and a type tag is stored with each collectible object. Parts of these data structures are also needed at run-time to support the garbage collector's reference counting.) In the current implementation these functions are too slow to compensate for the absence of full polymorphism in the language, which would permit values of type TYPE to be passed as parameters and stored in variables. Further discussion of the run-time type system, which is included in the Abstract Machine component of the Cedar system, can be found in Section 6.4, *Abstract Machine*.

## 2.7 Rope, Atom, and List Types

Several other flexible data types based on references have been introduced into Cedar. Notable among them are a variable-length immutable text-string type (known as a ROPE), a variable-length linked-list type (LIST), and atoms (ATOM).

An instance of a ROPE is an immutable collectible sequence of characters. Because ropes are immutable, the sequence of characters denoted by a given rope never changes; that is, every operation creates a new rope. As a result, ropes can be safely shared between programs without concern for storage ownership or synchronization. The rope interface provides a large set of useful operations on ropes, including rope concatenation, rope comparison, subrope extraction, and rope scanning. For efficiency, the standard implementation of ROPE represents a rope as a directed acyclic graph with heuristics that attempt to limit its depth. Therefore most rope concatenation and subrope extraction operations can be performed by manipulating small numbers of these structures rather than copying large numbers of characters. A client can provide a customized implementation of ROPE by implementing a small set of basic operations on the new representation. The combination of convenience and efficiency of Cedar ropes have led to their widespread use at all levels of the system.

The Cedar language has a new type constructor, LIST OF, to declare singly linked lists whose elements are all of the same type. The type definition

```
L: TYPE = LIST OF T;
```

is equivalent to the set of recursive type definitions

```
L: TYPE = REF N;
N: TYPE = RECORD [first: T, rest: L];
```

where N is a "private" name. The record fields first and rest provide the functionality of Lisp CAR and CDR.

The list constructor CONS adds a new element to the beginning of a list. A series of CONSes can be abbreviated using the LIST function. For example, to declare a list of ropes, initialize its value, and then add a new element, one could write

```
colors: LIST OF ROPE ← LIST["red", "yellow", "blue"]
colors ← CONS["white", colors]; --colors = LIST["white", "red", "yellow", "blue"]
```

A list assignment copies the reference rather than the entire structure. As a result, safety considerations require that a LIST OF REF ANY must have limited assignment compatibility. Suppose that variable lora has type LIST OF REF ANY and variable lort has type LIST OF REF T. The assignment lora ← lort is illegal, because it creates a potential aliasing problem. That is, new elements of other types could then be added to lora and accessed through lort. This storage mistyping would represent violation of the storage invariants.

Cedar's atoms are uniquely addressed values in a global name space, 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.

### 2.8 Language Shortfalls

Automatic storage management for a strongly typed language and the flexibility offered by Cedar's new reference data types have worked well. However, the initial design for the Cedar language included several items that are as yet unimplemented. Among these are a formal subclassing mechanism for objects; true polymorphism, which would allow variables of type TYPE; retained frames, which would eliminate restrictions on storing nested procedure values; and a canonical representation of programs that would allow programs as data.

## 3. STRUCTURAL OVERVIEW OF CEDAR

This section presents the concepts and methods used in structuring the Cedar environment as a collection of components that are arranged in layers. The open operating-system architecture developed for the Alto system has had a significant influence on Cedar. Several concepts underlying the structure of Cedar are presented, as well as the philosophies that guided its development. The contributions of existing programming environments for Mesa, Interlisp, and Smalltalk are discussed. Finally, the four layers of the Cedar system are introduced.

### 3.1 Cedar as an Open Operating System

The description of the Alto/BCPL system by Lampson and Sproull [29] defines the closed/open terminology as it is used in this paper.

A *closed* operating system has memory protection, generally in the form of hardware support, which provides separate address spaces for the operating system routines and for each user application. The operating system supplies user programs with special methods for invoking a fixed set of operations. The routines that implement these operations, unless they are also explicitly exported as system operations, are not available directly to client programs. This organization is typical of conventional multiuser operating systems.

An *open* operating system is simply a collection of program modules (containing sets of related procedures) that share the machine's single address space. The structure of the resulting system is determined by policies for organizing, loading, and running these modules. The policies may be imposed by convention or enforced by the system or the programming language.

The structure of the proposed programming environment and the relationship of its components to one another did not arise explicitly in the deliberations that led to Cedar. Rather, the open style is particularly well suited to the hardware architecture of the Dorado and Dandelion processors. The single, unsegmented address space of this architecture makes reliable and effective implementation of a closed-style system difficult. The need to support an integrated, interactive environment for a single user also favored this approach. Thus, an open operating system modeled on earlier BCPL- and Mesa-based systems for Xerox processors was a natural choice as a basis for building Cedar.

As the implementation of Cedar progressed, considerable effort was devoted both to refining the policies for structuring it and to producing the specific detailed organization of its components. As a result, Cedar's structure itself has emerged as an important aspect of the system. Furthermore, the viability of this structure has been enhanced by features inherited or borrowed from Cedar's Mesa forerunners, from other successful interactive environments (notably Lisp and Smalltalk), and from other parts of Cedar.

### 3.2 Structuring Methodologies in Cedar

The evolution of Cedar from its Alto/Mesa roots has been relatively continuous. By the time of the Cedar 4.0 release in March of 1983, the basic organization of the system, as well as the methods and rules for maintaining the organization, were well established. The concepts underlying Cedar's structure can be summarized in the following points:

—Operating system routines can be called as ordinary Cedar language 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 (except in a manner treated in the next paragraph). This restriction is enforced only by convention. Violations can result in system failure due to an attempt to invoke a procedure in a component that has not yet been loaded or initialized.
—Once initialization is complete, a component can supply a procedure value as the parameter to a lower level *service* procedure. The service procedure can later invoke the supplied procedure to obtain information, to report state changes, or otherwise to communicate with the higher level component. This method, which will be further refined below and in Section 8, *Methodologies*, provides a restricted form of what Clark calls "upcalls" [12].
—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 an open operating system the lower level modules remain directly available to clients at all higher levels. An application can generally choose to use components from any level or to replace them with custom-built components (which can still use the standard lower level components).

Without violating these principles, the challenge was to assign components to layers in the most effective way. A number of potentially conflicting objectives guided the organization of Cedar:

—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. Note that "loops" between interfaces make consistent compilation impossible, so they are effectively prohibited by the language. The observation here is that mutual dependencies between implementations also lead to difficulties in managing and understanding the system. Of course, loops

that arise due to upcalls to procedures supplied dynamically by higher level clients can be quite useful, and have not proven troublesome.
—The components located lowest in the structure should provide the most important and widely used facilities needed by other software.
—Subject to the above objectives, components should occupy positions as high in the structure as possible. This makes them easier to develop, maintain, and replace, 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, or these objectives will conflict with each other. Fortunately, in recent versions of Cedar (beginning with Cedar 5.0), these objectives appear to have been met particularly well. A rewrite of Cedar's virtual memory, storage allocation, disk, formatted input/output, file, and directory packages for that release eliminated many of the undesirable dependencies. At the same time, the number of components that could make full use of these important facilities was increased. Dynamically bound upward calls were employed to eliminate some of the loops.

Occasionally, a component was found that appeared to require placement lower in the structure than its dependence on other components would permit. Closer examination usually revealed that the component could be separated into a high-level part, such as one or more display-based or command-style user interfaces, and a lower level package providing a set of functions through a well-defined Cedar language interface. Although the package had to be located fairly low in the structure, perhaps because its services were also needed by other low-level components, the user interfaces could be moved much higher, where they could use a richer set of packages. Section 8.2 expands upon this philosophy of "build packages, then tools" and its implications for integrated program development.

We have not developed any objective measure of the quality of a particular component organization. Subjectively, there is considerable satisfaction with the current organization. The sections that follow describe this organization in sufficient detail for the reader to evaluate these claims.

### 3.3 Influences of the Mesa/Cedar Language

Alto BCPL was a useful open operating system, but it had many shortcomings. BCPL is a typeless language that provides many opportunities for errors that the Mesa/Cedar type system can prevent. Mesa/Cedar's strong type-checking has demonstrably improved the reliability and the ease of development of programs produced for Xerox processors [19].

Mesa/Cedar'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 without fear of the name conflicts and undetected binding errors that could arise in the Alto BCPL world. As we will see in Section 8.7, *Program Development*, tools such as the Tioga editor and the Viewers window package can even be used for testing and debugging their successors, through judicious use of configuration descriptions.

Interfaces and configurations do not provide a complete descriptive technique 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 statically bound upward calls.

A shortcoming common to both Mesa and Cedar language implementations has led to restrictions in the use of some procedure values. A nested procedure N (one declared within the scope of another procedure P) can be passed as a parameter to procedure X, but X cannot save the procedure value N for later invocation. This is illegal because P's activation record, which provides the context for the nested procedure, is not retained beyond the lifetime of P's invocation. Instead, X must invoke N directly as part of its operation. A procedure value that is used in this way, whether nested or not, is known as a *call-back* procedure.

A nonnested procedure R can also be passed to a procedure Y whose purpose is to save R's value for later invocation. A procedure value that is used in this way is known as a *registered* procedure, and Y is often referred to as a registration procedure.

Both call-back and registered procedures can be used to accomplish upward calling, which is often helpful in the orderly structuring of the system. Examples of both kinds of procedures appear repeatedly in Sections 5, *Nucleus*, 6, *Life Support*, and 8, *Methodologies*.

Cedar's lightweight processes, first introduced in the Mesa system, are important to the success of the open operating system style. Processes are not responsible for memory protection, storage management, or address space management. The execution point of a process is therefore free to move from one system level to another, relying only upon the semantics of ordinary Cedar procedure calls. Processes may preempt each other at any time (subject to the protection and synchronization provided by monitors [27]), so that high-priority processes may receive rapid service. Time-slice scheduling algorithms divide the processor fairly among processes at the same level, without cluttering client programs with explicit synchronization code.

### 3.4 Contributions from Interlisp and Smalltalk

Cedar's primary contribution to the evolution of open operating system organization is automatic storage management in a strongly typed language. None of Cedar's predecessors is immune to catastrophic damage or eventually fatal storage leaks that result from improper pointer management—the kinds of unrecoverable mishaps that traditional closed operating systems were designed to protect against. Although closed operating 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 run-time tests. These methods are known to work well in Lisp and Smalltalk implementations. Admittedly, storage leaks, while infrequent, can still occur even in the safe subset of Cedar (see the discussion of cyclic data structures in Section 5.4, *Safe Storage*).

The elimination of concern about the ownership of collectible objects has improved the convenience and reliability of communications between system layers, both downward and upward, through call-back and registered procedures.

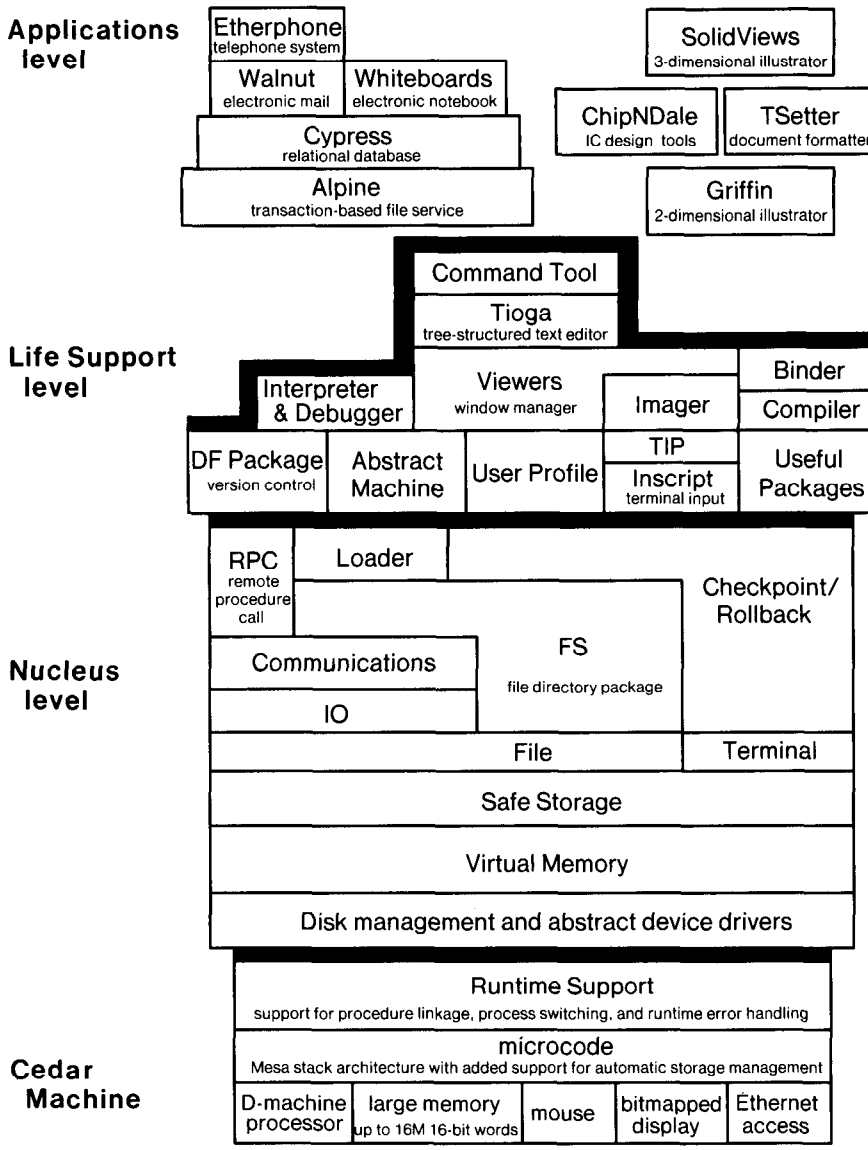


Fig. 2. The structural overview of Cedar.

In Cedar, the automatic 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 system's storage invariants.

### 3.5 Structural Organization of Cedar

Figure 2 presents a structural view of Cedar, as a set of major levels each comprising a set of layered components. Following the methodology developed

in this section, each component is built upon abstractions supplied by components lower in the structure. The figure was designed to express the orderings and dependencies among the components. A component C appears directly above another component D if and only if C uses the facilities of D directly. Block areas imply neither the relative importance nor the relative sizes of the components they represent. The irregular and rather riotous collections of shapes and component positions in each layer are intended to depict Cedar as a system under control but in a state of constant growth and evolution. (Alan Perlis has referred to systems with this sort of character as "organithms" [39].) Note that only a representative selection of Applications components is shown.

The four major Cedar layers are the Cedar Machine—hardware, microcode, and primitives needed to 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.

Sections 4 through 7 follow the organization of Figure 2 (from bottom to top) to present an overview of the major components of each of the four Cedar layers, emphasizing the lower three. This overview provides the factual basis for further discussion of how improvements in programming productivity, integration, software quality, and program development methodology are achieved in Cedar.

## 4. THE CEDAR MACHINE

The Cedar Machine includes the hardware, the microcode, and the primitives needed to execute the Cedar language.

### 4.1 Workstation Hardware

The Cedar programming environment runs on the family of Xerox Scientific Workstations, which includes the Dorado [26] and the Dandelion [25]. The Dorado is a high-performance personal workstation with 16-bit words, a cached virtual memory with a single, large virtual address space (24 bits, word-addressed), and up to 32 megabytes of physical memory (typically 4 to 16 megabytes). The writeable microstore allows customized instruction sets for different languages and environments (Cedar, Smalltalk, and Interlisp). Input/output devices include a large (1024 × 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 using a frame buffer in workstation memory. A typical personal workstation has a local disk of 80 megabytes or 315 megabytes, while servers can support up to four disks.

Cedar language interfaces are provided all the way down to the hardware. The processor, clock, and all I/O devices have Cedar language interfaces that may be used by other Cedar programs.

Cedar workstations operate in the Xerox Research Internet environment, which includes database and file servers, shared printers, name authentication servers, and distributed electronic mail services [3, 6, 9, 34].

### 4.2 Microcode

The Cedar microcode implements an extension of the Mesa machine architecture [25], 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 [48]. A compact program representation saves storage space and contributes to faster execution. The increased locality of a smaller program can reduce cache misses and page faults. In addition, the architecture allows for interleaved execution of several hundred processes. Microcoded instructions directly support a number of the important features of Cedar, among them:

—rapid allocation of activation records from a heap rather than a stack;
—powerful control transfer disciplines that accommodate coroutine and process-switching transfers in addition to conventional procedure calls;
—reference-counted store instructions that are vital to the efficient implementation of Cedar's garbage collection algorithms;
—direct linkages to fault and exception handlers that are implemented as ordinary Cedar procedures.

