The Impact of Mesa on System Design

Hugh C. Lauer
Edwin H. Satterthwaite
Xerox Corporation
Palo Alto, California

Abstract

The Mesa programming language supports program modularity in ways that permit subsystems to be developed separately but to be bound together with complete type safety. Separate and explicit interface definitions provide an effective means of communication, both between programs and between programmers. A configuration language describes the organization of a system and controls the scopes of interfaces. These facilities have had a profound impact on the way we design systems and organize development projects. This paper reports our recent experience with Mesa, particularly its use in the development of an operating system. It illustrates techniques for designing interfaces, for using the interface language as a specification language, and for organizing a system to achieve the practical benefits of program modularity without sacrificing strict type-checking.

Mesa is a programming language designed for system implementation. It is used within the Xerox Corporation both by research laboratories as a vehicle for experiments and by development organizations for 'production' programming. Some of our initial experience with Mesa was reported previously [Geschke *et al*, 1977]. Since that time, the language has evolved in several directions and has acquired a larger and more diverse community of users. That community has accumulated a substantial amount of experience in using Mesa to design and implement large systems, a number of which are now operational. It has become increasingly clear that the value of Mesa extends far beyond its enforcement of type-safety within individual programs. It has profoundly affected the ways we think about system design, organize development projects, and communicate our ideas about the systems we build.

This paper reports some of our recent experience with Mesa. It is based primarily upon the development of one particular system—what we refer to as the Pilot operating system—for a small, personal computer. We also draw upon the lessons learned from other systems. These represent a non-trivial amount of programming; a survey of just the authors' immediate colleagues at the end of 1978 uncovered several hundred thousand lines of stable, operational Mesa code. Pilot itself is a 'second generation' client of Mesa. It is the first major system to take advantage of explicit interface and configuration descriptions (discussed below) in its original design. In addition, its designers were able to make careful assessments of earlier systems to discover both the benefits and pitfalls of using Mesa. As a result, we were able to benefit from, as well as add to, the accumulated 'institutional learning' about the practical problems of developing large systems in Mesa.

The purpose of this paper is to communicate those lessons, which deserve more emphasis and discussion than they have received to date in the literature. We concentrate upon the impact and adequacy of the Mesa programming language and its influence upon system design; a companion paper [Lynch and Horsley, 1979] focuses upon organizational and management issues. This paper contains three main sections. First, the facilities provided by Mesa for supporting the development

and organization of modular programs are discussed. In the second section, we describe the role played by the Mesa interface and configuration languages in system design, particularly from the perspective of Pilot. The final section is a qualitative assessment of the adequacy of Mesa as a system implementation language.

Context

Mesa is both a language and a system. The Mesa language features strict type-checking much like that of PASCAL [Wirth, 1971] or EUCLID [Lampson *et al*, 1977], with similar advantages and disadvantages. In particular, the type-checking moves a substantial amount of debugging from run-time to compile-time. Much has been written on this subject; our views and design decisions have changed little since our earlier report [Geschke *et al*, 1977]. The type system of Mesa pervades all other aspects of the language and system. The latter consists of a compiler, a binder, a source language debugger, and a number of other tools and utilities. The system has been implemented on machines that can be microprogrammed at the register transfer level; thus we have also been able to design and implement a machine architecture specifically tailored to Mesa.

Mesa is an evolving language and system. The published version of the language manual [Mitchell *et al*, 1978] differs from descriptions presented in earlier papers in a number of ways, most notably in the explicit definition and control of interfaces. More recently, processes, monitors, and synchronization have been added; these resemble the 'procedure-oriented model' of [Lauer and Needham, 1978]. The compiler, instruction set, and microprogrammed interpreters have all evolved to accommodate these changes.

The Pilot operating system upon which this report is based is programmed entirely in Mesa, as are all of its clients. In addition to providing the usual set of operating system facilities, Pilot implements all of the run-time machinery needed to support the execution of Mesa programs, including itself. The clients are assumed to be friendly and cooperating, not hostile or malicious. Since no debugging takes place on machines that are simultaneously supporting other users, no attempt has been made to provide a strong protection mechanism; instead the goal has been to minimize the likelihood of uncontrolled damage due to residual errors. Pilot was designed and implemented by a core group of six people, with important contributions by members of other groups in specialized areas. By late 1978, the total system consisted of approximately twenty five thousand lines of Mesa code.

Modularity in Mesa

Systems built in Mesa are collections of modules. The general structure of a Mesa *module* is described in [Geschke *et al*, 1977]. Viewed as a piece of source text, a module resembles an ALGOL procedure or SIMULA class declaration. Although the Mesa language enforces no particular style of module usage, a *de facto* standard has evolved. An instance of a module typically manages a collection of *objects*. Each object contains information characterizing its own state. The module instance provides a set of procedures to create, operate upon, and destroy the objects; it contains any data shared by the entire collection (e.g., a table of allocated resources) and perhaps some initialization code also.

Modules communicate with each other via *interfaces*. A module may *import* an interface, in which case it may use facilities defined in that interface and implemented in other modules. We call the importer a *client* of the interface. A module may also *export* an interface, in which case it makes its own facilities available to other modules as defined by that interface. In Mesa, modules are identical to units of compilation; there is no provision for nesting modules within a single compilation unit. A collection of modules can be bound together into a *configuration* by the Mesa

binder or by a loader; this causes all imported interfaces to be connected to corresponding exported interfaces.