The fault and exception handlers manage arithmetic exceptions, machine failures, virtual memory faults, programmed traps, and memory references that would violate the storage invariants that support Cedar's automatic storage management. The routine that handles programmed traps provides low-level support for breakpoint and program-tracing activities (see Section 6.4, *Abstract Machine*).

Other microcoded routines implement the primitive operations for communicating with input/output facilities.

### 4.3 Run-Time Support

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

Low-level routines and data structure definitions provide a Cedar language interface to the microcoded processor architecture. Although this component is not written in the safe language, its interfaces are asserted to be safe. Hence, higher level software can be written in the safe subset of the Cedar language and freely use the facilities of run-time support.

## 5. THE NUCLEUS LAYER

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

### 5.1 Device Drivers

Cedar has borrowed from the Xerox Pilot system [40] the notion of abstract device interfaces. 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 defining the abstract behavior of disk storage devices. There are several disk device drivers, one for each type of device, that implement this abstract disk interface.

The Disk component described below can be programmed in terms of this abstract disk interface, without detailed knowledge of the peculiarities of different kinds of physical drives.

### 5.2 Disk

The Disk component provides shared access to disk storage on the workstation. 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.

### 5.3 Virtual Memory (VM)

The Cedar virtual memory (VM) differs in philosophy from its most recent ancestor, Pilot [40]. 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. (Recall that Cedar has a single virtual address space.) VM retains only one history bit per page and uses simple algorithms for page replacement. Experience has shown that these parameters provide adequate performance.

VM also 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. Using this mechanism, input/output buffers and the frame buffer for the color display are fixed in physical memory as needed.

Cedar file input/output is accomplished by explicit operations, rather than by VM mapping actions as they were in Pilot. The resulting 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 almost all of the Cedar system can operate in the virtual memory environment.

Most of the virtual memory package is interrupt-driven, responding to page faults immediately as they occur. One exception is the *laundry* process, an optimization that writes dirty pages to disk before the memory is actually needed. While the operation of the laundry process is not necessary for system integrity, it is critical for system performance. The laundry process becomes aggressive about cleaning memory only during times of high page-fault activity.

### 5.4 Safe Storage

The extensions to the Cedar language for automatic storage management are supported by the runtime type system described in Section 6.4, *Abstract Machine*, a storage allocator (implementing the NEW operator), and a combination of

garbage collection techniques. The allocator and garbage collection methods are supplied by the Safe Storage component.

The allocator stores a run-time type tag in each new object. These tags index run-time data structures that the garbage collectors use to locate embedded references. The garbage collection algorithms were derived from earlier designs by Deutsch and Bobrow [14]. A full description of the revised algorithms appears in Rovner's recent paper [42].

An *incremental* garbage collector runs at frequent intervals, triggered by specific elapsed-time or memory utilization criteria. It operates as a background process with little interference during normal system operation. The incremental collector takes a synchronous snapshot of key system data such as the activation records and then processes the snapshot at its leisure. The manipulation of collectible objects is prevented only during the snapshot, which takes about 12 milliseconds on a Dorado. 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. The reference counts of collectible objects are not adjusted when they are assigned to local variables. Instead it is assumed that any reference found in an activation record is valid (that is, the corresponding object must be retained), even if the object's reference count is zero. This is an important optimization, considerably reducing the expense of reference counting.

A further optimization, called the *conservative scan*, reduces the execution time and complexity of the incremental garbage collector. The conservative scan treats all activation record values that happen to denote addresses of collectible objects as if they were valid references to those objects. For example, some bit patterns for large integer values appear to be references to objects. As a result, some unreferenced objects may 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 essentially all unreferenced storage (it also uses the conservative scan), but monopolizes the machine for between twenty seconds and 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 the trace-and-sweep collector directly. Users may also invoke it manually. Performance monitoring tools exist to understand the use of objects and to locate cyclic data structures (see Section 7.1, *Performance Measurement Tools*).

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. Because the collectors that trigger finalization use the conservative scan, the finalization of an object has a very high probability of occurring but cannot be guaranteed.

### 5.5 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 [29, 40]. In particular, redundant information stored with each file page permits recovery if portions of files or directories are damaged. Only primitive file-level locking facilities are provided, permitting simultaneous access to either many readers or one writer.

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. The Cedar file system and Alpine, a transaction-based file server, are major clients of the File component.

### 5.6 File System (FS)

The Cedar workstation file management and directory package [44] 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. The concepts of naming and storage are separate, though often related. File names can represent either *local* files, where the only copy of the file resides on the workstation's disk, or *attached* files, where the file name is a symbolic path name to a remote file. Read-only copies of entire remote files are retrieved and cached as needed on the local disk. FS provides these facilities by maintaining a *local file name table* and a *remote file cache table* describing the contents of the local disk.

The local file name table, which is implemented as a B-Tree [1] for efficiency of access, 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 subdirectory can be created that describes a complete system or set of related tools consisting of a combination of local and remote files.

The remote file cache table organizes the set of remote files for which local copies exist. Files may be referenced via attachments, which are listed in the local file name table, or via full symbolic path names. 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 the least-recently-used copies of remote files from the cache when additional space is needed. Cache entries refer to specific versions of remote files by name and creation time.

The server part of a file name may be either a real server name or a logical server name. A logical server is a mapping of a logical server name to a single write server and a set of read servers. The idea is to write to the single write server, but to permit reading from any of the read servers, both to distribute the load among the servers and to provide alternatives when a server fails. This requires the replication of important directories on several physical servers.

Replication of updates is not entirely automatic, but one can invoke maintenance procedures periodically to propagate the updates.

The current Cedar file servers prohibit symmetric treatment of remote file reads and writes. Therefore, FS will not accept a request to open a remotely named file for writing. Instead, 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. We will return to this subject in the version and release management discussion of Section 6.11, *DF Package*.

### 5.7 Input/Output Streams (IO)

The IO interface defines an object type called a STREAM. Conceptually, a STREAM is a sequence of items, such as bytes, characters, or records. The IO interface defines generic procedures for creating and using STREAM objects, including useful input scanning and output formatting routines. Cedar contains over a dozen specific implementations of the STREAM class supporting several sources and destinations, among them disk files, the keyboard and display, the Ethernet, and pipes.[3]

The STREAM is a flexible and widely used data type. It is easy to define specialized streams for specific applications. Programs that read and write streams can be written without explicit knowledge of the source or destination media.

### 5.8 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. [6]. 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, a byte stream protocol, 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, and later 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.

RPC includes facilities for dynamic binding. Clients can specify a service or machine to which an interface is to be bound (dynamic import), and servers can dynamically make remote interfaces available (dynamic export). The Cedar RPC

[3] Pipes provide the buffering and synchronization needed to connect an output stream from one process directly to the input stream of another [41].

package, described by Birrell and Nelson [4], performs two additional 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 [36] 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 and voice annotation service, and a "Compute Server." All three are described further in Section 7, *Applications*. 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.

### 5.9 Terminal

Most Cedar applications are content with the higher level display-management and user input facilities supplied by *Viewers* and *TIP* (Sections 6.6 and 6.3). However, more radical applications may need to use the display 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-display bitmap and optional color frame-buffer display memory. Operations are available to switch the use of the physical hardware (and thus the entire contents of the display memory) among the Terminal instances. The standard Cedar display is obtained through the use of just one Terminal instance. Another Terminal instance is employed to drive a much simpler user interface while the system is being loaded.

### 5.10 Running Programs

This section describes the components that are responsible for initializing the system, loading programs, and saving copies of the running system at interesting checkpoints. The methodologies that have been developed to construct a useful environment using these components are discussed in Section 8, *Methodologies*.

A Cedar *boot file* is a compact representation of a set of code and data segments, assigning each segment to some locations in virtual memory. The boot file specifies the virtual memory map so that physical memory can be initialized when the boot file is run. It also specifies an initial execution context (program counter, process, and activation record).

The *germ*, a self-contained program, loads and starts a boot file, and performs other activities that need to be performed when very little is running and when few amenities are available. The germ is small and simple enough to be fetched from the local disk or the Ethernet, installed in memory, and started by the primitive bootstrap capabilities of the host machine's hardware and microcode.

The *loader* is responsible for loading and starting program components during normal system operation. It first assigns a component to virtual memory locations and copies its code segments to the assigned locations. It searches a global symbol table known as the *load state* to locate bindings for the interfaces that the component imports, then records in the load state the interfaces that are exported by the component (for use in satisfying the import needs of programs that are loaded later). Interfaces supplied by a new version of a component supercede older versions, although in the present system existing bindings are not broken.

This binding process is a simplified version of the methods employed by the Cedar *binder* to produce configurations from configuration descriptions (see Sections 2.1, *Modules* and 6.8, *Compiler and Binder*). Finally, the loader executes the initialization code for the component.

At any time the user may invoke a command that stops the system and creates a *checkpoint* file, whose form is the same as a boot file. A checkpoint file captures the state of all active virtual memory, producing a system image that includes running code, activation records, global frames, and other active data. This data includes the display bitmaps, so a checkpoint captures the visible system state as well as the internal state. It does not record the state of files, directories, or other I/O devices. In order to leave the programs managing these and other external values in a clean state at the time of the checkpoint, programs may register cleanup procedures that will be executed just before the checkpoint file is produced.

To use a checkpoint, one boots the system from a checkpoint file instead of a boot file. This *rollback* operation restores the memory configuration of the processor to its earlier state. It then calls another set of registered procedures, which will check the state of the external environment and reverse the effects of the checkpoint cleanup procedures.

### 5.11 Discussion of the Nucleus Structure

Components of the Nucleus that do not have access to the basic memory management facilities provided by 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. Such components should therefore be located as low in the structure as possible. Since the redesign of low-level components that occurred during the Cedar 5.0 release in late 1983, the only component above the Cedar Machine that does not use virtual memory is the virtual memory (VM) implementation itself. Even device drivers and the Disk package, which are located below VM, can address their resident memory buffers using virtual memory addresses supplied to them by higher level initialization routines. VM is so low in the structure that it cannot even find for itself the disk file used to back up memory. During Cedar initialization, the File package is used to inform VM of the backing file location. The simple design of VM makes this possible because neither file directories nor file concepts are required to get VM to work.

Safe Storage resides just above VM, having been placed 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, and therefore can benefit from the increased reliability and convenience that automatic storage management provides. 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 (described in Section 6.4), because IO needed some of the Abstract Machine's advanced features, such as the ability to print symbolically the value of a REF ANY. 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

advanced features as registered procedures. It is necessary for components located between IO and the Abstract Machine to avoid using the advanced features until the Abstract Machine has been initialized.

There are problems associated with moving programs to lower positions in the hierarchy. One such problem 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 arm's length debugging techniques described in Section 8.4, *Installing and Debugging Programs*, must often be used when working in this region.

The placement of other components in the Nucleus and Life Support layers follow similar reasoning based on the structural objectives stated above. Facilities such as the Tioga editor appear within Life Support at levels 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 to find an acceptable initialization order for interrelated programs, or to connect them in such a way that the initialization order does not matter.

## 6. THE LIFE SUPPORT LAYER

The Life Support layer includes most of the standard program development tools in the Cedar environment. In fact, the name Life Support evolved from the notion of a minimal set of tools needed to provide a complete development environment for a new Cedar release. Many of the Life Support components are quite large, providing functions directly to Cedar users or applications programmers, and in this sense they resemble applications more than operating system components. Life Support includes components for the Cedar user interface, such as a display manager, a text editor, command and expression interpreters as well as components for software development and management.

Cedar programmers tend to write their packages, when appropriate, so that the packages can be used in three ways: via a client program interface, via the command interpreter (see Section 6.9), and via a viewer-based user interface (also known as a *display-based* interface). The point is that the full functionality of the package should be made available through the client program interface, rather than hidden under a command or viewer-based user interface (see Section 8.2, *Tools and Packages*). Of course, in some packages it only makes sense to provide a client program interface.

From the Life Support level on up, 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 8.4, *Installing and Debugging Programs*, and 8.7, *Program Development*.

### 6.1 Useful Packages

During the development of Cedar, many generally useful packages have been produced. Examples include packages for sorting arbitrary values, maintaining

symbol tables, and managing a registry of commands. A library of well-designed packages can reduce the need to reimplement common methods for each application. Improved reliability and performance are often side benefits. These packages have been collected as Cedar components in the Life Support layer, low enough in the Cedar structure that they can be used by as many components as possible.

There are several standard symbol table packages in Cedar. A *SymTab* implements a simple one-level symbol table, mapping ropes to REF ANY values. Atoms provide a kind of global one-level symbol table, but a SymTab allows one to limit the scope of names. A *RefTab* provides a table and operations with the same semantics as SymTabs, except that the keys are also REF ANY values. Neither RefTabs nor SymTabs assume any ordering among the keys whose values they store. When such an ordering is important, one may choose to use a *RedBlackTree* data structure [21] instead.

The *Real* package implements a library of numeric algorithms conforming to the IEEE single-precision floating-point standard. The package is compatible with the microcoded REAL operations, which also are implemented to the IEEE specification [23]. The *Random* package generates pseudorandom 32-bit integers on request, beginning with a seed that is derived from the system clock or a client-supplied value.

The *Commander* package is a general registry for user commands. Applications that wish to be driven by simple commands, each composed of a verb with parameters, may register command names and the procedures that implement them. The intent is that tools, including but not limited to the Cedar Command Tool (see Section 6.9), will accept user commands and interpret them by consulting the Commander registry to locate command procedures. The Commander is located at a low level in the Life Support layer so that components can register commands during system initialization, long before the Command Tool and user-level applications have been loaded.

### 6.2 User Profile

A *user profile* is a personalized collection of the custom-tailoring options that applications and packages make available to users. The user profile for a given user is a text file, stored on a file server, that is automatically fetched during user authentication. Among its uses are the specification of the final steps of the full boot process and the specification of standard defaults for packages, such as the name of the default printer. The User Profile package maintains a parsed representation of these specifications for rapid and convenient access by client programs.

### 6.3 Inscript and TIP Tables

Input devices available to the Cedar user include a conventional keyboard, the motions and buttons of a mouse pointing device, a keyset chording device, and a graphics tablet. The *Inscript* package produces a single serial buffer of time-stamped input events from these devices. 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. Events are less likely to be missed and timing of events will be more regular. Each client of Inscript must determine which of the input events are intended for it, ignoring those intended for other clients.

Most applications interpret users' actions through the *Terminal Input Processor*, or *TIP*. TIP uses a finite-state machine implementation to interpret Inscript input events based on easy-to-write specifications that are parsed into TIP tables. For each event (such as keystroke, mouse click, or mouse movement) 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 sequence of action tokens that represent the semantics of the event.

The TIP package assumes that characters entered on the keyboard are intended to be displayed sequentially at a display location previously selected by a user action or by a program, rather than at the present cursor location. Keyboard events are therefore translated into actions specified by an *input focus* representing this preselected location. The input focus can be changed by clicking the mouse, by typing characters, or under program control.

A high-priority process called the *notifier* interprets input events according to the known set of TIP tables. Standard rules determine the choice of TIP table to invoke for each event, based on the current input focus for keyboard events and the cursor's display location for mouse events. The system notifier collects the TIP action tokens and invokes a client notifier associated with the TIP table. Typically, the client notifier creates a new process to carry out the desired action then returns immediately so that the system notifier can react quickly to the next event. In this way the user can initiate or control many concurrent applications. Furthermore, applications can be written in a way that does not preempt the user's ability to choose from moment to moment which application to interact with.

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. Additional user-specified TIP tables may be layered on top of existing TIP tables to give the knowledgeable user the ability to custom tailor an existing application.

### 6.4 Abstract Machine

An original goal for the Cedar environment was to combine a compiled, strongly typed language with the interpretive symbolic power of Interlisp or Smalltalk. The Cedar *Abstract Machine* facility represents a step in this direction.

The Abstract Machine provides descriptions of types and run-time values, programs and program instances in terms of Cedar language semantics. It models active data and program instances in a running program and provides the ability to modify data and some program values. Because the Abstract Machine is a powerful facility whose description has not been published previously, we present its features in more detail.

6.4.1 *Overview of the Abstract Machine.* All of the Abstract Machine's facilities are based ultimately on the symbol tables and program graphs produced by the

compiler, the binder, and the loader. To a client the Abstract Machine appears as a set of abstractions (data types and associated operations) that allows types and run-time values to be examined, picked apart, and modified.

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

*Run-time Types.* For each Cedar language TYPE defined in a running Cedar world, the Abstract Machine provides a unique run-time value called a Type. These Type values are used by all the Abstract Machine interfaces as run-time representations of their corresponding Cedar language types. The same values are also used by the Safe Storage component to label collectible objects.

*Type Information.* The *AMTypes* interface provides procedural access to the names and structures of data types. The interface includes a complete set of operations for discovering name, size, and other attributes about each Type, for following the definition chain of a type to its underlying basic or constructed type, and for analyzing the internal structure of composite types. The Type values are categorized into *classes*. One can determine from a Type's class whether it is a primitive Cedar language TYPE (each has a class to itself), a definition (a name bound to another type), or a type produced by one of a number of type constructors (pointers, records, procedures, and the like).

*Value Manipulation.* Other AMTypes procedures permit examination and modification of run-time values. An object called a TypedVariable (TV) can represent the Type and current value of any Cedar language variable—a local variable, a global variable, a field of a record variable, or a value reached via a pointer or a REF. Operations on TypedVariables support interpretive programs that can manipulate arbitrary data structures. An example is the standard Cedar debugger package that can print a textual representation of any Cedar value. These operations typically take three orders of magnitude longer to execute than the equivalent compiled code operating directly on the same objects.

*Abstract/Concrete Translation.* Operations in the *AMBridge* interface produce TypedVariables from Cedar language values (REF variables, ordinary values, local and global frames). 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. AMBridge also provides inverse operations to extract Cedar language values from TypedVariables.

*Program and Process Structure Information.* The *AMModel* interface contains operations for investigating a program's structure: procedures in terms of their embedded blocks, program modules in terms of their procedures, and 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.

| *AMTypes call* | *Type* | *Class* | *Count* | *Name* |
|---|---|---|---|---|
| TVType[refTV] | 1252 | reference | — | — |
| Range[1252] | 1254 | definition | — | — |
| UnderType[1254] | 1247 | record | — | — |
| NComponents[1247] | — | — | 3 | — |
| IndexToName[1247,1] | 1412 | longInteger | — | "x" |
| IndexToName[1247,2] | 1412 | longInteger | — | "y" |
| IndexToName[1247,3] | 91 | definition | — | "visible" |
| UnderType[91] | 1058 | enumerated | — | — |
| NValues[1058] | — | — | 2 | — |
| First[1058] | — | — | — | "FALSE" |
| Last[1058] | — | — | — | "TRUE" |

Fig. 3. The Abstract Machine permits a client to examine the run-time type information about a running program. These are the results of calling various AMTypes procedures concerning the example in this section.

*Program Control.* The *AMEvents* interface provides a set of low-level operations for setting breakpoints and for tracing program flow.

*Access to Multiple Virtual Memories.* The Abstract Machine uses the *WorldVM* interface for all references to run-time values and to run-time program and process structures. WorldVM supports symbolic access to three address spaces: *local*, the current (running) address space; *world-swap*, a restartable memory image saved on disk; or *teledebug*, an environment accessed using network communications. These arm's length methods are infrequently used, but are invaluable when the local methods fail (see Section 6.10, *Debugging*).

6.4.2 *Example of Abstract Machine Usage.* We present an example to show how clients use the Cedar Abstract Machine facilities. Consider a client program, perhaps a debugging application, which needs to extract the names and types of all the fields in a record. The client is given a REF ANY, named ref, that points to the record. Suppose that the relevant types (as yet unknown to the client) are:

```
Point: TYPE = REF PointRec;
PointRec: TYPE = RECORD[
  x, y: INT,
  visible: BOOL];
```

The results of examining this record structure via the Cedar Abstract Machine are summarized in Figure 3.

To begin, the client uses the AMBridge interface to associate a TypedVariable with the variable ref:

```
refTV: AMTypes.TypedVariable ← AMBridge.TVForReferent[ref];
```

This TypedVariable contains both type and value information for the reference. The type information is examined through the AMTypes interface to determine the run-time type, an AMTypes.Type value, and its

associated type class. In this example, the type classes we encounter will be reference, definition, record, longInteger, and enumerated.

```
type: AMTypes.Type ← AMTypes.TVType[refTV];   - -1252 in this example
class: AMTypes.Class ← AMTypes.TypeClass[type];
                                   - -reference class in this example
```

The client in this example expects a reference class, but in general, the client program would select an action based on the type class. In the case of a reference, one requests the type of the referent through the AMTypes.Range procedure (which is also used to request the range type of arrays). The resulting type class is definition, indicating that the reference has type REF PointRec rather than REF RECORD[...]. The AMTypes.UnderType procedure strips away any layers of type definitions until a nondefinition type is found:

```
refType ← AMTypes.Range[type];   - -type of referent
recordType ← AMTypes.UnderType[refType];   - -strip away type definitions
class ← AMTypes.TypeClass[recordType];   - -record class in this case
```

For this example, we expect a record structure. Records contain several fields; therefore one must determine how many fields are in the record type and process each field iteratively by extracting the field type from the record type.

```
nComponents: NAT ← AMTypes.NComponents[recordType];
FOR i: NAT IN [1..nComponents]DO
  name ← AMTypes.IndexToName[recordType, i];
  type ← AMTypes.IndexToType[recordType, i];
  class ← AMTypes.TypeClass[type];
  - -select processing based on type class of this field
  ENDLOOP;
```

The PointRec record has three component fields. The two INT fields return type class longInteger and the BOOL field returns type class definition, which in turn is defined to be type class enumerated. Enumerated types can be examined by AMTypes procedures to determine the number of values in the enumeration, and the first, last, next, and previous values.

6.4.3 *Assessment of the Abstract Machine.* The present Abstract Machine facilities were designed primarily as a basis for program debugging and development. For this function, they have served very well. Several different approaches to interactive Cedar expression and statement interpretation, source-level breakpoint management, and controls for errant processes have been developed using the Abstract Machine.

A complete Abstract Machine description of Cedar would provide an interpretation for the semantics of the entire language, allowing the specification of the meanings of Cedar programs without any reference to the characteristics of the underlying hardware. A generally useful Abstract Machine implementation would, in addition, be efficient enough to serve as the basis for truly polymorphic language features (the type REF ANY is useful, but insufficient). Although the current implementation does not yet achieve these goals, the Abstract Machine

forms the basis for an interpreter that is fast enough for interactive and debugging use (see Section 6.10).

As it stands, however, the Cedar Abstract Machine is one of the system's more novel components. It has demonstrated that a strongly typed compiled language is not incompatible with the notions of run-time types, programs as data, and other powerful and flexible concepts usually found only in more interpretive languages.

## 6.5 Imager

Interactive Cedar applications rely on the power and flexibility of high-resolution bitmapped display terminals. In earlier Xerox systems, support for interactive graphics was limited to low-level bitmap operations, such as the RasterOp (BitBlt) function described by Newman and Sproull [37]. While one may manipulate bitmaps directly in Cedar, most applications instead use the *Imager*, which is a device-independent graphics package for high-quality two-dimensional imaging of text, line art, and scanned images. The Imager appears quite low in the layered structure of Cedar to permit high-quality graphics in most interactive applications. The Viewers window manager and Tioga editor both make extensive use of the Imager graphics package for their operation.

The imaging model [56] of the Imager is based on the Interpress page description language [58]. The Imager supports the presentation of a variety of image material: text in various fonts, lines and curves of various thicknesses, strokes or enclosed outlines, sampled images, and various color models. Image transformations can scale, rotate, translate, and clip images through simple specifications.

Due to the device-independent design, images may be rendered on a variety of devices, some of which include black-and-white displays, full-color displays, color-mapped displays, laser printers, color printers, film or video recorders, as well as pseudodevices such as Interpress masters, pixel arrays for capturing scan-converted bitmaps, or display lists. The Imager implementation makes extensive use of object-style programming to permit extension to new devices and customization by new graphical applications.

## 6.6 Viewers

Most Cedar applications are intended to be used in a cooperative fashion, sharing the display real estate with other concurrent applications. They do this using *Viewers*. The Viewers abstraction provides an application with a virtual display, keyboard, and mouse. Each viewer is a rectangular region whose position and overall size is 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. Closing a Viewer causes it to appear at the bottom of the display as an icon (a small evocative picture). An open Viewer uses TIP tables to connect the user's input actions and the application-specific functions, serializing these actions when the user reacts faster than the actions can be performed.

In actuality, a hierarchy of viewers exists in the system. The top-level viewers we have been discussing here may include nested subviewers—perhaps to scroll the contents of a subwindow separately, to permit another application to supply the subwindow contents, or to request information to be entered by the user. 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 viewer sharing the column with other viewers assigned to that column. If an auxiliary color display is available, a third column of viewers can appear there. Figure 4 shows a sample of the Cedar displays during editing of this paper. Viewers can either have variable height to share the available space equitably, or can suggest some preferred height. The user can easily 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 provide a predictable model for window placement, to minimize user interaction required for window scaling and positioning, and to achieve high-performance display updating. The underlying graphics facilities also support the more common overlapping-window model.

The Viewers package also serves as a focal point for integrating 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 (see Section 8.1, *Procedure Variables*). A viewer can also be associated with a custom TIP table and with other attachments that customize its operation.

Another interesting aspect of the Viewers implementation is that a window does not have a "process behind it." Rather, processes are created dynamically in response to user actions—often a new process for each action. For most viewer classes, the quiescent state has no processes associated with the viewer. However, the Command Tool described in Section 6.9 has the command interpreter process "behind it," with other processes created dynamically as needed.

## 6.7 Tioga

The Tioga document composition system provides the tools to create and edit formatted documents, including Cedar programs. Tioga documents are tree-structured, with each node corresponding approximately to a paragraph or statement. Tioga nodes can be decorated with user-specified style information that controls their displayed and printed appearance. Tioga is a "what you see is what you get" galley editor; it does not provide automatic support for page makeup.

Tioga displays documents in text viewers, making extensive use of TIP tables to specify 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

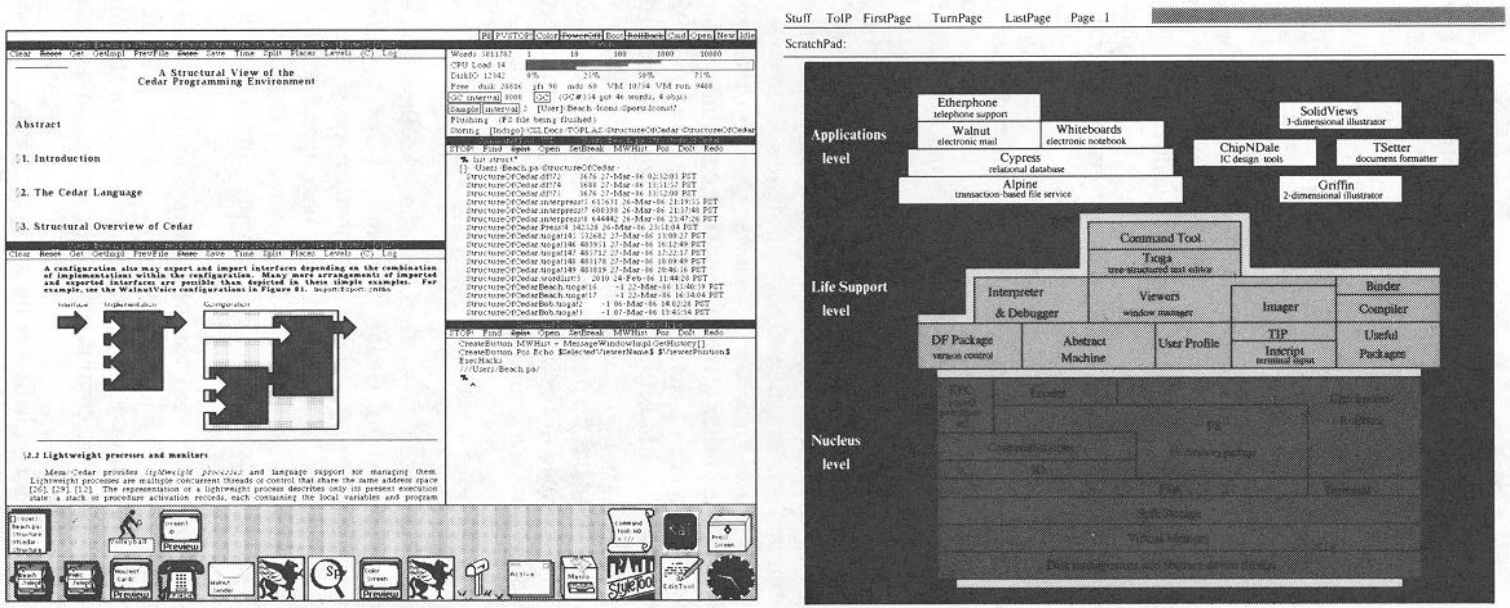


Fig. 4. Sample of the Cedar displays: black-and-white on the left and color on the right. Viewers tile the displays in three columns, icons at the bottom.

like command language interpreters and specialized viewer-based 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 select file name arguments for commands from anywhere on the display, scroll through the command execution history, or invoke a command by copying it from a "recipe-book" document, using only the mouse-driven text-editing and scrolling operations of Tioga.

Although Tioga does not understand Cedar language syntax, we find that using Tioga as a program editor has several important benefits. First, viewing programs as formatted documents with common typographic 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 programming environments.

—Simple pattern-matching allows Tioga's abbreviation expansion command to construct templates for language constructs and procedure call parameters. Tioga's hierarchical node structure allows 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 other modern syntax-directed editors [13, 50, 57].

—Tioga also performs the use-to-definition portion of the Masterscope functions in Interlisp [53]. A selection of the form interface.item may be used to request a new viewer displaying the file that defines or 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. That would require the capabilities of *Cedar system modeling*, a partially implemented extension to the DF Package (see Section 6.11).

—Tioga's client interface permits the Cedar debugger to show a breakpoint or error location as a highlighted region in a source file viewer, so that the user can see procedure and variable names in context. By using the ability to copy text freely among viewers, the user can copy expressions from the source to the debugger area for interpretation. This contrasts with the method used by the Blit debugger [10], which constructs menus of currently visible procedure, variable, and field names to ease user input.

In this paper we have not emphasized user interface issues. Teitelman's paper [51] 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 [2].

### 6.8 Compiler and Binder

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* 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 timestamps of exported interfaces match those specified by the components that import them.

## 6.9 Command Tool

Cedar Life Support includes a conventional command interpreter in the form of a text viewer into which the user types commands and into which the system responds with results. The command syntax, an amalgam derived both from the UNIX shell [7] 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).

The Command Tool registers a small number of initial 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 Commander package (Section 6.1), supplying procedures that extend the set of available operations. One such application is the Interpreter (Section 6.10), which registers a command to evaluate Cedar language expressions. 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 use may also create more than one Command Tool viewer, then issue commands in each that may run concurrently.

The Command Tool inherits from its environment several capabilities that make it more useful. Perhaps the most noticeable is the full availability of the Tioga editor for constructing commands. This includes copying commands from other sources, such as earlier points in the same Command Tool's text script, other Command scripts, or any other Tioga document.

The Command Tool also uses *property lists* (lists of key-value pairs, where the keys are usually atoms) to provide a reasonably efficient dynamic binding mechanism. Each Command Tool viewer has a separate property list. Standard Command Tool properties include command-lookup rules and directory search lists, working directories to use as a default in evaluating file names, statistics gathering procedures to be applied before and after every command is run, and default input and output streams. This property list is available during command parsing and during the execution of the registered procedure that implements each command (it is passed as a parameter when a registered command procedure is called).

## 6.10 Source-Level Debugging

Cedar's source-level debugging facilities are a good example of the method described in Section 3.2, *Structuring Methods*, that splits an application into a low-level, widely available package and one or more higher level user interfaces that are clients of the low-level package.

The *expression interpreter* accepts a rope value representing a Cedar language expression and a context value representing an execution context. It evaluates

the expression in that context, returning a TypedVariable as a result. The expression interpreter is located just above the Abstract Machine. In fact, it should probably be designated as a high-level extension of the Abstract Machine, whose facilities it uses to interpret the expression. The expression interpreter is used in a number of applications to obtain the effect of dynamic program composition where high performance is not required. In addition, specialized diagnostic routines can be built for specific purposes by calling on the facilities of the Abstract Machine and the interpreter.

The standard Cedar debugging tools rely not only on the Abstract Machine and the expression interpreter, but also on Viewers, the Tioga editor, and the Command Tool. They are therefore located much higher in the Life Support layer. An *interpreter viewer* is one of the basic programming tools. It can be created at the user's request, or when a process stops due to a breakpoint or an uncaught exception condition. The user can examine the process execution stack, evaluate expressions, and examine or modify variable values. A menu button instructs the debugger to locate the suspended execution point for some activation record and to highlight the corresponding source statement in a separate Tioga viewer. Breakpoints, specified by selecting a statement in a source viewer, may be similarly set and cleared from compiled code. Interpreter functions are also available directly from the Command Tool.