This section contains a brief, simplified description of Mesa interface definitions and of the configuration description language. At the end of the section is a note on the consistent compilation requirement, a constraint that has an important impact on the style and organization of any large system programmed in Mesa.

Interfaces

Interfaces are defined by compilation units called DEFINITIONS modules. An interface described by such a module can be partitioned into two parts, either of which can be empty. A *constant part* defines types and constants that can be used by any module with access to the interface. The *variable part* defines the operations available to clients importing the interface. In general, the operations are defined in terms of procedures and signals (dynamically bound unique names, used primarily for exception handling). The variable part of an interface may be thought of as a record consisting of procedure- and signal-valued fields. We call this an *interface record.* Figures 1a and 1b are excerpts from the definition of a hypothetical **Channel** interface. They illustrate the declarations typically found in the constant and variable parts respectively. Note that the attribute PRIVATE hides the definition of **Object** from clients; it is declared here because it is required for the declaration of **Handle**.

A module that uses an interface is said to *import* an instance of the corresponding interface record. Every module lists the DEFINITIONS modules for the interfaces that it imports. In essence, the importer is parametrized with respect to these interfaces. The compiler reads (the compiled version of) each of the imported modules and obtains all of the information necessary to compile the importing module. No knowledge about any implementors of the interfaces is required, but the types and parameters of all references to an interface are fully checked at compile time. The compiler also allocates space in the object program for (the required components of) the imported interface records but does not initialize it.

Similarly, a module that implements an interface is said to *export* it. Such a module contains procedure and/or signal declarations, each with the PUBLIC attribute, for the procedures and/or signals defined in the interface. The compiler ensures that the types in the exporter are assignment compatible with the corresponding fields of the interface record and thus with the types expected by importers of the interface. In essence, instantiation of an exporter yields an instance of the exported interface record in which procedure and signal descriptors have been assigned to the fields. Figure 1c suggests the form of a module that exports **Channel**. In this example, **ChannelImplementation** imports another interface, **Device**, so that it can use operations defined there.

The Mesa binder (and at run-time, the loader) collect exported interface records and assign their values to the corresponding interface records of the importers. The rules for collection and assignment are expressed in a configuration description language, which is discussed below.

The Mesa approach to interfaces has several important advantages:

> Once an interface has been agreed upon, construction of the importer and exporter can proceed independently. In particular, interfaces and implementations are decoupled. Not only is information better hidden, but minor programming bugs can be fixed in exporting modules without invalidating a previously established interface and without sacrificing full type-checking across module boundaries.

In large projects, interface specifications are units of communication among design and programming groups (see below under *Interfaces and Specifications*).

Interfaces partition the name space and effectively reduce the number of global names that must be kept distinct within a project.

Interfaces enforce consistency in the connections among modules. The operations upon a class of objects are collected into a single interface, not defined individually and in potentially incompatible ways. An earlier binding scheme, using component-by-component connection, could for example obtain *Allocate* from one module and *Free* from an entirely unrelated one.

Nearly all of the work required for the type-checking of interfaces is done by the compiler.

This approach should be contrasted with the alternatives. Interfaces in typical assembly language programming are defined implicitly, by attributes attached to symbols scattered through the text of the implementors. The associated binders (linkage editors) and loaders do no type checking and impose little structure on the use of names. Implementations of higher-level languages that are constrained to use the same binders seldom do any better, even when they offer strict intra-module type-checking. *We believe that the type-checking of interfaces is the most important application of the type machinery of Mesa.* In a few PASCAL derivatives (see for example [Kieburtz *et al*, 1978]), inter-module type-checking is provided by a special binder, but interfaces are still defined implicitly.

If importers and exporters refer to inconsistent versions of an interface, the type-checking scheme used by Mesa will fail. The following rather conservative approach has therefore been adopted to guarantee consistency. Whenever a **DEFINITIONS** module is compiled, the compiler generates a unique internal name for the interface (essentially a time stamp). Interfaces are 'the same' for the purposes of binding only if they have the same internal name. This rule is an extension of Mesa's equivalence rule for record types (see [Geschke *et al*, 1977] for further discussion). The compiler places the unique name of the interface in the object code generated for any importer or exporter compiled using that interface. *It is this internal name that is used by the binder or loader to match interfaces.* Thus the binder or loader checks that each interface is used in the *same version* by every importer and exporter.

This strategy has profound effects on the organization and management of large systems. It guarantees complete type-safety and consistency among all modules in a system communicating via a particular interface. On the other hand, it introduces both direct and indirect dependencies among modules to the level of exact versions; establishing consistency can require a great deal of recompilation. Subsequent sections discuss these issues.

Configurations and Binding

Mesa provides a separate *configuration description language*, C/Mesa, for specifying how separately compiled modules are to be bound together to form *configurations*. In the simple cases considered here, configuration descriptions are just lists of modules and (sub)configurations. These descriptions can be nested, however, and the nesting implicitly determines the scope of an interface according to the following rules:

A component of a configuration (i.e., a module or 'sub-configuration' named within the configuration description) may import an interface if and only if that interface is either imported by the configuration itself or is exported by some component of that configuration.