The *Debug Tool*, another viewer-based tool, exhibits the state of all the running processes, configurations, and their components in the system. The Debug Tool can selectively suspend running processes and can open an interpreter window set to the top of a suspended process's stack. It is valuable in locating process deadlocks (resulting from improperly used monitors) or runaway processes.

All of the debugging facilities may be applied to a remote machine's memory environment by using teledebugging techniques, or to a suspended world-swapped environment (see Section 6.4).

### 6.11 Tools for Version and Release Management

File and version management in Cedar is made tractable by a suite of packages and tools built around *description files* (DF files). Details about DF files and how they work are included in Schmidt's thesis [43]. A brief description here is followed by a discussion in Section 8.5, *Release Management*, of how DF files are used.

A DF file describes the set of source, object, configuration description, and ancillary files required to construct and document a Cedar system component. The DF file lists the files that define the component by specifying their names, fully qualified with their network locations and create dates. In addition, it lists other DF files (and specified files within each) that describe any additional components needed to compile, bind, and use the component.

The *DF Package* supplies three primary operations. The *bringover* procedure operates on DF files to copy its defined and included files onto a local file-system working directory (by establishing attachments to the remote files). The *storeback* procedure, using a DF file as a guide, copies changed files to their designated remote locations, while revising the DF file to reflect the changes. The *verify* procedure examines the contents of a DF file for consistency and completeness, reporting problems such as missing files, superfluous files, or files with incorrect

versions. A DF file that is verified without errors provides a consistent and complete description of the component.

DF files are also used to describe the sections and figures of a paper or book, the drawings and wiring lists specifying a hardware design, or any other collection of related files.

One client of the DF Package is a program that fetches critical files during system initialization. As a result the DF Package occupies a low-level position within the Life Support layer. Its standard user interfaces (one a specialized viewer-based interface, the other a set of Command Tool operations) are run as ordinary applications.

A variant of the Cedar DF Package has been adapted for use in the Xerox Development Environment (XDE) [47].

## 7. CEDAR APPLICATIONS LAYER

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 originally developed as applications are often evaluated, modified, and then incorporated into lower levels, usually into the Life Support layer. Other components are more clearly user-oriented programs providing functions for specialized needs. However, even among components in the Applications level, the packages are layered into various abstractions. A detailed example of this layering appears in Section 9.2, *WalnutVoice Case Study*.

Space does not permit a 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 that Cedar supports.

### 7.1 Performance Measurement Tools

Cedar programmers have implemented an array of performance monitoring tools and debugging enhancements. The *Spy* (a descendant of the Mesa Spy [33]) monitors CPU usage, storage allocation, or page-fault performance by recording the call stack of the active process at specified intervals. *Celtics* uses very low-cost breakpoints to display statement execution counts dynamically. This provides some of the benefits of interactive debugging even within high-priority processes or time-sensitive code. The *BreakTool* provides enhanced breakpoint facilities, such as program tracing and conditional breakpoints. In addition, it supports the evaluation of arbitrary interpreted expressions at specified program locations, thus allowing a mixture of compiled and interpreted code during program development.

The *Watch* tool maintains and displays statistics on selected system resources and events. Among the data displayed are statistics on file system activity, virtual memory utilization, Ethernet communications, disk utilization, and garbage collection activity. *Pupwatch* logs Ethernet packets transmitted to and from a selected host and displays them in a viewer.

Additional packages provide statistics and examination tools for Cedar data structures. *SweepCollectibleStorage* enumerates all collectible objects. *ExamineStorage* looks at all collectible objects and reports statistics. It can also be used repeatedly to track the changes in the number of objects by type.

*CircularGarbage* discovers circular data structures that are otherwise unreferenced. Given a collectible object, *RecursivelyNil* applies itself to all of the references within the object and then assigns NIL to all references found in this way. This breaks circular references and generally improves garbage collection performance.

### 7.2 Database Support

Cedar includes a number of packages that support database applications. *Alpine* furnishes a transaction-based network file service [9] for Cedar. Two major Alpine features are page-level locking of data and support for multiserver transactions. Alpine supplies a file system directory and implements the same Pup file transfer protocol (FTP) used by our other file servers. An Alpine file server acts as one of the multiple pseudoservers for the Cedar release directories, providing backup in the event that the regular file servers fail.

The entity-relationship database package *Cypress* [11] runs in a user's workstation but stores its database on Alpine servers. Cypress implements a simple semantic data model, as well as supporting most of the features of a relational database.

### 7.3 Electronic Mail

The *Walnut* electronic mail system operates in conjunction with the Grapevine message transport mechanism [3]. It stores message databases (using Cypress) on Alpine file servers. Messages are acquired from Grapevine and stored in the database, where they may be moved between message set categories, answered, forwarded, deleted, archived, or unarchived. Walnut users typically use multiple viewers to display message sets, which contain message headers, and messages. Within each viewer they use the standard Tioga editing functions for composing new messages and for filling in message-reply templates.

The Walnut electronic mail system is an example of a layered package in the Applications level. Walnut provides the user interface and mail database implementation, but uses Cypress to manage its database. Cypress in turn uses the Alpine file server to manage its files.

### 7.4 Whiteboard

The *Whiteboard* package [17] is used to organize information spatially in a collection of viewers. Subviewers of various kinds (text, icons, tools, illustrations, mail messages, and even other whiteboards) can be arranged by the user. The user can select almost any viewer on his display and add it to a whiteboard.

Whiteboards have been used to organize large collections of useful information, such as an introduction to Cedar and project information for the CSL research notebook. Large whiteboards use nested whiteboards to cross-reference information.

The whiteboard contents and their layout are saved on a Cypress database, and may be concurrently displayed or modified by several users using their own workstations. Concurrent updates to the same whiteboard are detected by the database system.

### 7.5 Imaging Tools

Of the numerous imaging applications within Cedar, we describe only a few here. The *Griffin* illustrator produces single-page color illustrations from line drawings and shaded areas. The *SolidViews* three-dimensional solid-modeling illustrator creates synthesized graphical objects and renders them with various lighting and texture mapping techniques. The *ColorTool* provides an interactive color selection tool in which the user manipulates the color of a patch by using a variety of color systems. The *Magnifier* provides a handy tool for demonstrations because it can magnify any region of the black-and-white or color display. *PreView* builds viewers for various types of printable file formats on the display, such as scanned images and Interpress [58] files. Additional tools include a digital darkroom for manipulating scanned images, including two-dimensional fast Fourier transforms for image enhancement. All of these tools create images to be rendered on a variety of imaging devices: displays, laser printers, color proofing systems, videotape, and film [2].

### 7.6 Etherphone

The experimental *Etherphone* system [46] uses Ethernet communications to transmit digitized voice. The system consists of microprocessor-based electronic telephones, a centralized switching server, a voice file server, and workstation programs to support voice communications and voice recording services. From a workstation, a user can place and receive telephone calls, maintain private telephone directories, and manage a database of voice messages. A voice annotation package allows voice to be added to Tioga documents and provides simple voice-editing functions. In addition, a commercial text-to-speech synthesizer exists as a server in the Etherphone network. The synthesizer allows the system to "speak" text, initiated either by the user (perhaps by selecting the text in a viewer) or by a program (such as speaking an error message or proofreading a document).

### 7.7 Compute Server

The *Compute Server* [22] uses the remote procedure call protocol to coordinate the assignment of computing tasks to processors with compute cycles to spare, and then to manage the execution of these tasks. Initial clients of the compute server are two typesetting packages, the compiler, MakeDo (see Section 7.10), and a three-dimensional image-rendering package.

### 7.8 VLSI Design Tools

Hardware design researchers have produced a suite of integrated VLSI design, simulation, and analysis tools that operate in the Cedar environment. *ChipNDale*, an interactive VLSI layout tool, makes use of multiple viewers (including the color display), uses extensive parallel processing for interactive tasks, and serves as a focal point for the integration of VLSI design tools. Tools integrated with ChipNDale include a circuit extractor, a CIF file generator, a mask generator, plotting packages, and design rule checkers. The simulation package *Rosemary* and the timing analysis package *Thyme* both accept circuit designs created by ChipNDale.

### 7.9 ViewRec

Some application-level packages are intended to be used within other packages. One example, *ViewRec*, constructs a simple viewer-based user interface given any Cedar record value (including local activation records and global frames). ViewRec determines the names and types of all top-level fields and then displays them symbolically. The client may expand REF-containing fields into additional ViewRec viewers. The user of such a viewer may modify fields by editing their visual representation, may compose parameter values for any procedure values displayed, and may invoke the corresponding procedures. Some prototype applications use ViewRec viewers exclusively for their user interfaces because they are so simple to construct and so easy to use.

### 7.10 MakeDo

*MakeDo* automatically determines dependencies between files and issues the commands needed to bring derived files up to a consistent state. The basic dependencies that MakeDo understands include compilation, binding, and the automatic generation of RPC stub modules from interface modules (see Section 5.8, *Communications*, for the description of RPC). The MakeDo implementation extends to dependencies among files of arbitrary types. As a result it is also used within the VLSI design automation process described in Section 7.8, *VLSI Tools*. MakeDo was inspired by the UNIX *make* command [18], although *make* relies on the user to supply the dependencies and the commands to reestablish consistency.

## 8. CEDAR PROGRAMMING METHODOLOGIES

We stated in the Introduction that flexible program development methodologies were among the four important benefits to Cedar users, the others being improved programmer productivity, integration of software systems, and improved quality of software. In this section we focus on some of the methodologies that have been developed for using the Cedar language, programming packages, and tools to produce experimental programs and to manage the resulting collection of software. We have included representative methods that we believe to be carefully designed and documented. These methods are generally well understood and well accepted within the Cedar programming community. The remainder of this section discusses the following topics:

—using procedure variables
—producing tools and programming packages
—integrating user interfaces
—installing, running, and debugging programs
—version and release management
—software browsing
—developing and testing programs

This is a discussion of what people do with the system. As such it will inevitably provide insights into the ways in which Cedar achieves the other important benefits claimed for it in the Introduction. However, we have deferred until the Conclusions an analysis of how and how well the system meets these objectives.

### 8.1 Methods for Using Procedure Variables

Cedar language interfaces and configuration descriptions are flexible (although static) methods for binding clients to specific implementations of the interfaces they import. However, even these flexible binding methods are not sufficient in some circumstances. Three popular methods have evolved for using procedure variables to achieve more dynamic behavior, without eliminating the protection and convenience of strong typing: call-back procedures, registered procedures, and object-style procedure invocation (see Section 3.3, *Mesa Contributions*). These uses of procedure variables allow higher level clients to affect the operations of lower level components in ways that relax the hierarchical layering imposed by Cedar's structural rules.

8.1.1 *Call-Back Procedures.* Call-back procedures are most often used as an enumeration technique. The underlying package enumerates a structure whose representation is unknown to the client. The action to be performed on each element of the enumeration is supplied by a client procedure. Thus the call-back mechanism extends a lower level component of Cedar to include the client semantics during the invocation of the component.

An example of the use of call-back procedures for enumeration appears in the Rope package. The representation of a Cedar rope is private to the Rope package. A client that needs to perform some action for each character of a rope r can call Rope.Map[r, actionProc], where actionProc is a client-specified call-back procedure. The map procedure applies actionProc to every character in the rope. Other situations where enumerations are used frequently include file directories, Walnut messages, and various ordered data structures.

Another use of call-back procedures occurs in the Imager graphics package, where two levels of call-backs are used to implement generalized paths. The Imager wants to construct a geometric path from coordinates in the $x$–$y$ plane. The client of the Imager's generalized paths has some data structure from which the path can be extracted. The client provides a path procedure, and the Imager calls the client back to request coordinate values by supplying *its own* call-back procedures for the client path procedure to insert coordinates into the path.

8.1.2 *Registered Procedures.* Registered procedures are used in two functionally similar but conceptually different ways: to enable *event notification* and to achieve what might be called *behavior modification.* Two examples should suffice to explain and distinguish these techniques.

Event notification allows Cedar packages to notice when a workstation is not being used. When there has been no user activity at a workstation for some time, Cedar blanks the display and enters an idle state until a user again requests service. The implementation of the Idle interface notifies interested clients whenever the system goes idle or becomes busy again. (For instance, Walnut does not retrieve new mail while the workstation is idle.) To receive these notifications the client supplies a notification procedure by calling a registration procedure in the Idle interface, which adds the notification procedure to a list. The procedures on this list will be called with a parameter specifying either the becomingIdle or the becomingBusy event whenever the idle state changes. Other examples of event notification include notifications that the Viewers package

generates when something important happens to a viewer (such as creating it, destroying it, or saving the file displayed in it), and notifications that the system is about to be—or has just been—booted, "rolled back", or power-cycled.

The Tioga editor uses behavior modification to add functionality to its menu buttons. When the user pushes the Get menu button attached to a text viewer, the Tioga editor uses a file name selected by the user to identify the next file to edit in the viewer. If the name is not a fully qualified file name, Tioga will look for the file on a single local working directory before giving up. If unsuccessful, Tioga is willing to call any registered procedures that have been supplied to modify (or extend) the behavior of Get. One such procedure consults system symbol tables to locate the fully qualified file names of source files that are part of the Cedar release (see Section 8.6, *Browsing*). More sophisticated extensions to the functionality of the Get operation include Tioga's use-to-definition capabilities (Section 6.7, *Tioga*).

A common design for registration interfaces allows the client to supply data at registration time that enables the called procedure to access the client's state information. This is necessary because Cedar does not support retained execution frames.

8.1.3 *Simple Procedural Objects.* Several variations of object-style programming exist in Cedar. All are based on the same underlying implementation of an object: a reference to a record that contains both instance data and procedures defining the operations on that data (see Section 2, *The Cedar Language*). Procedure variables and strong type-checking make this approach safe and effective. By convention, the object itself is supplied as the first parameter to each operation. The operations for a particular object type are defined along with the object type within a Cedar language interface.

Procedural objects extend the functionality defined in an interface by permitting new implementations to be provided dynamically at object-creation time.

For example, Cedar ropes are implemented as references that denote text sequences either directly or as client-supplied procedural objects. A ROPE object is defined by three operations: fetching a character, mapping a specified action onto all the characters, and appending characters to the ROPE object. All other ROPE operations are defined in terms of those supplied operations.

As an example of a client-supplied rope object, consider a rope that represents the characters of the ASCII character set in collating-sequence order: that is, the sequence 256 characters long, beginning with the character '\000, and ending with '\255. This rope can be implemented as follows:

```
Byte: TYPE = [0..255];
MyFetch: PROC [base: REF ANY, index: Byte] RETURNS [CHAR] = {
  RETURN [VAL[index]]};   -- VAL coerces a number to a CHAR
MakeAsciiSet: PROC [] RETURNS [ROPE] = {
  RETURN [Rope.MakeRope[base: NIL, size: 256, fetch: MyFetch,
          map: NIL, append: NIL]]};
```

MakeAsciiSet creates a rope that can be used in the same way as any other rope. (The map and append operations default to implementations using fetch.)

Although it behaves exactly like a sequence of 256 characters in rope operations, it occupies less space. Client-supplied ropes can also be used to make an entire file behave like a rope (via appropriate file accesses and buffering), so that rope functions can be applied to arbitrarily large files. Tioga relies heavily on file ropes.

Similarly, Viewers and IO stream objects use procedural objects for client-supplied window and stream abstractions.

8.1.4 *Object Classes.* An important attribute of most object-style programming languages is that each object instance is a member of a *class*, the members of which share the same behavior and may share some global state. In Cedar a class (a set of classes conforming to the same interface, actually) is defined by a record type containing the operation procedures. An object type is defined by a record type containing instance-specific data and a reference to the operation record. Every member of a class shares the same operation record. Two examples discussed here are IO streams and Viewer classes.

Cedar character streams are implemented as classes conforming to the interface specification IO.STREAM. Several components in the Cedar release provide implementations for standard stream classes, such as file streams and keyboard streams. A client may easily implement an additional stream class suited to the client-specific purposes by supplying a new operation record and registering the new STREAM class.

Cedar viewers are implemented as a set of *viewer classes* defined by the interface specification ViewerClasses.Viewer. As an added convenience the Viewers package maintains a registry of named viewer classes. A viewer class implementation provides operations to initialize a viewer, to save its contents, to destroy a viewer, to paint its contents on the display, and so on. Standard viewer classes implement viewers supporting the Tioga editor ($Text class), the Command Tool ($Typescript), generic support for viewers consisting of nested viewers ($Container), and a few others ($Button, $Label, $Rule). Clients create new viewer instances by calling ViewerOps.CreateViewer[className]. They can also implement additional viewer classes.