A configuration may export an interface only if it is exported by one of its components.

The Mesa configuration language is, in fact, more general than this; it has many of the attributes of a 'module interconnection language' as defined by [DeRemer and Kron, 1976]. C/Mesa provides such features as multiple, named instances of interfaces, the assignment of specific instances to specific importers, and the joint or shared implementation of an interface by more than one module. This generality is little used by Pilot and is not discussed here.

A complete system is represented by a hierarchy of configuration descriptions. The scope rules for interfaces permit an interface to be confined to, or excluded from, any given branch of the hierarchy. This can be best illustrated by an example. Let **A, B, C,** . . . be interfaces and let **U, V, W, X,** . . . be modules that import and export them as indicated in the comments. Consider the following three C/Mesa configuration descriptions:

```
Config1: CONFIGURATION
        IMPORTS A
        EXPORTS B =
BEGIN
    U;          --imports A, C
    V;          --exports B, C
END.

Config2: CONFIGURATION
        IMPORTS B =
BEGIN
    W;          --imports B, exports C
    X;          --imports B, C
END.

Config3: CONFIGURATION
        IMPORTS A =
BEGIN
    Config1;
    Config2;
END.
```

These configuration descriptions guarantee the following properties of the interfaces (among others):

> The scope of interface **C** in **Config1** is just that configuration; that is, this instance of **C** is known to all components of **Config1** but to no component outside of it. Every component of **Config1** that imports **C** will be bound to the same implementation, the one provided by **V**.
>
> The interface **C** in **Config2** is entirely independent of the interface **C** in **Config1**. Whether these two interfaces are different instances of the same interface definition does not matter; *they do not represent the same implementation.* All components of **Config2** that import **C** are bound to the implementation in **W**, not **V**.
>
> Interface **A** is imported into **Config3** (from some yet to be specified larger configuration), but it is imported only into the branch of the hierarchy represented by **Config1**. Thus no component of **Config2** may import **A**, even though it is known at a higher level in the hierarchy.

The scope rules for configurations provide a powerful tool for controlling the interactions among different parts of the system. Individual subgroups of the development team can define their own interfaces for their own purposes without involving larger units, without having to cope with unexpected calls from unrelated parts of the system, and without having any naming conflicts.

Similarly, the organization of the whole system is subject to scrutiny, and *all* interfaces between different parts of the system are fully exposed. No private, undocumented interfaces between low-level components in unrelated branches of the configuration hierarchy can exist.

Pilot makes extensive use of nested configurations to limit the scopes of interfaces. The configuration descriptions are organized as a four level heirarchy. The highest of these exports just the 'public' interfaces defined in the *Functional Specification* (see below). At the next level are the major internal interfaces, used for communication among the major subsystems of Pilot—e.g., input/output, memory management, etc. At lower levels are the interfaces that provide communication within a subsystem. At each level, the interfaces are defined and managed by the group or individual responsible for that configuration. This has been an important factor in keeping the logistics of the project manageable and its schedule reasonable.

Consistent Compilation

When one module is referenced during the course of compiling another, a *compilation dependency* is established. This dependency imposes a partial ordering on a collection of modules. If one module is changed and recompiled, all those that follow it in the ordering must also be recompiled before the collection is again consistent. It is seldom possible to bind a system together as long as any inconsistencies remain. An example illustrates the problem. Let **A** be an interface between modules **U** and **V**. If some change is required in **A**, it is a relatively simple matter to recompile first **A** and then **U** and **V**. These three are then consistent with each other and may be correctly bound together. If only **U** or only **V** were recompiled, the binder would report an error. Suppose, however, that interface **B** uses a type defined in **A**, say as the type of an object pointed to by a field of a record. Suppose further that modules **X** and **Y** communicate using **B**. If **X** also references **A**, any attempt to recompile **X** will fail until **B** is recompiled; then consistent binding requires recompilation of **Y** also. Thus **Y** has an indirect compilation dependency on **A**. Whenever **A** is recompiled, **B**, **X**, and **Y** must be also.

If the number of modules and interfaces in a system is large and if interfaces are evolving, ensuring this strictly-checked consistency becomes a major logistic problem for the project manager. The practical effect of this *consistent compilation* requirement is to force system designers to pay very close attention to when and how modules are updated. Without careful planning and system design, small changes to one or a few interfaces can trigger a recompilation of an entire system. For small systems, this is not significant, but for larger projects, it is a headache and it sacrifices many of the operational benefits of modularity. All members of the project must bring their work into phase and 'check in' their outstanding modules. These must then be recompiled in a sequence consistent with the partial order.

In our experience, such a universal recompilation effort nearly always reveals newly introduced inconsistencies and interactions between modules. These must be resolved immediately to allow the recompilation and rebinding to proceed. In the development of Pilot, the recompilation effort took more than a week the first time it was tried; this eventually converged to one-and-one-half or two days once the logistics were debugged. Note that this period is one of enforced inactivity among the members of the project—i.e., they are not able to continue coding and development of the system being integrated. (Because of the hierarchical structure of Pilot, universal recompilations were rare. In most cases, only the components of one of the nested configurations needed to be recompiled, requiring much less time and effort and affecting fewer people.)