Operations on viewers are implemented in terms of extensive facilities in the Viewers package. The ViewerClasses.Viewer record defines data fields, such as size information and display coordinates, that are common to all viewer classes. It also provides a clientData field, which is a REF ANY that the class implementation may use to store implementation-dependent instance data. For the cost of a NARROW statement to validate the clientData value in the implementation of each operation, the flexibility of multiple object classes is obtained without compromising type safety.

Object classes in Cedar are defined by convention, not by the Cedar language. Multiple inheritance, hierarchical or otherwise, can be achieved through direct program manipulation of copies of the procedure records that specify existing classes, but the system provides no direct support for these concepts.

## 8.2 Methods for Producing Tools and Programming Packages

We have said that software integration provides leverage in building new systems out of previously existing components. Cedar interfaces are a useful tool in structuring each component so as to promote integration.

At a low level, Cedar interfaces provide the mechanism for strong typing in modular programs. Checking by the compiler, binder, and loader ensures that types are used correctly across many modules. But Cedar interfaces also act as a form of specification language, describing the public behavior of an abstraction. Programmers are encouraged to provide functionality that will be useful to client programs.

8.2.1 *Build Packages, then Tools.* An important key to the integration of Cedar applications is captured in the admonition to "build packages, then tools." The developer of a user application is encouraged to express its full functionality as a Cedar language interface, to build the implementation of the interface, and to produce the intended user interface as its first client. Whether the application's complete functionality or its user interface is designed first, and whether the package is designed in a top-down or bottom-up fashion, "build packages, then tools" is intended to describe the end result of the design process.

This approach fosters integration in a number of ways. It enables an application to be driven by more than one user interface. For instance, the functions of the DF Package are available through a viewer-based interface and through Command Tool operations. More important, the existence of a programming interface makes it possible to compose higher level application packages directly in terms of the lower level functions. It is not necessary to simulate the input/output behavior of lower level user interfaces nor to borrow and customize the lower level source code. The resulting reuse of packages provides traditional software engineering benefits: bug fixes and performance improvements need only be made once to apply to all uses. An extensive example is presented in Section 9.2, *WalnutVoice Case Study.*

Multiple uses for packages also encourages good craftsmanship in their construction. First, it encourages the programmer to design the component's functionality carefully, aiming for a complete and comprehensible set of data types and operations. Examination of a programming interface tends to point out missing functionality before the package is released to users. Second, it encourages attention to the efficiency of frequently used special cases.

This design process sometimes creates difficulties when an application does not lend itself to a simple hierarchical decomposition into layered abstractions. It often requires several iterations to arrive at the proper partitioning of functions among interfaces. The results have generally proved worth the effort. This careful reworking of interfaces and implementations has consumed a considerable fraction of the Cedar development effort, but it is also in large part responsible for the quality of the result.

8.2.2 *Layered Design of Interfaces.* The design of interfaces within individual Cedar components parallels the layered structure of Cedar as a whole. Within typical, large Cedar components there are four levels of interfaces: the user interface, the event notifier, the client programming interface, and internal interfaces. At the interactive *user interface* level, the TIP package maps keyboard clicks and mouse movements into action tokens. The *event notifier interface* maps these action tokens into registered procedure calls on the application's *client interface.* The client interface defines the actions to be performed by the application package. *Internal interfaces* supply operations within the application

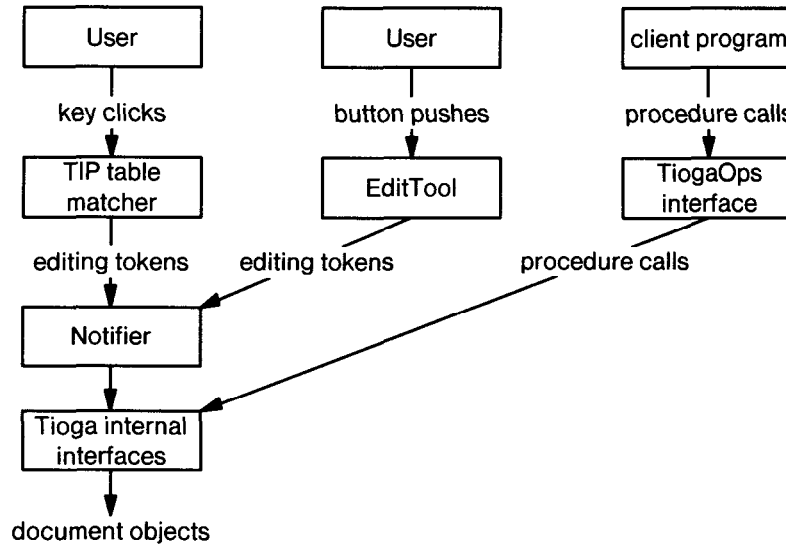


Fig. 5. Layers of interfaces in the Tioga editor.

when external invariants imposed by the client/implementation interface are already met.

The Tioga editor is a canonical example of these layers of interfaces, presented schematically in Figure 5. The Tioga TIP table maps user actions into editing requests. The Tioga notifier receives these actions and calls registered editing operations, while maintaining an edit history log of editing events. The client-callable TiogaOps interface performs the same actions as the notifier but may be invoked by a program rather than a user interface. Finally, the Tioga internal interfaces assume that appropriate locks and invariants are already established and perform the basic editing operations on the document structures.

Other applications may use functions of the Tioga editor at any of these levels. The *EditTool* is a viewer-based user interface that supplies buttons and a macro expander to invoke the editing operations. The Walnut electronic mail system uses the TiogaOps interface to prepare message-reply templates automatically.

## 8.3 Methods for Integrating User Interfaces

The sharing of facilities and resources that we have described from a programming standpoint is also evident to the Cedar user. At the user interface level it manifests itself in a somewhat different way. To the user it is immaterial how programs are structured or how much code is shared. What is significant is the hierarchical structure of Cedar viewers on the display and the commonality of the user interfaces within different viewers that have similar appearances and functions.

The most immediately obvious example is the common behavior of pushbutton-style menu actions within all kinds of viewers. In fact, often the same button,

representing the same behavior, appears in more than one class of viewer. For instance, the Tioga Find button appears in document viewers, typescripts, and Walnut message set viewers. The use of the mouse to select text or other objects is also treated in a consistent manner.

Another important example is the universal availabilty of the Tioga text editing commands. Tioga's facilities are available within text editor viewers, within typescript viewers, and within any viewer or subviewer that contains user-modifiable text. Using only Tioga actions, text can be copied or moved from one viewer to another—from document to document, or from a document into a Command Tool viewer, where the text contents will be executed as the next command.

### 8.4 Methods for Installing, Running, and Debugging Programs

To this point, we have described Cedar as it exists in the steady state, with all system and user programs already present in virtual memory and properly initialized. Clearly, we need methods for getting started and for introducing new programs into the system.

In a typical closed operating system, running a program is straightforward: the system loads a monolithic program image into the address space of a new process and starts it up. In Cedar there are more choices and some additional difficulties. One choice would be to require that each user's system be fully described by a single Cedar configuration. A bootstrap loader would assign each module to virtual memory locations and start the system. This method alone would not allow adequate flexibility in the choice of components to run, nor would it allow the incremental introduction of new or modified programs. An extreme contrasting approach would be to introduce components one at a time, binding them to components introduced earlier and starting them as they are loaded. If the entire system were built in this way each time the system was initialized, the process would take too long and require inordinate care in choosing the loading and startup order. Here we describe the middle course between these approaches that is used in Cedar.

8.4.1 *Cedar Installation.* Cedar is distributed to users as a file containing the Cedar microcode, a file containing the germ (an initialization program, see Section 5.10, *Running Programs*), a single boot file containing the Cedar code for the Cedar Machine and the Nucleus layers, and a large collection of Cedar configurations representing the Life Support layer and widely used applications. These files are installed by a small program obtained from either the Ethernet or the local disk by a tiny sequence of bootstrapping microcode that is built into the machine. To boot Cedar, this program loads and starts the germ giving it the address of the Cedar boot file.

The Nucleus includes enough of the system to manage memory, to read and write files, to send and receive Ethernet messages, and generally to support ordinary Cedar programs. The Nucleus also includes the loader. Shortly after it is started, the Cedar boot file consults a list of the Life Support packages to be loaded and started. The user can override the default package list with a customized one. Once the Life Support layer has finished

boot-strapping itself, the display has been initialized and the environment is ready to support applications.

Finally, a file of user-set parameters called the user profile (see Section 6.2) is retrieved from the user's file server directory. This profile includes specifications of additional components that need to be fetched and possibly started in order to produce the user's desired standard configuration. Once these steps have occurred, the system is ready for use. This entire process is known as a *full boot.*

8.4.2 *Checkpoint and Rollback.* Performing a full boot is a flexible and effective method for producing custom-designed systems, but it takes a long time—two to ten minutes on a Dorado, even longer on slower processors. Moreover, it does not permit one to save the results of any initialization activities which might also be lengthy operations nor to capture a particular configuration as the evidence of program failure. *Checkpoint files* (see Section 5.10, *Running Programs*) address these needs. Having performed a full boot and perhaps performed one or two manual initialization steps, standard practice is to produce a checkpoint representing the resulting system state.

Most often, one begins a Cedar session by rolling back to a previous checkpoint, a process that takes less than a minute. Full boots are needed only when testing new versions of packages in the checkpoint, or when new versions are released.

8.4.3 *World-Swap Debugging and Teledebugging.* Checkpointing methods can also be used for debugging. Most problems can be diagnosed using debugging tools that operate as ordinary processes in the same address space (or *world*) with the target applications being debugged. But when problems develop that prevent the proper operation of the debugging capabilities, low-level facilities in the germ can be manually triggered to store a special checkpoint image, then to boot a separate debugging environment that can examine the state of the saved image. This action is called a *world swap.* If the problem can be repaired in the saved image, another world swap will allow execution to proceed. Otherwise, the next logical step is a rollback or a full boot.

A popular alternative to world-swap debugging, *teledebugging*, is also supported by the germ. Normal system operation can be suspended and control transferred to a germ program called the *debug nub*, which expects low-level debugging commands in the form of Ethernet packets from a debugging program running on another machine. These teledebugging primitives include fetches and stores of specified memory locations, breakpoint manipulation operations, and a proceed command. The suspended system itself can be used as the target system image, in place of a checkpoint file. Teledebugging can be much faster than world-swap debugging, especially since breakpoints require many world swaps, and the world-swap operation is measured in tens of seconds.

## 8.5 Methods for Version and Release Management

Managing the thousands of files that constitute the Cedar system requires automated assistance. Several tools and supporting conventions make the development of Cedar components an orderly process. In Section 6.11, *DF Package*, we introduced DF files, which describe a collection of related files for a package or application. Just as interfaces and configurations impose a checkable hierarchy

that makes programming manageable by reducing the amount of information that the programmer must deal with at one time, DF files form a similar hierarchy for describing and managing the files that make up the system.

The Cedar release methodologies utilize DF files and file-naming conventions to enforce the structure of Cedar among many programmers and across many system releases.

8.5.1 *How DF Files are Used.* A typical modification of a package proceeds by first performing a bringover operation to copy the DF file and all the files that it specifies into a subdirectory on the workstation. Because the resulting directory entries are only attachments, this process is quite rapid. Next, selected sources are edited, compiled, bound, and tested. In most cases the DF file serves as the only input the MakeDo program (see Section 7.10) needs in order to perform the compilation and binding operations automatically, processing only those files that have changed or whose dependencies have changed.

The modifications create new local files that are not visible to any other Cedar user. When the modifications have been completed and tested, the DF Package is invoked to store a new consistent version of the files on the file server and to verify their completeness and consistency. The problems of managing the files comprising a component, determining the actions needed to produce new versions, and managing multiple versions have been reduced from earlier fully manual procedures to the simpler problem of maintaining a correct DF file description of the component.

The concepts underlying DF files have been extended to serve as a full description of a running program. These *system models* [28, 43] may eventually form the basis for automatic recompilation, run-time module replacement, and answers to queries about a program's structure, such as the locations of all uses of some variable (similar to Interlisp's Masterscope).

8.5.2 *Cedar Release Management Methodologies.* A Cedar release is a consistent version of the system specified as a set of DF files, each describing the files contained in one of the release's components. The Nucleus and Life Support layers are expressed as umbrella DF files that list the DF files for their components. These umbrella DF files are verified to ensure that they describe a consistent and self-contained set of interfaces and implementations. The release of application packages is managed separately with slightly different conventions, as described below.

The methodology for releasing Cedar has evolved considerably. Early laissez-faire methods gave way to more tightly controlled ones. All of the files in a release reside in a single tree-structured file directory replicated on one or more file servers. Standard file-naming conventions for storing Cedar DF files and components enable programmers to browse the large directory structure. During development, this directory may be updated by any system implementor who announces a submission through a standard-format electronic mail message. After the release the contents of this directory become nearly immutable, in that files may be changed or supplanted only for critical bug fixes and enhancements, and only under careful administration. Between releases, interfaces of released components may not change for any reason. Instead, new functions are provided

by new "extras" interfaces (interfaces that contain additions to existing interfaces) that will be merged with the standard interfaces in the next release.

A similar but less stringent methodology applies to the Cedar Applications layer. A separate release directory, known as the CedarChest, contains applications and tools. Although consistency checks are applied periodically, there is no official release process for Applications layer software. Applications programmers are free to release new versions, and frequently do. Interface changes are negotiated among the relevant implementors.

One of the requirements for each system or Applications component is a documentation file. This file, whose name is derived from the component name, is stored in a standard documentation subdirectory and is included in the DF file for the component. Two software catalogs, one for Cedar and another for CedarChest, are automatically constructed from the set of documentation files for components in the release directories. For each component these catalogs provide a summary of its functionality, a set of descriptive keywords, a list of Command Tool commands (if any) that it supplies, and a pointer to its complete documentation file.

Applications that are not part of the official Cedar or CedarChest releases are typically released using similar methods applied to their own file directories.

## 8.6 Methods for Software Browsing

Cedar is large, distributed, and constantly changing. The creation of integrated software can involve the use of items from many interfaces, located at all levels of the environment. Tools for navigating easily and quickly through the system are a necessity. In earlier systems, finding the correct file server and directory for a component was sometimes a challenge in itself. The many conventions and the documentation catalogs described in the release process above help programmers discover and locate relevant information about the system. The tools and methods described below provide additional browsing facilities.

8.6.1 *From Use to Definition.* A few programming environments include comprehensive browsing facilities, which assist with the development and debugging of programs by exploiting the known linkages among definitions, implementations, and uses of program components. The Masterscope function in Interlisp is an excellent example [53]. Cedar lacks a unified browsing capability, but it does include a number of tools that, when used with the Tioga editor, provide considerable assistance in locating things.

The most important of these tools is the Cedar interpreter, which can evaluate expressions, display the fully qualified names of variables, and open Tioga viewers on the source files corresponding to breakpoint locations.

Another invaluable tool is Tioga's use-to-definition function (Section 6.7, *Tioga*), which can open a viewer on the source for either the interface definition or the implementation of a qualified name. These names can be selected from the output of the interpreter or from another source viewer. Since Mesa/Cedar programs often involve chains of definitions that traverse several source files, the ability to proceed within seconds from definition to subdefinition is crucial. This ability to browse without explicit knowledge of the size or organization of the file server name space allows the user to focus on the small numbers of files

that are relevant to the current task, rather than on the enormity of the system as a whole. The symbol tables described in the next section support this use-to-definition facility.

8.6.2 *Symbol Table Management for Cedar Releases.* Tioga's use-to-definition features, the symbolic debugger, and other clients of the Abstract Machine need to perform various mappings between Cedar program values and the source files in which they are declared. The program modules produced by the compiler and binder contain only the location-independent names of the corresponding source files (known as their *short names*), rather than the full path names that could locate them on a release or private directory. For programs under development, one can arrange to have copies of the sources in a local directory where Tioga and the Abstract Machine can look for them. However, additional mechanisms are required to allow users to locate the sources for released components without requiring the entire system directory to reside on each workstation. Without ready access to the sources for the release, the browsing facilities would be much less valuable and the system much less open.

An ad hoc approach to the location of system sources is currently used in Cedar. A set of files known as *version maps* are produced by traversing the tree of DF files corresponding to the released boot file. Version maps are efficient mappings that allow source and object files to be found on their release directories, given their short names or the unique identifiers assigned to their object modules. When there has been a change, updated version maps are automatically copied to the local disk at the beginning of every interactive session.

8.6.3 *Other Browsing Tools.* There are several useful Command Tool operations for browsing through the Cedar system. The *FindR* command examines the version maps for entries matching a file name pattern and simply lists any matches. *OpenR* uses the version maps to open a source viewer, given only a short name. *GetFromRelease* examines the latest compilation error log, creates file attachments in the current working directory for release files that were needed but not found, and suggests additions to the DF file for the package being modified.

The set of browsing tools in Cedar is by no means complete. When all else fails, users resort to a command that performs a regular expression search through a specified set of source files. At a few seconds per file on a Dorado, this method is often effective in finding a starting point for a more structured investigation.

## 8.7 Methods for Developing and Testing Programs

All program development in Cedar, aside from simple experiments using the interpreter, involves compiling and loading new versions of Cedar language source programs. The methods work in the same way whether system components or private applications are being developed, and whether the intent is experimentation or permanent change. However, the methods that are applicable do vary, depending on whether a new component is being added to the system, a component is being replaced by another, or the behavior of a component is simply being modified. As we have seen before, the methods allow more flexibility but are often more elaborate than the methods for similar development in closed operating systems.

8.7.1 *Adding New Components.* If the package or tool to be run has not yet been loaded, and if it does not need to be a part of the Nucleus layer to run correctly, the process of testing it is simple: one simply loads it and starts it (see Section 5.10, *Running Programs*). If its permanent home is in the Nucleus or Life Support layers, the replacement methods described below will eventually be required.

8.7.2 *Replacing Components.* If a component is to be permanently substituted for another, or if it is important that clients of the old component be rebound to the new one, the *replacement* method must be used. Replacing a component requires producing a new instance of the system that contains the new component version instead of the old one.

In some cases it is feasible to replace a program module without restarting the system or destroying the values of its global variables. The new code can be loaded, and clients of the previous instance can be "rebound" to the new one. The old code can then be removed from virtual memory. Module replacement has been demonstrated experimentally in Cedar, but has not been released for general use.

In the absence of dynamic module replacement, the only way to replace a component is to restart the system with the new component in it. The difficulty of doing this depends on the component's level. Applications components can be readily replaced after a rollback to a checkpoint that does not include them. Life Support components can be replaced by performing a full boot using a modified list of boot packages. Finally, replacing a Nucleus component requires the installation of a modified boot file, followed by a full boot. However it is done, when a component is replaced, all clients will execute the new one.

8.7.3 *Modifying Component Behavior.* The techniques described in Section 8.1, *Using Procedure Variables*, are used in many places throughout Cedar to enable client modification of existing components without requiring recompilation or replacement of the components themselves. To use these techniques, the original component implementation must be written with flexible client extension in mind. Otherwise the modification will involve changes to the components themselves. Sometimes these changes can be tested without resorting to component replacement by using a technique called *augmentation.*

Augmentation involves running a new copy of a component without disturbing the previous versions. One can confine the effects of the new component to a selected set of clients by binding them together into a configuration that does not export the component's interfaces. Such a configuration effectively hides the interfaces from the load state. Alternatively, one can make the new component's interface public by substituting its interfaces for the existing ones in the current load state. In the former case, any clients that are run after the augmented configuration is loaded will be bound to the original component; in the latter case, they will get the new one.

It is usually possible to augment a system with several new instances of a component before the exhaustion of some system resource, such as virtual memory or a dedicated part of it, forces a rollback. Most Cedar applications are developed in this way.

## 9. CASE STUDIES

In order to give a feeling for how Cedar is really used for experimental program development, we present two case studies. The first describes the replacement of the Cedar Graphics package by the Imager package during a recent Cedar release. The second demonstrates the value of the layering of components in the development of an integrated voice-annotated electronic mail application.

### 9.1 Case Study: Imager Replacement of Cedar Graphics

The augmentation approach discussed in the previous section has a number of advantages in an interactive system. One such advantage is the possibility of developing and debugging new interactive tools and interfaces using the facilities of their predecessors. Conceptually, earlier open operating systems could support this kind of interactive bootstrapping. In practice, however, difficulties such as naming conflicts, limited memory, or the inability to share user input/output resources have prevented effective application of this idea. An elaborate instance of this approach was successfully applied during the development of Cedar 6.0.

The left half of Figure 6 depicts the structure of the Cedar 5.2 system, emphasizing the user input/output components and the program editing, development, control, and debugging tools that use them. These components have all been described in considerable detail in Section 5, *Nucleus* and in Section 6, *Life Support*, except for the Cedar Graphics package, which was the predecessor to the Cedar 6.0 Imager. Cedar Graphics implemented an equally powerful graphical model, but was directed primarily at managing the Cedar display. Having built the more device-independent Imager, the developers were faced with the problem of debugging corresponding versions of Viewers, Tioga, and the various applications that depend upon them. Because user interaction with the local debugger relies on the services of the Imager, Viewers, and Tioga, debugging new versions of them presented a difficult problem. To examine and control a wayward Imager-based system, the developers could use either the world-swap debugger or the teledebugger, both based on the existing Cedar Graphics package. However, the augmentation approach permitted the developers to use the more convenient local debugger.

The lowest level of the terminal input/output facilities is the Terminal component (Section 5.9), which was carefully designed to support completely independent, possibly conflicting uses of the display, keyboard, and mouse. Each suite of applications relies on the facilities of a virtual terminal. The user can select a virtual terminal to view and manipulate by depressing an unlikely assortment of keys.

Figure 6 as a whole indicates how the Cedar 5.2 system was augmented to include a configuration containing the new Imager. Bound with the new Imager were its applications and a copy of the user input components, Inscript and TIP. Inscript and TIP were set up to obtain input from a new virtual terminal whose display was controlled by the Imager. Because this configuration did not export any interfaces, all packages outside it continued to use the existing Cedar Graphics-based system. Until this configuration was working reliably, the developers were able to use the Cedar Graphics-based program development tools, still associated with the standard virtual terminal, to debug the new components.

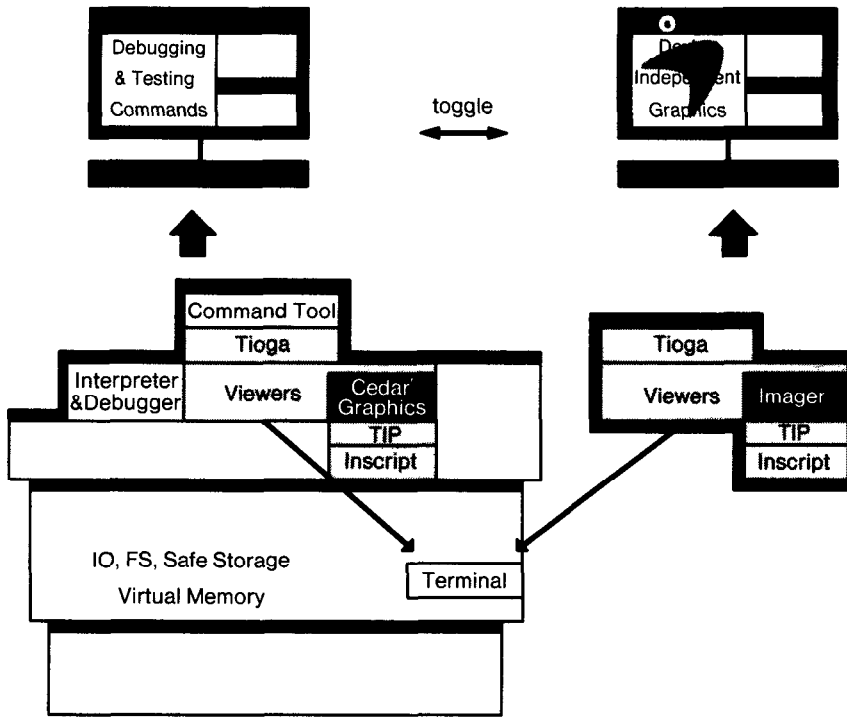


Fig. 6. The replacement of the Cedar Graphics package by the new Imager package required binding the Imager and its clients together to avoid interface mismatches.

In a subsequent configuration, not shown, the role of the Imager and Cedar Graphics were reversed. This produced a version of the now nearly complete Cedar 6.0 system, with a version of the earlier program development tools available (in an alternate virtual terminal) to handle obscure problems with the new facilities.

## 9.2 Case Study: WalnutVoice and Layers of Applications

The layering methodologies apply to Cedar applications as well as to the Cedar system. This layering of applications helps programmers manage the large amount of Cedar software and cope with dependencies on packages undergoing concurrent development. WalnutVoice allows electronic mail messages to be annotated with digitized voice recordings; it is a client of Walnut and is layered on top of Walnut. Walnut supports both a user interface for managing one's personal mail in a database and a client program interface for other applications to integrate with Walnut. Clients may depend on this stable interface despite continued performance improvements in the Walnut implementation. Walnut in turn depends on several application components, including the database software (Cypress), a user interaction package that supports buttons composed of formatted Tioga documents (TiogaButtons), and the typesetting software (TSetter). Walnut itself is layered into several parts, one for the user interface to the mail database, one for managing the database, one for sending messages over the

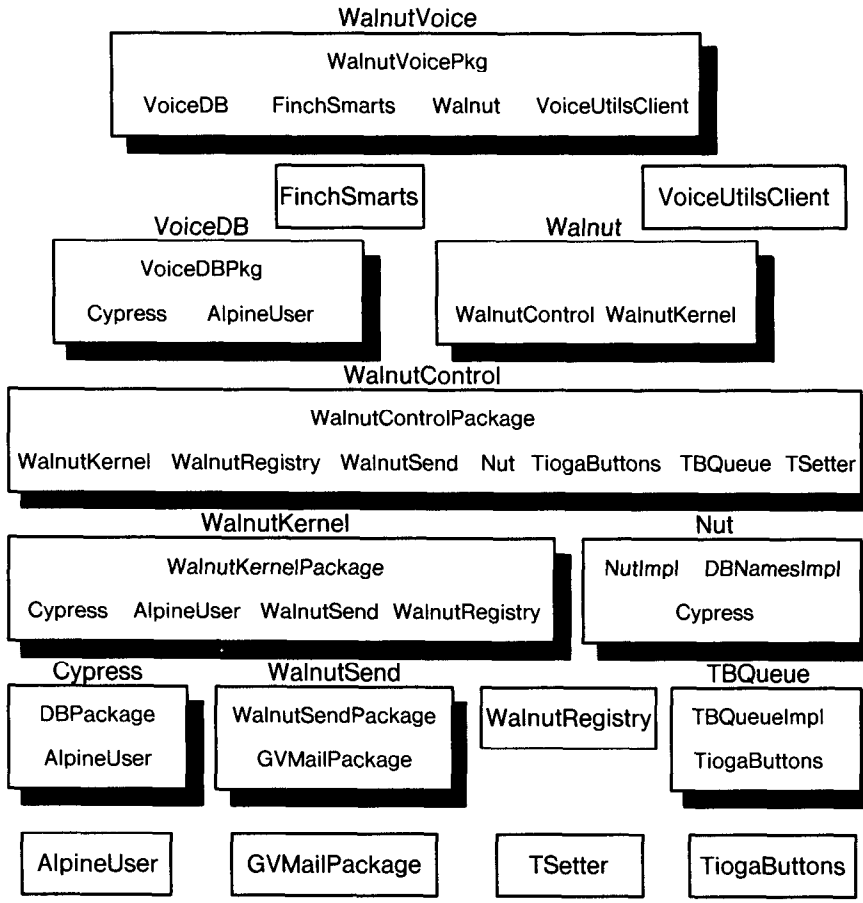


Fig. 7. Application components that make up WalnutVoice, which offers voice annotation of electronic mail messages, may be layered much like Cedar. Each box represents a component whose name appears above the box (unexpanded components, which have no application component dependencies, are shown as smaller boxes). The top line of names within a box lists the major implementation modules (Walnut has none as it combines two other components). The bottom line lists other application components that are imports (other imports from the Life Support and Nucleus layers are not described). Several component and module names appear here that do not appear elsewhere in the paper; they are included here for completeness.

internetwork, and one for registering Walnut event service handlers. Figure 7 shows all of the layers of application components in Cedar upon which WalnutVoice depends. It is important to note that the open operating systems approach permits a component to access lower level components through the permeable layers. For example, WalnutVoice depends directly on the Cypress database package for its directory information as well as indirectly through Walnut for the maintenance of the Walnut message database.

Another way to view the open operating system in this layered set of applications is to concentrate on one component. This examination reflects the working set of interfaces that a programmer might deal with when designing a single

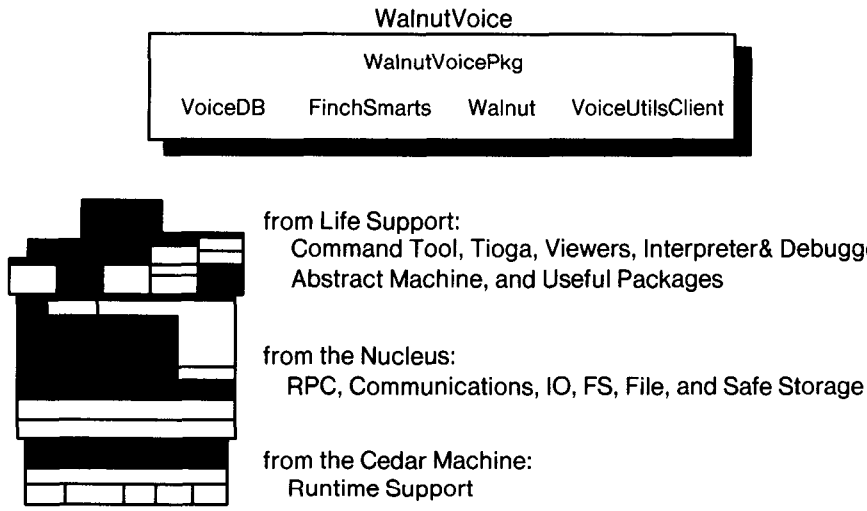


Fig. 8. The WalnutVoice component depends directly on four application components and fifteen Cedar interfaces from the Life Support, Nucleus, and Cedar Machine layers. The fifteen system components are shown as dark areas on the small version of Figure 2 below.

component. WalnutVoice depends directly on four application components and about 15 interfaces from the Life Support, Nucleus, and Cedar Machine layers of Cedar. These dependencies appear in Figure 8. While the implementations of these dependent components may be quite large, the programmer works only with the interface specifications. Convenient browsing facilities permit the programmer to examine relevant interface details quickly.

## 10. COMPARISONS WITH OTHER ENVIRONMENTS

To put Cedar in perspective, we 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 the UNIX system, 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 turn-around for program changes during system development, and the specifics of the user interface. We concentrate instead on structural aspects.

### 10.1 Interlisp-D

Interlisp, a dialect of Lisp, was initially created as an application program running in the Tenex operating system [5]. 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 original 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 Dolphin, Dandelion, and Dorado. It has been enhanced with a bitmapped display and a powerful window management package, based on earlier prototype work using Tenex Interlisp with Altos as terminals. The resulting system is known as Interlisp-D [57]. Interlisp-D should be classed as an open operating system, in the sense that all of the system's components are available to client programs.

All Lisp dialects rely centrally on automatic storage management of their list structures. In fact, it was the clear success of Lisp garbage-collection methods that 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 nonpreemptive process scheduler with no semaphore or monitoring facilities. Errors in process synchronization cannot interfere with proper storage management, but one must exercise care to avoid races and deadlocks.

Interlisp does not have strong typing. There has been a long-running debate about the trade-off between the flexibility of typeless systems, such as Interlisp, and strongly typed systems, which are able to find many errors at compile time and which offer the potential for greater efficiency at execution time. Cedar takes an intermediate position in this debate by using strong typing, but still allowing for delayed type binding and generic references (see Section 2, *The Cedar Language*).

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.

### 10.2 Smalltalk-80

Smalltalk systems, from Smalltalk-72 through the present Smalltalk-80 [20], 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 every aspect of Smalltalk except for a very small kernel is available to Smalltalk programmers.

Smalltalk systems also rely upon automatic storage management, and their allocated objects are more complex than those of Interlisp. Objects are represented as variable-sized records containing embedded object references. The Interlisp and Smalltalk implementations provided a partial existence-proof for the kind of storage management Cedar needed. The overall safety of Smalltalk-80 is 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 one of Cedar's goals—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, in which specific object classes are specified as extensions to the specifications of more general ones. Class inheritance is a structuring approach that is orthogonal to the explicit layering of Cedar components. Inheritance 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 a further organization of operations within a class, encouraging the programmer to collect these operations into subgroups called *categories*. This is also an idea that could be used to advantage in Cedar.

## 10.3 The UNIX System

We have selected the UNIX system as an example of 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 the user processes from each other. The closed approach has disadvantages that led to the development of open operating systems like Cedar, but it also has important advantages.

*Disadvantages of a closed operating system*

—The clear boundary between the application and the system is apparent in application programs, usually appearing explicitly as a system call of some kind. System facilities that are available as system calls can contain useful subcomponents. However, these subcomponents are often not directly available to applications.