The enforcement of consistent compilation is a result of Mesa's strict type- and version-checking at the module level. We have found that a utility program capable of computing the partial ordering and scheduling the required compilations is of great help in dealing with consistent compilation. Three more drastic alternatives can be imagined:

First, compatibility of interfaces might be defined recursively, in terms of component-by-component compatibility of types and values. This not only involves the binder in much more elaborate type checking but also requires access to large symbol tables during binding and loading. Previous use of this scheme in Mesa demonstrated that it had unacceptable performance and introduced a different set of operational problems.

Second, the compiler and binder could be more discriminating and enforce recompilation of **B**, **X**, and **Y** only when they are actually affected by the changes made to **A**. So far, attempts to do this in ways that do not reduce to the first alternative have not been very successful.

Finally, the onus could be placed on the programmer to recompile **B**, **X**, and **Y** when required. This, however, sacrifices the type-safeness of the Mesa language in one of the places where it is most required: at the interface between two modules. Failure to recompile at the appropriate times will result in a discrepancy between them *that is not apparent in any source text.* (In fact, one early version of Mesa, used 'unique' names that were incorrectly computed and were not always unique. We found that debugging in the presence of undetected version mismatches was extremely tedious and frustrating.)

The universal recompilation effort is, in effect, the root of a software release policy. Observe that the clients of Pilot itself must be recompiled whenever the external interfaces (those exported by Pilot) are recompiled. This, of course, can be very time-consuming and costly. Therefore, new releases of system software—i.e., new versions with updated interfaces—must be carefully planned and must not be undertaken lightly. 'Maintenence' releases, on the other hand, involve updates only to program modules or strictly internal interfaces. These releases can be absorbed very easily by clients at will and at the cost of a few seconds or minutes binding.

While consistent compilation is a logistic problem for the project manager, it is a programming benefit. If it becomes necessary to change an interface, the type- and version-checking done by the compiler and binder will detect all references to that interface and will expose all parts of the system that must be modified to accommodate the change. *The experience of many projects in Mesa is that once a previously running system has been successfully recompiled and rebound following changes to its internal or external interfaces, it will immediately run with the same reliability as before.* The correct use of strict interface checking is not always obvious, but it must be mastered if the potential benefits are to be obtained. (This parallels our experience with intra-module type checking.)

Programming in the Interface Language of Mesa

Designing interfaces and reducing them to Mesa **DEFINITIONS** modules is as much an act of programming as designing algorithms and reducing them to executable code. In Mesa, *interfaces are not derived* ex post facto *from the compiled modules constituting a system.* Most of the early 'programming' of Pilot was, in fact, interface programming, and one member of the design team was recognized as the 'interface programmer.' This was a senior member of the group who had the responsibility of ensuring that all interfaces were complete, were consistent with each other, and conformed to project standards.

The notion of an interface programmer did not exist *a priori* but arose from the methods used in the specification and design of Pilot. The original assignment of the interface programmer was to act as editor of the *Functional Specification*, a document describing the external characteristics of the Pilot operating system. However, it soon became apparent that Mesa text was an inherent part of this specification. In addition, while each of the designers contributed an interface and draft specification that was satisfactory for the area of his responsibility, the collection of these had to be integrated into a coherent whole. Thus, the editing task evolved into one resembling programming. The first part of this section illustrates the specification method and the use of the

Mesa interface language for defining the external characteristics of Pilot.

One of the most important responsibilities of the interface programmer was to ensure that there were no compilation dependencies between client programs and internal details of Pilot. This is not as easy as it sounds, and we had suffered some bitter experience in previous systems that failed to do this. In one case, a field of a record representing a low-level data structure was accidentally omitted in some code shared between Pilot and the Mesa system itself. The omission did not affect the operation of the Mesa system and was discovered only after most of the testing of a new release of that system had been completed. Unfortunately, the **DEFINITIONS** module in which the record was located was near the root of the tree of compilation dependencies and, because of schedule commitments, could not be corrected prior to release. As a consequence, all versions of Pilot built on that release of Mesa had to avoid using a fundamental feature of the system architecture. Considerable pains were taken in the subsequent design of the Pilot interfaces to avoid this kind of problem. The second part of this section describes a language feature that helps to minimize such undesirable interactions.

The third part of this section describes how the explicit and strictly checked interfaces of Mesa permitted the functional simulation of Pilot using an older operating system. Contrary to our expectations and previous experience in operating system design, the conversion of the client programs from the simulated system to the real one was painless.

Interfaces and Specifications

The interface language of Mesa served as the nucleus of the functional specification of the Pilot operating system. This provided a means for defining the scope and character of the system, for documenting it for clients and potential clients, and for focusing the programming effort.

In this particular project, two versions of a *Functional Specification* document were prepared before coding began. The first of these was the culmination of a long study in which the general nature of the system, its goals, and its requirements were identified. The first version of the *Functional Specification* was circulated and detailed design of the system began. Approximately six months later, the second version of the *Functional Specification* was prepared. It incorporated changes and refinements resulting from the design effort and from comments by the client organizations. Following this, Pilot was coded and tested for a period of approximately six months. Finally, the *Functional Specification* was edited to make minor changes and distributed as a programmer's reference manual.

The external specification of Pilot at the functional level is essentially a specification of its public interfaces—i.e., of the types and constants defined by the system, of the procedures that clients can call, and of the signals representing error conditions detected by the system. These interfaces consist of approximately a dozen **DEFINITIONS** modules representing the major functional areas of the system. They are named according to function, e.g., **File** and **Volume** to describe the file storage system, **Space** to describe memory management, etc.

Figure 2 illustrates two fragments of the *Functional Specification* for the **File** interface. The two parts of the figure illustrate, respectively, the definition of the notion of file capabilities in this system and the operation for creating files. File capabilities are simply and conveniently described in terms of the type **File.ID** (described earlier in the *Functional Specification*). The null value of a file capability is also defined at this point in terms of **File.nullID**, a previously defined null value of **File.ID**, and the empty set. Figure 2a contains all of the information about file capabilities needed by a Mesa programmer designing a client of Pilot, and it illustrates the self-documentating nature of high-level languages such as Mesa.

In Figure 2b, the file creation operation is defined. First, definitions of the procedure and associated error signals are presented as they appear in the interface (note that the **Create** operation defined in the **File** interface can cause signals defined in the **Volume** interface to be raised). Following this is a narrative describing the function of the **Create** operation and the error responses that can occur. The initial state of the file is fully defined (including values of attributes defined elsewhere in the *Functional Specification*). The **type** attribute of the file is defined in conjunction with **Create** and consists of a CARDINAL (i.e., non-negative integer) encapsulated in a record (to create a unique type).

When the *Functional Specification* was completed, the Mesa text was extracted using a text editor, embedded in a prototype DEFINITIONS module and compiled. This revealed a host of minor errors and several circularities. Several omissions were also detected, indicating that the document was incomplete in these respects. These, of course, were corrected both in the interfaces and in the document. The result was twofold: First, the interfaces compiled from the document became the 'official' versions and were used in the implementation. Second, we had confidence that we had adequately documented the whole system as an integral part of its development, in advance and not as a last minute chore.

From the Pilot experience, we conclude that the combination of Mesa and English in the style we have described is an effective specification tool. There is no formal or mechanical verification method to ensure or 'prove' that the resulting system satisfies the specifications. Nevertheless, our experience has been that human 'verification' is tractable; i.e., the redundancy in this description plus ordinary debugging and testing techniques are sufficient to convince us that the operating system meets its specifications with a reasonable degree of reliability. There were very few cases in which the specifications were misinterpreted or interpreted differently by different people.

A Note on Exporting Types

At the time Pilot was developed, Mesa did not permit modules to export types, only procedures and signals. Constants and types could, of course, be declared in interfaces, but these were known at compile-time to both the importers and the exporters of the interfaces. Unlike procedures and signals, types to be used by one module could not be bound at some later time to types defined by implementation modules elsewhere. Thus every module using instances of a type had to be compiled in an environment in which that type was completely defined, even if the compilation actually required no knowledge of the internal structure of the type.

This restriction introduced unreasonable compilation dependencies between implementation details and the external interfaces of Pilot. This is partly a result of the 'object' style of programming. Consider, for example, the specification of the **Channel** interface introduced previously. The desired interface must provide the type **Channel.Handle**, to be used by Pilot to identify objects describing channels, and a number of operations, such as **Channel.Create**, requiring handles as arguments or returning them as results. Figures 1a and 1b suggest the obvious mapping of these requirements into a Mesa DEFINITIONS module.

While Figure 1 shows the most type-safe way to define a **Channel.Handle**, that interface has a serious operational shortcoming. A client program is not concerned with the actual values of **Channel.Handle**; it only stores them and passes them as parameters. The implementation might use a pointer, an array index, or some other kind of token to represent a **Channel.Handle**. In particular, it should be free to change its representation without impacting **Channel** clients (i.e., without forcing them to be recompiled). Unfortunately, the definition in Figure 1 requires a commitment to the representation of not only **Channel.Handle** but also **Channel.Object** at the time the interface is defined. The only flexibility retained by the implementor is in the algorithms and data structures hidden within **ChannelImplementation**. Thus, fixing bugs and improving the system behaviour must be confined to major releases of Pilot, at which time it is expected that all clients will, at least, be recompiled.

Clients also suffer in this approach. Because the representation of the Channel.**Object** is clearly exposed in the interface (even though it is marked PRIVATE), the client programmer is tempted to make unwarranted assumptions about the properties of the objects. Indeed, he can even reference objects directly (using a very simple breach of the type system subject only to administrative control), rather than via the exported procedures of the interface. If the implementation of channels is changed in a subsequent release of Pilot, the client program must be revised, not just recompiled.

In Pilot, introducing implementation details into public interfaces was avoided by carefully placed breaches of the type system. The Mesa version of the **Channel** specification was defined as shown in Figure 3. The declaration in Figure 3a defines Channel.**Handle** to be a unique record type occupying one word of storage. This change has no effect on clients of the interface (see Figure 3b) and does not sacrifice type checking of channel handles within clients. The actual representation of the Channel.**Handle** is defined in the implementation module as suggested by Figure 3c, where the ***LOOPHOLE*** construct changes the type of its first argument to its second argument, with no change in representation.