—Applications that run as parts of an integrated open operating system often benefit from the ability to share common memory. In particular, the management of the shared display within systems like Cedar are heavily dependent on shared memory. System performance and programming convenience suffer when applications are forced to take a more arm's length approach to information sharing.

Much of the strength of the UNIX system has come from the interoperation of commands. A typical command in the UNIX system is a filter. That is, it takes an input stream, modifies the data, and writes the result on an output stream. Such filters can often be composed to produce the desired program. The commands *interoperate*[4] in that the output of one command can become the input of the next command. In essence, such interoperating programs communicate through their command or user interfaces. The weakness of this approach

[4] Our use of the distinction between interoperation and integration is due to a conversation with Robert Sproull.

is that the information must be expressed as a stream of bytes. Often this is inconvenient, because there may be no simple mapping of the data to a stream of bytes. (For example, a directed graph does not map well into a linear structure.) Cedar instead provides *integration* of packages, in that the intermediate forms of one computation are used within another computation.
—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 of a closed operating system*

—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 storage-management decisions and calling conventions (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 memory and other run-time state associated with a target process. The target process 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 teledebugging or world-swap debugging, both fairly clumsy methods. However, Cedar's nonlocal methods are less clumsy than the methods available for debugging the UNIX kernel, which use a symbolic assembly-level debugger to examine a core dump.

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

## 11. CONCLUSIONS

This paper identifies those systems aspects of the Cedar programming environment that distinguish it from other experimental programming environments. The paper indicates how the novel features combine to achieve a select set of benefits: improved programmer productivity, improved quality of software, integration of system software, and flexible program development methodologies. Some of these benefits were original goals of the Cedar project, and others were identified along the way.

Programmer productivity in Cedar is improved significantly by the integration of tools in a high-quality environment. In addition, a number of powerful tools enhance the programming process. The structured Tioga document editor, Abstract Machine facilities for examining program structures, debugging techniques that operate in the same or remote address spaces, performance measurement tools, and tools for version and release management—all available in an

environment that also contains the programs under development and the everyday office-related necessities of the user—account for this improvement.

High-quality programs result primarily from careful design and implementation. The issue in this paper has been to identify how the machine architecture and the system design can help. The Dorado, with its high-speed processor, large address space, and large physical memory, was designed to remove impediments to experimental development caused by limited resources. Cedar uses its strongly typed language, automatic storage management, careful organization of components into layers, and lightweight processes (among other things) to exploit the power of the machine while reducing or eliminating the most common causes of unreliability.

For an environment to be considered integrated, it must first support multiple activities simultaneously. It must permit the resources of the machine to be shared so that independent activities do not interfere with each other. It must also support the sharing of components, the building of specialized components in terms of more general ones, and the sharing of user interface implementations and methodologies across a wide range of applications. The Cedar attributes that help improve program quality are all important for integration. Again, lightweight processes, the contributions of automatic storage management, and Cedar's layered organization are contributing factors. Procedure registration and the object-style programming enabled by procedure variables are particularly valuable tools for integration.

These benefits are all interrelated. Without some of them, others would be harder to achieve. The program development methodologies outlined in Section 8, *Methodologies*, are particularly important. Their absence would have made it impossible to build an integrated system as large and as rich as Cedar. In addition, these methodologies serve as models for building other experimental systems.

The remainder of this section presents several reflections on the Cedar environment.

### 11.1 Insights

We can now discuss the ramifications of the novel aspects of the Cedar environment, outlined in Section 1.3, *Novel Aspects of Cedar*. The decision to emphasize these particular aspects resulted from insights gained during the design and development of Cedar. These discussions have been deferred until most of the details and context were presented.

11.1.1 *Automatic Storage Management.* The original goal of the Cedar safe language features has been met. It is very rare for Cedar programmers to have to deal with memory smashes. In addition, automatic storage management has provided other benefits.

In systems without automatic storage management, one must deal with the *ownership* of objects, especially parameters to procedures. For example, a routine that prints text strings might be supplied either with a constant string, whose storage should not be released because 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. These problems are magnified when trying to design consistent rules across a large system.

In Cedar we can share objects without concern for ownership of the object and without complicating the interface. Procedures that create objects may simply return an appropriate value, be it a primitive data type or a composite record reference. This clarifies the interface specification as well as the client code, resulting in faster design of interfaces and implementations. The finalization of objects when they are about to be reclaimed permits an object's implementation to control the cleanup of associated data structures, even if the object had been shared extensively.

It is not immediately obvious, but automatic storage management 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 automatic storage management, concern over the lifetime of allocated objects has led to restrictions on the use of procedure variables in system calls [40]. In closed operating systems, difficulties in establishing the proper memory environment generally prohibit the use of either registration or call-back procedures.

The performance of the Cedar incremental garbage collector is sufficiently good that it is rarely noticed by Cedar users. More than 90 percent of all garbage collections occur in the background.

11.1.2 *Procedure Variables and Objects.* Cedar's methods for using procedure variables make experimentation easier by allowing the programmer to extend strongly typed, statically compiled, and statically bound interfaces. Programmers can add functionality without changing low-level parts of the system and without understanding implementation internals. In fact, it is often impossible to determine the difference between client-supplied objects and predefined ones. For example, all Tioga commands, viewer classes, and command names are registered as procedure variables. There is only a minimal performance penalty for this kind of extensibility. However, one must amortize the initial overhead of designing or understanding a registration mechanism over later clients and experiments.

The generic reference type, REF ANY, permits objects to contain and procedures to handle uninterpreted data references. Cedar's use of procedure variables differs from Mesa's because an object or procedure declaration may specify a generic reference without the necessity to define in advance all the types of possible references.

Object class definitions, which characterize the common behavior of objects, are useful structuring concepts across a wide range of applications. Sharing objects makes integration easier because applications can use the same data structures and gain leverage from the same user interface buttons and menus.

11.1.3 *Interfaces, Structural Conventions, and Other Chunking Mechanisms.* Integrated systems are big, and Cedar is enormous. Fortunately, there are several mechanisms in Cedar that help present the system in manageable chunks that people find easier to remember. The four layers organize components into the Cedar Machine, Nucleus, Life Support, and Applications. Each component

provides a small set of interfaces for each abstraction in the component. Finally, implementations of the interfaces accomplish the functionality. For example, programmers interact with the Tioga text editor only through a few client interfaces that spawn over a hundred internal interfaces and implementations.

The structural layers in Cedar lead to a stable understanding of the environment, in spite of constant change. Clients are unaffected by changes in implementations for performance or bug fixes. Layering of applications also means that added functionality is easily inherited. For example, Walnut manipulates Tioga documents as the content of electronic mail messages, and when Tioga was enhanced to display pictures, then electronic mail messages could contain illustrations without any change to Walnut.

Cedar supports "programming in the small" through the interpreter, module interfaces, and implementations, "programming in the medium" through configurations describing packages and components, and "programming in the large" through version management tools, version maps, and file servers with replicated directories. The two case studies in Section 9, *Case Studies*, demonstrated that it is practical to throw away large parts of Cedar and reprogram them with new parts, and that we use the layering methodologies even when building applications.

11.1.4. *The Abstract Machine and its Applications.* The Abstract Machine enables the robust local debugging techniques, backed up by remote debugging, that are fundamental to the productivity and quality of the experimental programs developed in Cedar. Through the Abstract Machine, Cedar supports interpreting the Cedar language, source-level debugging in context, and browsing Cedar programs through use-to-definition and version maps. Interestingly, Cedar does not have a privileged system program called the debugger. Instead, the Tioga editor, the expression interpreter, typescript viewers, and a collection of specialized tools that draw upon the facilities of the Abstract Machine package provide the basic tools for debugging.

When problems arise that cannot be handled by the existing tools, new ones are built. A good example is Celtics, a "fast breaks" package, which counts selected statement executions without disrupting running processes as full-fledged breakpoints would. This tool, created as a nonprivileged client of the Abstract Machine, enables the debugging of time-critical components.

Another productivity issue is automating the construction of programs. We get along without a syntax-directed editor, yet manage to achieve many of its features: correct syntax through templates, consistent typography and style through templates and formatting heuristics, and convenient access to procedure arguments and record fields through abbreviation expansions. Many of these facilities are built on top of the Tioga editor, the Abstract Machine, version maps, and the DF package.

11.1.5 *The Imager and its Applications, Especially Tioga.* Graphics in Cedar is ubiquitous because the Imager handles all display output for the window package and text editor, as well as graphical illustrator programs. Applications can now reliably capture any black-and-white or color image in a device-independent representation, and either display it on the terminal or print it on any of a range

of experimental Interpress printers. Although the imaging model of the Imager is stressed by some interactive applications, notably due to the lack of transparent colors, the overall impact of an efficient, functional, and ubiquitous graphics package has enhanced our ability to experiment with graphical systems and data.

The impact of sharing the Tioga user interface throughout the Cedar system has had significant productivity and integration effects. Users can apply the editing operations to almost any text on the display because applications can easily and efficiently share access to the Tioga editor. Knowledgeable users customize input action events to accelerate common actions or to satisfy personal preferences.

11.1.6 *Facilities for Managing Large Sizes in Cedar.* Achieving improved program quality and complete integration of components requires a commitment to handling large sizes gracefully. The Cedar virtual memory is large. Processes are lightweight and many can run concurrently. The file system permits very large files and directories. Furthermore, data structures within an integrated system may become large. The Cedar system contains an interface design methodology using objects that permits several alternative implementations of the same interface. Thus, for example, a small Tioga document may be composed of in-memory ROPE fragments, while a large document might be composed of file-based RopeFile ROPE fragments. Once created, Tioga does not distinguish between these implementations, but rather manipulates them identically through the single Rope interface. The alternative implementations of a common abstraction permit Cedar to evolve gracefully to accommodate very large problems.

The version and release management methods have made it possible to develop and maintain a system as large and diverse as Cedar. Through DF files and the DF Package, it is possible to maintain control over the organization of the overall system into components, and over the transport of consistent versions of their files to and from the local working directories where the components are developed or used. The motivation for DF files was to regain some of the file management capabilities that were lost in the evolution from time-sharing systems to personal workstations in a distributed computing environment. In fact, the DF Package has provided us with a number of automatic and semi-automatic ways to maintain and verify consistent versions that go beyond what can be achieved through conventional file directory facilities.

## 11.2 Shortfalls

Cedar has its share of inadequacies. Some have been addressed and repaired, such as redesigning the virtual memory and file systems to take better advantage of the machine's ample resources. The remainder fall into three categories: things that need fixing, things that are incomplete, and things that would need to be added before we could claim that Cedar is a fully satisfactory environment.

The Cedar language has inherited from its Mesa ancestors an orientation towards a 16-bit word size and even a 16-bit address space for critical run-time objects such as procedure activation records and global frames. These limitations artificially reduce the number of modules that can be loaded and the number of

processes that can operate concurrently. Cedar has inadequate performance on Dandelion hardware because its memory management facilities were tuned for the larger and faster Dorado.

The Cedar directory package, FS, which manages file names in a way that extends the file system to include the entire Xerox Research Internetwork, does not attempt to make the existence of the network and the local file cache transparent to the programmer or user. Manual steps are still required to move files from the local disk to their permanent storage locations.

Cedar's network communications components have not yet reached the level of maturity of most of the other low-level Cedar packages, nor do they support all of the communication protocols that are in use within Xerox. A much cleaner implementation is nearing completion.

The Abstract Machine interface provides complete access to Cedar types and values at run-time in a consistent fashion. Unfortunately, the interface is complicated and clumsy to use, despite many attempts to simplify it. Further research is required to allow easier program access to this information.

Cedar viewers provide considerable assistance with the control and presentation of information, but they are showing their age. The state of the art of display management facilities has advanced considerably since the Viewers package was designed. There is an effort underway to produce a replacement that will support more ambitious viewing paradigms while improving performance and programming flexibility.

Many of the applications being developed in Cedar are information management applications. The databases and file systems that are needed to support these applications are not yet adequate to the task, although there is active research in this area.

The Cedar language does not support abstract data type concepts as well as it should. Syntactic and semantic support for opaque types, object-style programming, and polymorphic data types are incomplete or absent from the current language.

The system also lacks a complete, efficient, interactive interpreter for the Cedar language. The original intent was to modify the compiler for interactive use, and we still believe that is the right approach. There is also a growing consensus that command interpreters such as the one in the Command Tool should be more fully integrated with the Cedar language interpreter, and that there should be support for specialized user-defined languages.

Cedar runs on proprietary hardware, is expressed in a proprietary language, provides its own proprietary operating system and file format, and communicates using proprietary protocols. While there are good reasons for each of these choices, many of which have been argued in this paper, insufficient attention has been paid to developing methods in Cedar for interacting with other environments, languages, and systems. Several remedies to these difficulties are being pursued.

### 11.3 Summary

The Cedar programming environment is heavily used within the Xerox research community to build experimental software systems. Cedar programmers, both as systems researchers and as beneficiaries of its facilities, are productive in a robust

and integrated environment. The current version of Cedar (Cedar 6.0) occupies more than 17 million bytes of disk storage and contains over 1,500 source files, more than 400,000 lines of source code, approximately 150 DF files, and over 100 configurations. Very rough productivity measurements of the Cedar Life Support layer indicate that over 50 work-years, about 6,000 lines of code per year were produced. Similar measurements for the CedarChest applications software indicate 10,000 lines of code per year over 50 work-years, and for the on-going VLSI design tools project indicate 11,000 lines of code per year over 18 work-years. Some individuals have sustained productive output rates about twice as high. Although Cedar continues to grow, the management tools appear to be keeping pace.

We have examined the Cedar system from its goals, benefits, methodologies, and the components that make Cedar work. This structural overview of Cedar has revealed how it is used for building experimental systems, how it improves programmer productivity, and how it improves the quality of programs. Even with its identified shortfalls, Cedar has reached a level of maturity where it should be valuable in its present form for some time.

## APPENDIX A. GLOSSARY OF CEDAR TERMINOLOGY

| | |
|---|---|
| Abstract Machine | program debugging and analysis facilities (Sect. 6.4) |
| Alto | a small personal computer designed at PARC in 1973 (Sects. 1.1, 3.1) |
| Applications layer | fourth layer of Cedar: packages and tools (Sect. 7) |
| ATOM | uniquely identified objects with properties (Sect. 2.7) |
| attached files | symbolic links to remote files (Sect. 5.6) |
| automatic storage management | automatic storage deallocation through garbage collection, supported by the Cedar safe language subset (Sect. 5.4) |
| BCPL | a typeless system programming language (Sects. 1.1, 3.1) |
| binder | a linkage editor (Sect. 6.8) |
| boot file | a binary form of the Nucleus (Sect. 5.10) |
| bringover | copying DF file and attaching local file names to remote files (Sect. 6.11) |
| call-back procedure | a procedure passed as an argument (Sects. 3.3, 8.1) |
| Cedar Machine | first layer of Cedar: hardware, microcode, and primitives (Sect. 4) |
| checkpoint | a bootable snapshot of a running system (Sects. 5.10, 8.4) |
| client | a program (rather than a person) that uses another program or system (Sect. 2.1) |
| closed operating system | a system with hardware memory protection for separate address spaces (Sects. 3.1, 10.3) |

| | |
|---|---|
| collectible storage | that part of memory where objects may be allocated and garbage collected (Sects. 2.5, 5.4, 7.1) |
| configuration | control information for the binder: result of a binding (Sects. 2.1, 9.2) |
| conservative scan | an optimizaton used in garbage collection (Sect. 5.4) |
| Dandelion | the Xerox 1108 workstation (Sects. 1.1, 4.1, 11.2) |
| dangling reference | an invalid pointer to an object after the object has been deallocated (Sect. 2.5) |
| dependency | package A depends on package B if A uses any interfaces from B (Sect. 2.1) |
| DF Package | software to manipulate packages managed by DF files (Sect. 6.11) |
| DF file | a file that fully describes the files that make up a package (Sect. 6.11) |
| delayed type binding | manipulation of typeless objects at run-time (Sect. 2.6) |
| Dorado | the Xerox 1132 workstation (Sects. 1.1, 4.1) |
| export | implementors of a procedure for an interface are said to export the procedure (Sect. 2.1) |
| finalization | actions that occur when an object is no longer accessible to clients (Sect. 5.4) |
| FS | the Cedar file system (Sect. 5.6) |
| full boot | a boot of Cedar from its components (Sect. 5.10) |
| garbage collection | freeing of collectible objects that are no longer needed (Sect. 5.4) |
| generic reference | *see* REF ANY |
| germ | a small bootstrap program for initializing Cedar (Sect. 5.10) |
| global frame | run-time data associated with an implementation module (Sects. 2.1, 6.4) |
| Imager | device-independent graphics package (Sects. 6.5, 9.1) |
| immutable value | a value that cannot be changed after it has been created (Sects. 2, 2.7, 8.5) |
| implementation module | a program module that contains data declarations and executable statements (Sect. 2.1) |
| import | clients of an interface are said to import the interface (Sect. 2.1) |
| incremental garbage collector | a garbage collector that does its job concurrently with other system activities (Sect. 5.4) |
| interface module | a program module that describes the public part of a data abstraction (Sect. 2.1) |
| Interlisp | a dialect of Lisp with a large integrated library of facilities (Sects. 3.4, 10.1) |