Note that the implementation module can be recompiled whenever necessary and rebound to the rest of the system without affecting any interfaces. In particular, the implementation details of the embedded types **Object** and **InternalHandle** (except the latter's size) can be changed at will. The type **InternalHandle** is bound at compile time to the current version of **Object**, but the type Channel.**Handle** is constant for the life of the interface.

This need to breach the type system to minimize compilation dependencies has suggested an improvement to the Mesa language, namely the exporting of types. To do this, we replace the declaration of Channel.**Handle** in Figure 3a by:

```
Handle: TYPE;
```

This defines Channel.**Handle** to be a type that will be bound at a later time. An implementation module then exports the type in exactly the same way it exports procedures—by declaring a PUBLIC type with the required name. In Figure 3c, the declaration of **InternalHandle** is replaced by:

```
Handle: PUBLIC TYPE = POINTER TO Object;
```

references to **h1** are replaced by references to **h**, and the assignments using ***LOOPHOLE***s are removed. Breaches of the type system are no longer required in the source code. Clients of **Channel** are unaffected. The binder checks that each exported type is exported by precisely one implementing module and that therefore all modules of a configuration refer to the same type. The only information that needs to be known about the type when the interface is designed or a client is compiled is the size of its representation. Note that an exported type does not have a run-time representation that is available to clients; only the exporter can have any knowledge of the internal structure of that type.

Functional Simulation of the Pilot Operating System

A side effect of the explicit definition of interfaces in separate compilation units is that the same set of interfaces can be implemented by two different systems, and a client can be bound to either one. Provided that corresponding procedures of the two systems implement the same 'semantics,' the client perceives no functional difference between them. This proved to be a valuable feature for the early clients of Pilot. To allow them to begin their own testing before Pilot was complete, a simulated version of Pilot was provided using an older operating system.

This simulated version used exactly the same interfaces (i.e., source and object **DEFINITIONS** modules) as the real one. It consisted of only a small amount of code that converted calls upon Pilot procedures into calls upon old operating system procedures. In the configuration description of the simulated system, all interfaces of the old system were carefully concealed from clients. For all of the basic operating system facilities, the simulated system and the real one provided virtually identical functional behaviour.

The conversion from the simulated environment to the real environment took very little time and effort. In one typical case, an operational version of an application system was demonstrated using the simulated Pilot system. Within two weeks, it was operational on the real system and had successfully executed the same tests as it had in the simulated environment. We attribute this success primarily to the strongly checked interfaces of Mesa which, along with the English narrative in the *Functional Specification*, provided sufficient redundancy to permit the implementation of exactly the same functions on two different systems.

This simulated system was not our first attempt. In an earlier effort, the old operating system interfaces were not concealed, and the interface modules of the simulated system were only 'approximately' the same as those of the real system. As a result, conversion from the simulated system was a very painful process. Programs that worked well on the simulated system needed extensive revision prior to conversion because (much to the surprise of their implementors!) they were found to contain extensive dependencies upon the facilities of the old system, which were still available and visible.

Adequacy of Mesa as a System Programming Language

Previous sections have discussed some potential benefits of high-level languages, particularly in the areas of consistency checking, information hiding, and control of interfaces. These languages offer other well known advantages, such as greater descriptive power and the suppression of many coding details. A question often raised, however, is whether a language such as Mesa is adequate for implementing components of 'real' systems, especially very low-level programs such as the kernel of an operating system or a device driver. In the case of the Pilot project, the answer is an unqualified 'yes.' *All* system software, including all run-time support for the language, trap handlers, interrupt routines, etc., is coded in Mesa. Even a bootstrap loader that fits into a single 256-word disk block has been written in Mesa.

We must, however, expand upon our answer. In our opinion, several easily overlooked characteristics of our environment contributed substantially to our success. The more important of these are discussed in this section.

Access to the Hardware

Mesa was designed to provide complete but controlled access to the underlying machine. There are several aspects of this. Note that the features described below appear quite infrequently in our code, and the use of most of them is subject to strict administrative control. Each one, however, seems crucial in certain situations.

The programmer has the option of specifying the representation to be used for a particular type. If, for example, the attribute **MACHINE DEPENDENT** is attached to a record declaration, the mapping between the fields of that record and the bit positions in its representation is precisely defined and guaranteed by the compiler. An important use of this attribute is to create structures that exactly match hardware-defined formats; thereafter, interaction with the hardware can be described symbolically. A somewhat less satisfying use is to specify the formats of records placed on

secondary storage media. The Mesa system is still evolving; each release defines a 'virtual machine' that may differ from its predecessors in certain details. Any data structure likely to outlive a particular release is, in effect, dependent upon the virtual machine that created it. Clients are encouraged to recognize this dependency explicitly, either by specifying some fixed format in the original declaration or by inventing their own unique naming scheme for version control.

The Mesa language allows explicit breaches of the type system. For essentially the same reasons reported previously [Geschke *et al*, 1977], we have made modest use of such breaches, often to decode representations. Trap handling, for example, sometimes requires inspection of a procedure descriptor as a string of bits. We use another breach, the assignment of an integer to a pointer, to access hardware-defined memory locations. This is one of the rare cases in which a non-pointer value must be assigned to a pointer, and it is almost always done by a constant declaration in an internal **DEFINITIONS** module rather than by an executable program.