| | |
|---|---|
| IO | the input/output package: implements STREAM (Sect. 5.7) |
| Life Support layer | third layer of Cedar: basic program development facilities (Sect. 6) |
| lightweight process | a process with a very fast context switch and no responsibility for memory management (Sect. 2.2) |
| LIST | variable-length linked list (Sect. 2.7) |
| load state | symbol table for exported items in a running Cedar system (Sect. 5.10) |
| local debugging | debugging in the same address space as the program being debugged (Sects. 5.10, 8.4) |
| Mesa | a Pascal-like, strongly typed, system programming language (Sects. 1.1, 2, 3.3) |
| Mesa/Cedar | term used throughout this paper to refer to features common to the Mesa and Cedar languages |
| monitor | a language method to provide mutually exclusive access to shared data (Sect. 2.2) |
| NARROW | a type validation function for generic references (Sect. 2.6) |
| Nucleus | second layer of Cedar: operating system kernel (Sect. 5) |
| object-style programming | a philosophy of how to use abstract data types (Sect. 2.4) |
| open operating system | collection of program modules for an operating system (Sect. 3.1) |
| Pilot | an operating system based on Mesa (Sects. 1.1, 5.1) |
| polymorphic language | allows values of type TYPE to be passed as parameters and stored in variables (Sect. 2.6) |
| procedural object | a record that includes data and procedure variables (Sects. 2.4, 8.1) |
| procedure variable | procedure descriptor; may be passed as a parameter and stored in a variable (Sects. 2.3, 8.1) |
| programs as data | dynamic construction and execution of programs (Sect. 6.4) |
| property list | a list of key-value pairs, where a key is usually an ATOM (Sect. 6.9) |
| registered procedure | a call-back procedure that is retained for later invocation (Sect. 3.3) |
| reference (REF) | a typed, reference-counted pointer to a collectible object (Sect. 2.5) |
| REF ANY | an untyped reference to any collectible object (Sects. 2.6, 6.1, 6.4, 8.1) |
| reference count | the count of REF's pointing to an object (Sect. 2.6) |

| | |
|---|---|
| remote file | a file stored on a file server (Sect. 5.6) |
| retained execution frame | activation record retained beyond the lifetime of the corresponding procedure invocation: sometimes known as a closure (Sects. 2.8, 3.3, 8.1) |
| rollback | restart of a Cedar world saved in a checkpoint (Sect. 5.10) |
| ROPE | immutable garbage-collected sequence of characters (Sect. 2.7) |
| RPC | remote procedure call (Sect. 5.8) |
| safe subset | Cedar language subset that always maintains the storage invariants required for automatic storage management (Sect. 2.5) |
| server | a computer dedicated to performing service functions (Sects. 4.1, 5.4, 5.8) |
| service | a program or system that responds to clients (Sect. 5.8) |
| Smalltalk | an integrated object-oriented programming system (Sects. 3.4, 10.2) |
| STREAM | a data abstraction describing a sequence of bytes (Sects. 5.7, 8.1) |
| storage leak | failure to deallocate unreferenced allocated objects (Sects. 2.5, 3.4, 5.4) |
| storeback | copying updated files described by a DF file onto a file server (Sect. 6.11) |
| teledebugging | debugging one world from another using two machines and the Ethernet (Sect. 5.10) |
| Tioga | a galley editor for formatted documents (Sect. 6.7) |
| TIP tables | specification for interpreting terminal input (Sect. 6.3) |
| type discrimination | discovering the type of a REF ANY (Sect. 2.6) |
| user | a person (rather than a program) who uses some program or system |
| user profile | a collection of parameters set by the user (Sect. 6.2) |
| viewer | a rectangular region of the display; window (Sect. 6.6) |
| VM | virtual memory package (Sect. 5.3) |
| Walnut | an electronic mail database system (Sect. 7.3) |
| world | an instance of a system, including its virtual memory and processes (Sect. 6.4) |
| world-swap debugging | debugging one world from another using the same machine (Sect. 5.10) |
| XDE | the programming environment for Mesa (Sects. 1.1, 6.11) |

## APPENDIX B. CEDAR RELEASE HISTORY

| *Release* | *Date* | *Major features* | *Number of components* |
|---|---|---|---|
| Cedar1.? | Aug. 1981 | First release of Cedar to clients | – |
| Cedar2.0 | Oct. 1981 | First automatic release of Cedar | 22 |
| Cedar2.1 | Nov. 1981 | | 24 |
| Cedar2.2 | Dec. 1981 | Safe language introduced | 26 |
| Cedar2.3 | Jan. 1982 | | 32 |
| Cedar2.4 | Feb. 1982 | Viewers & Tioga introduced | 40 |
| Cedar2.5 | Mar. 1982 | User Executive (earlier version of Command Tool), DF tools | 48 |
| Cedar2.6 | Apr. 1982 | *maintenance release* | 50 |
| Cedar3.0 | May 1982 | no Tajo dependencies, Database, Cedar interim file system, Press printing | 62 |
| Cedar3.1 | May 1982 | maintenance release | 63 |
| Cedar3.2 | Jul. 1982 | Safe interfaces for Nucleus, Color display support, Remote procedure call, Viewers and Tioga work well | 79 |
| Cedar3.3 | Aug. 1982 | maintenance release | 78 |
| Cedar3.4 | Oct. 1982 | Local debugger, Walnut electronic mail database, initial Dandelion support, Cedar kernel language | 87 |
| Cedar3.5 | Dec. 1982 | Major maintenance release, Abstract Machine introduced | 95 |
| Cedar4.0 | Mar. 1983 | No CoPilot dependencies, Abstract Machine, world-swap debugging, performance improvements, Alpine transaction file server introduced | – |
| Cedar4.1 | May 1983 | Command Tool, Interpreter Tool, maintenance release | – |
| Cedar4.2 | Jun. 1983 | RopeFile, Squirrel database browsing tool, *maintenance release* | 146 |
| Cedar4.3 | – | maintenance release | 146 |
| Cedar4.4 | – | maintenance release | 146 |
| Cedar5.0 | Dec. 1983 | Cedar off Pilot base, FS, VM, Safe Storage, IO, four major layers: Machine, Nucleus, Life Support, Applications | – |
| Cedar5.1 | Mar. 1984 | Dandelion support, specific release directories | 153 |
| Cedar5.2 | Jun. 1984 | performance improved, >8 megabyte memory support, many new applications | – |
| Cedar6.0 | Jun. 1985 | Interface housecleaning, Imager, Interpress, logical file servers (246 application components in CedarChest) | 104 |

### 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, Peter Deutsch, Bob Hagmann, Butler Lampson, Roy Levin, Scott McGregor, Jim Morris, Hal Murray, 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. More

than 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 also like to thank those who helped us with the preparation of this paper: Jim Donahue for permitting us to incorporate ideas from his integration paper [16]; Peter Kessler and Bertrand Serlet for their thorough readings; Ed Satterthwaite for comments on the language section; Luis Felipe Cabrera, Pavel Curtis, Carl Hauser, Roy Levin, Michael Plass, Paul Rovner, and Larry Stewart for reading early versions; Jack Kent and John Beatty for reading the paper with the view of a neophyte Cedar user. The referees' comments and questions also helped to clarify our presentation.

## REFERENCES

1. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 3 (1972), 173–189.
2. BEACH, R. Experience with the Cedar programming environment for computer graphics research. *Gr. Interface 84*.
3. BIRRELL, A., LEVIN, R., NEEDHAM, R., AND SCHROEDER, M. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (Apr. 1982).
4. BIRRELL, A., AND NELSON, B. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984).
5. BOBROW, D., BURCHFIEL, J., MURPHY, D., AND TOMLINSON, R. TENEX: A paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
6. BOGGS, D., SHOCH, J., TAFT, E., AND METCALFE, R. Pup: An internetwork architecture. *IEEE Trans. Commun. 28*, 4 (Apr. 1980), 612–624.
7. BOURNE, S. The UNIX shell. *Bell Syst. Tech. J. 57*, 6, Pt. 2 (Jul.–Aug. 1978), 1971–1990.
8. BRINCH HANSEN, P. *Operating Systems Principles*. Prentice-Hall, Englewood Cliffs, N.J., Jul. 1973.
9. BROWN, M., KOLLING, K., AND TAFT, E. The Alpine file system. Xerox PARC Rep. CSL-84-4, 1984.
10. CARGILL, T. Debugging C programs with the Blit. *AT&T Bell Lab. Tech. J. 63*, 8, Pt. 2 (1984), 1633–1648.
11. CATTELL, R. G. G. Design and implementation of a relationship-entity-datum data model. Xerox PARC Rep. CSL-83-4, 1983.
12. CLARK, D. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operation Systems Principles* (Dec. 1985), 171–180.
13. DELISLE, N., MENICOSY, D., AND SCHWARTZ, M. Viewing a programming environment as a single tool. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Apr. 1984), 49–56.
14. DEUTSCH, P., AND BOBROW, D. An efficient, incremental, automatic garbage collector. *Commun. ACM 19*, 7 (Jul. 1976).
15. DEUTSCH, P., AND TAFT, E. Requirements for an experimental programming environment. Xerox PARC Rep. CSL-80-10, 1980.
16. DONAHUE, J. Integration mechanisms in Cedar. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments* (Seattle, Wash., Jun. 1985). *SIGPLAN Not. 20*, 7 (Jul. 1985).
17. DONAHUE, J., AND WIDOM. J. Whiteboards: A graphical database tool. Xerox PARC Rep. CSL-85-4, 1985.
18. FELDMAN, S. Make—a program for maintaining computer programs. In *UNIX Programmer's Manual, Supplementary Documents, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*. Computer Science Div., Univ. of California, Berkeley, 1984.
19. GESCHKE, C., MORRIS, J., AND SATTERTHWAITE, E. Early experience with Mesa. *Commun. ACM 20*, 8 (Aug. 1977).

20. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation*, McGraw-Hill, New York, 1983.
21. GUIBAS, L., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, (Ann Arbor. Mich., Oct. 1978).
22. HAGMANN, R. Process server: Sharing processing power in a workstation environment. In *Proceedings of the 6th International Conference on Distributed Computing Systems* (Boston, May 1986).
23. IEEE. A proposed standard for binary floating-point arithmetic. *Computer 14*, 3 (Mar. 1981), 51–62.
24. HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
25. JOHNSSON, R., AND WICK, J. An overview of the Mesa processor architecture. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operation Systems* (Apr. 1982). *SIGPLAN Not. 17*, 4 (Mar. 1982).
26. LAMPSON, B., AND PIER, K.: LAMPSON, B., MCDANIEL, G., AND ORNSTEIN, S.: CLARK, D., LAMPSON, B., AND PIER, K. The Dorado: A high performance personal computer, three papers. Xerox PARC Rep. CSL-81-1, 1981.
27. LAMPSON, B., AND REDELL, D. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb. 1980), 105–117.
28. LAMPSON, B., AND SCHMIDT, E. Organizing software in a distributed environment. In *Proceedings of the SIGPLAN 83 Symposium on Programming Language Issues in Software Systems* (San Francisco, Jun. 1983).
29. LAMPSON, B., AND SPROULL, R. An open system for a single-user machine. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Dec. 1979), 98–105.
30. LAUER, H., AND NEEDHAM, R. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems* (Rocquencourt, France, Oct. 1978), IRIA. Reprinted in *Oper. Syst. Rev. 13*, 2 (Apr. 1979), 3–19.
31. LAUER, H., AND SATTERTHWAITE, E. The impact of Mesa on system design. In *Proceedings of the 4th International Conference on Software Engineering* (Munich, Sept. 1979).
32. MCCREIGHT, E. The Dragon computer system: An early overview. In *Proceedings of the NATO Advanced Study Institute on Microarchitecture of VLSI Computers* (Urbino, Italy, Jul. 1984).
33. MCDANIEL, G. The Mesa Spy: An interactive tool for performance debugging. In *Proceedings of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Aug. 1982).
34. METCALFE, R., AND BOGGS, D.: CRANE, R., AND TAFT, E.: SHOCH, J., AND HUPP. J. The Ethernet local network: Three reports. Xerox PARC Rep. CSL-80-2, 1980.
35. MITCHELL, J. Mesa language manual. Xerox PARC Rep. CSL-79-3, 1979.
36. NBS. *Data Encryption Standard.* FIPS publ. 46, National Bureau of Standards, U.S. Dept. of Commerce, Washington, D.C., 1977.
37. NEWMAN, W., AND SPROULL, R. *Principles of Interactive Computer Graphics.* 2nd Ed., McGraw-Hill, New York, 1979.
38. OWICKI, S. Making the world safe for garbage collection. *POPL 8* (Jan. 1981).
39. PERLIS, A. Another view of software. In *Proceedings of the 8th International Symposium on Software Engineering* (Imperial College, London, Aug. 1985).
40. REDELL, D., DALAL, Y., HORSLEY, T., LAUER, H., LYNCH, W., MCJONES, P., MURRAY, H., AND PURCELL, S. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb. 1980).
41. RITCHIE, D., AND THOMPSON, K. The UNIX time-sharing system. *Bell Syst. Tech. J. 57*, 6, Pt. 2 (Jul.–Aug. 1978), 1905–1930.
42. ROVNER, P. On adding garbage collection and runtime types to a strongly-typed, statically-checked concurrent language. Xerox PARC Rep. CSL-84-7, 1985.
43. SCHMIDT, E. *Controlling large software development in a distributed environment.* Ph.D. thesis. Univ. of California, Berkeley, 1982; also available as Xerox PARC Report CSL-82-7, 1982.
44. SCHROEDER, M., GIFFORD, D., AND NEEDHAM, R. A caching file system for a programmer's workstation. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Dec. 1985), 25–34.

45. SERLET, B. Object-oriented programming in Cedar. In *Proceedings of Journees Langages Orientes Objet* (Paris, Jan. 1986). Also appears in *Actes des Journees Langages Orientes Objet*. Bigre Globule, 64–68.
46. STEWART, L., SWINEHART, D., AND ORNSTEIN, S. Adding voice to an office computer network. In *Proceedings of the GlobeCom 83, IEEE Communications Society Conference* (Nov. 1983).
47. SWEET, R. The Mesa programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments* (Jun. 1985). *SIGPLAN Not. 20*, 7 (Jul. 1985).
48. SWEET, R., AND SANDMAN, J., JR. Empirical analysis of the Mesa instruction set. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*. (Apr. 1982). *SIGPLAN Not. 17*, 4 (Mar. 1982).
49. SWINEHART, D., ZELLWEGER, P., AND HAGMANN, R. The structure of Cedar. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments* (Jun. 1985). *SIGPLAN Not. 20*, 7 (Jul. 1985).
50. TEITELBAUM, T., AND REPS., T. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept. 1981), 563–573.
51. TEITELMAN, W. A tour through Cedar. *IEEE Softw.* (Apr. 1984).
52. TEITELMAN, W. The Cedar programming environment: A midterm report and examination. Xerox PARC Rep. CSL-83-11, 1984.
53. TEITELMAN, W., AND MASINTER, L. The Interlisp programming environment. *Computer 14*, 4 (Apr. 1981), 25–34.
54. THACKER, C., MCCREIGHT, E., LAMPSON, B., SPROULL, R., AND BOGGS, D. Alto: A personal computer. Xerox PARC Rep. CSL-79-11, 1979.
55. WALLACE, D. Tajo functional specification, version 6.0. Xerox internal document, Oct. 1980.
56. WARNOCK, J., AND WYATT, D. A device-independent graphics imaging model for use with raster devices. *Comput. Gr. 16*, 3 (Jul. 1982), 313–319.
57. Xerox Corp. *Interlisp Reference Manual*. Oct. 1983.
58. Xerox Corp. *Interpress Electronic Printing Standard, Version 2.1*. Xerox System Integration Standard XSIS 048404, Apr. 1984.
59. ZIMMERMANN, H. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. 28*, 4 (Apr. 1980), 425–432.

Received September 1985; revised March and May 1986; accepted May 1986