The language also makes available the low-level 'transfer' primitive, as defined in [Lampson *et al*, 1974], for the transfer of control between contexts. Use of this primitive sacrifices readability and a certain amount of type checking; in conjunction with the heap (non-stack) allocation of frames, however, it has allowed us to experiment with unconventional control structures and to implement the lowest levels of trap handlers, interrupt routines, process schedulers and the like in Mesa.

Finally, Mesa permits bodies of procedures to be specified as sequences of machine instructions. When one of these procedures is called, that sequence is compiled 'inline' in the body of the caller. This facility permits direct access to any special operations of the machine not reflected in the Mesa language, such as I/O control, interrupt masking, etc.

Efficiency

Implementing Mesa on a microprogrammed machine has given us the opportunity to design an instruction set that is well-matched to the requirements of the language. In our experience, space has proved more critical than time in most systems for small, personal computers; overall performance depends more upon the amount of primary memory available than on raw execution speed. We have therefore emphasized compactness in our design.

Mesa object code is very compact. This is primarily due to the design of the instruction set itself. We used techniques for program analysis similar to those described in [Sweet, 1978] to discover common operations and to choose efficient encodings of them. The current compiler does little global analysis and optimization, but extensive 'peephole' optimization does contribute further to the compactness of the object code. That code is considerably more compact than the code produced by most other compilers known to us, even those that perform extensive optimization. In fact, Mesa object code is often more compact than good assembly code for machines with a conventional instruction set.

We have been careful to define operations that have reasonable implementations in microcode. Execution speed is therefore adequate also; critical timing-dependent code, such as a disk interrupt handler that operates on each sector, can be satisfactorily programmed in Mesa without making undue demands on processor time. We seldom find it necessary to resort to obscure coding styles to achieve fast programs; when bottlenecks are discovered, it is often more profitable to improve the microcode.

Tools

Another essential requirement for programming in a langauge such as Mesa is a set of tools that maintain the illusion of a Mesa 'virtual machine'. The most notable of these is a powerful source-

language debugger, which is routinely used by all Mesa programmers. To allow the debugging of programs such as Pilot itself, our debugger operates on the 'world swap' principle. Embedded in the program to be debugged is a small *nub* which fields traps, faults, breakpoints, and other conditions. Using a few carefully chosen primitive operations, this nub causes the entire state of the memory to be saved on a file and then loads a debugging system to examine that file. Because of the swap, an errant program cannot damage the debugger, and the debugger is not dependent upon the system being debugged for any of its operations.

The debugger provides the usual facilities; for example it is possible to display variables symbolically, to set conditional breakpoints and to display the state or call stack of any process. All interactions with the programmer are symbolic and are expressed in terms of his original program. Thus each displayed value is formatted according to its type, the original source code is used to specify the location of a breakpoint, etc. In addition, the debugger contains an interpreter of a subset of Mesa; it is valuable for following paths through data structures, setting variables, and calling procedures in the user's memory image.

System Integration

The entire Mesa system is integrated and can evolve to meet new requirements as they are recognized. We can influence all levels of the implementation; to add new facilities or remove a bottleneck, changes can be made where they are most appropriate.

The evolution of processes in Mesa demonstrates this. Earlier versions of the language had no special support for processes in any form. Because of the accessibility of the underlying machine, particularly the transfer primitives, users were able to write their own packages supporting process creation and scheduling. In fact, several such packages were written, each designed to perform well for certain classes of applications. Most of the packages were mutually incompatible, however, and since the language had no notion of a 'process' or 'critical section', the compiler could offer no help in checking for process-related inconsistencies.

After much discussion of the alternatives, we decided to adopt a 'procedure-oriented model' of processes [Lauer and Needham, 1978] as our standard. The concepts of processes, monitors, and condition variables were added to the language. While it is possible (and, at the lowest levels of the system, sometimes necessary) to ignore these additions, they provide a standard way of programming that is adequate for most applications. The compiler was extended not only to accept these constructs but also to check for obvious inconsistencies in their use. In our initial implementation, process scheduling was done largely in software; this was relatively easy and gave us some flexibility for experimentation. Subsequently, certain parts of the scheduler were moved into microcode to obtain a substantial performance improvement.

Conclusions

The correct uses of the type system, interface language, and configuration language of Mesa are not always obvious. They must be mastered both by individuals and by organizations if the benefits are to be obtained. The benefits, however, can be very substantial. Mesa provides a measure of control over the design and development of systems that greatly exceeds anything else available to us within the resources of a modest-sized development project. As a result, sophisticated systems can be implemented robustly and reliably by small groups within reasonable times. One of the most important practical benefits of Mesa is that the 'easy' bugs are eliminated almost at once and the 'hard' bugs are encountered much sooner in the life of a system.

Acknowledgements

Many of our colleagues have shared experiences and insights that contributed to the ideas expressed in this paper. We are particularly indebted to the other implementors of Mesa and Pilot. Butler Lampson, Charles Simonyi and John Wick made major contributions to the design of Mesa's interfaces and configuration descriptions.

References

DeRemer, F., and Kron, H. H., "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering* **SE-2** 2 (June 1976), pp. 80-86.

Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., "Early Experience with Mesa," *Communciations of the ACM* **20** 8 (August 1977), pp. 540-553.

Kieburtz, R. B., Barabash, W., and Hill, C. R., "A Type-Checking Program Linkage System for Pascal," in *Proceedings 3rd International Conference on Software Engineering,* (Atlanta, May 1978), pp. 23-28.

Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L., "Report on the Programming Language Euclid," *Sigplan Notices* **12** 2 (February 1977).

Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H., "On the transfer of control between contexts," in *Lecture Notes in Computer Science, Vol. 19,* G. Goos and J. Hartmannis, Eds., Springer-Verlag, New York (1974), 181-203.

Lauer, H. C., and Needham, R. M, "On the Duality of Operating System Structures," in *Proceedings of the Second International Symposium on Operating Systems,* IRIA, Rocquencourt, France, October 1978.

Lynch, W. C., and Horsley, T. R. "Pilot: a Software Engineering Case Study," submitted to this conference, 1979.

Mitchell, J. G., Maybury, W., and Sweet, R. E., *Mesa Language Manual,* Technical report CSL-78-1, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, February 1978.

Sweet, R. E., *Empirical Estimates of Program Entropy,* Technical report CSL-78-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, September 1978.

Wirth, N., "The programming language Pascal," *Acta Informatica* **1** (1971).

```
Object: PRIVATE TYPE = RECORD [...];

Handle: TYPE = POINTER TO Channel.Object;

nullHandle: Channel.Handle = NIL;
```

Figure 1a

```
Create: PROCEDURE [a: arguments] RETURNS [h: Channel.Handle];
Operation: PROCEDURE [h: Channel.Handle, a: arguments];
```

Figure 1b

```
ChannelImplementation: PROGRAM IMPORTS Device EXPORTS Channel =
BEGIN
    ...
    Create: PUBLIC PROCEDURE [a: arguments] RETURNS [h: Channel.Handle] =
    BEGIN
        ...
    END;
    ...
    Operation: PUBLIC PROCEDURE [h: Channel.Handle, a: arguments] =
    BEGIN
        ...
    END;
    ...
END.
```

Figure 1c

A File.**Capability** is an encapsulation of a File.**ID**, along with a set of **permissions**, and is used to represent the right to perform a specific set of operations on a specific file or volume.

```
File.Capability: TYPE = PRIVATE RECORD [
    fID: File.ID, permissions: File.Permissions];
File.Permissions: TYPE = SET OF {read, write, grow, shrink, delete};
File.nullCapability: File.Capability = [fID: File.nullID, permissions: {}];
. . .
```

Note: Capabilities are *redundant specifications of intent*, not "ironclad" vehicles for protection. If a client program conscientiously limits the permissions in its capabilities to those it expects to use, it will reduce its chances of accidentally destroying its own data in case of minor hardware or software malfunctions.

Figure 2a

```
File.Create: PROCEDURE [volume: Volume.ID, initialSize: File.PageCount,
    type: File.Type] RETURNS [file: File.Capability];
File.Error: ERROR [type: File.ErrorType];
File.ErrorType: TYPE = {reservedType, . . .};
Volume.InsufficientSpace: ERROR;
Volume.Unknown: ERROR [volume: Volume.ID];
```

The **Create** operation creates a new file on the specified volume. The operation returns a File.**Capability** (with all permissions) for the new file. If **volume** does not name a volume known to Pilot, Volume.**Unknown** is signaled. The signal Volume.**InsufficientSpace** is generated if there is not enough space on the volume to contain the file. The file initially contains the number of pages specified by **initialSize** (filled with zeros) and has the following other attributes (see §*5.2.5*):

type = **type** parameter to **Create**
immutable = FALSE
temporary = TRUE

The **type** attribute of the file is a tag provided by Pilot for the use of higher level software. . . .

```
File.Type: TYPE = RECORD [CARDINAL];
```

The type of a file is set at the time it is created and may not be changed. . . .

Create may signal File.**Error[reservedType]** if its **type** argument is one of a set of values reserved by the Pilot file implementation.

Figure 2b

```
Handle: TYPE = PRIVATE RECORD[UNSPECIFIED];
```

Figure 3a

```
Create: PROCEDURE [a: arguments] RETURNS [h: Channel.Handle];
Operation: PROCEDURE [h: Channel.Handle, a: arguments];
```

Figure 3b

```
ChannelImplementation: PROGRAM IMPORTS Device EXPORTS Channel =
BEGIN
        Object: TYPE = RECORD [ . . . ];
        InternalHandle: TYPE = POINTER TO Object;
        . . .
        Create: PUBLIC PROCEDURE [a: arguments] RETURNS [h: Channel.Handle] =
        BEGIN
                h1: InternalHandle;
                . . .
                h1 ← . . . ;
                . . .
                h ← LOOPHOLE[h1, Channel.Handle];
        END;
        . . .
        Operation: PUBLIC PROCEDURE [h: Channel.Handle, a: arguments] =
        BEGIN
                h1: InternalHandle = LOOPHOLE[h, InternalHandle];
                . . .
        END;
        . . .
END.
```

Figure 3c