Pilot: A Software Engineering Case Study

by Thomas R. Horsley and William C. Lynch

July 10, 1979

ABSTRACT

Pilot is an operating system implemented in the strongly typed language Mesa and produced in an environment containing a number of sophisticated software engineering and development tools. We report here on the strengths and deficiencies of these tools and techniques as observed in the Pilot project. We report on the ways that these tools have allowed a division of labor among several programming teams, and we examine the problems introduced within each different kind of development programming activity (ie. source editing, compiling, binding, integration, and testing).

XEROX
SYSTEMS DEVELOPMENT DEPARTMENT
3408 Hillview Ave / Palo Alto / California 94304

Introduction

The purpose of this paper is to describe our experiences in implementing an operating system called Pilot using a software engineering support system based on the strongly typed language Mesa [Geschke *et al*, 1977, Mitchell *et al*, 1978], a distributed network of personal computers [Metcalfe *et al*, 1976], and a filing and indexing system on that network designed to coordinate the activities of a score or more of programmers. In this paper we will present a broad overview of our experience with this project, briefly describing our successes and the next layer of problems and issues engendered by this approach. Most of these new problems will not be given a comprehensive discussion in this paper, as they are interesting and challenging enough to deserve separate treatment.

That the Mesa system, coupled with our mode of usage, enabled us to solve the organizational and communication problems usually associated with a development team of a score of people. These facilities allowed us to give stable and non-interactive direction to the several sub-teams.

We developed and used a technique of incremental integration which avoids the difficulties and schedule risk usually associated with system integration and testing.

The use of a Program Secretary, not unlike Harlan Mills' program librarian, proved to be quite valuable, particularly in dealing with situations where our tools had weaknesses. We showed the worth of the program librarian tool, which helped coordinate the substantial parallel activity we sustained; and we identified the need for some additional tools, particularly tools for scheduling consistent compilations and for controlling incremental integrations.

We determined that these additional tools require an integrated data base wherein consistent and correct information about the system as a whole can be found.

Background

Pilot is a medium-sized operating system designed and implemented as a usable tool rather than as an object lesson in operating system design. Its construction was subjected to the fiscal, schedule, and performance pressures normally associated with an industrial enterprise.

Pilot is implemented in Mesa, a modular programming system. As reported in [Mitchell *et al*, 1978], Mesa supports both definitions and implementing modules (see below). Pilot is comprised of some 92 definitions modules and 79 implementation modules, with an average module size of approximately 300 lines.

Pilot consists of tens of thousands of Mesa source lines; it was implemented and released in a few months. The team responsible for the development of Pilot necessarily consisted of a score of people, of which at least a dozen contributed Mesa code to the final result. The coordination of four separately managed sub-teams was required.

There are a number of innovative features in Pilot, and it employs some interesting operating system technology. However, the structure of Pilot is not particularly relevant here and will be reported in a series of papers to come [Redell *et al*, 1979], [Lampson *et al*, 1979].

Development Environment and Tools

The hardware system supporting the development environment is based on the Alto, a personal interactive computer [Lampson 1979], [Boggs, *et al*, 1979]. Each developer has his own personal machine, leading to a potentially large amount of concurrent development activity and the potential for a great degree of concurrent development difficulty. These personal computers are linked together by means of an Ethernet multi-access communication system [Metcalfe *et al*, 1976]. As the Altos have limited disk storage, a file server machine with hundreds of megabytes of storage is also connected to the communications facility. Likewise, high-speed printers are locally available via the same mechanism. The accessing, indexing, and bookkeeping of the large number of files in the project is a serious problem (see below). To deal with this, a file indexing facility (librarian) is also available through the communications system.

The Alto supports a number of significant wideranging software tools (of which the Mesa system is just one) developed over a period of years by various contributors. As one might imagine, the level of integration of these tools is less than perfect, which led to a number of difficulties and deficiencies in the Pilot project. Many of these tools were constructed as separate, almost stand-alone systems.

The major software tools which we employed are described below.

Mesa is a modular programming language [Geschke *et al*, 1977]. The Mesa system consists of a compiler for the language, a Mesa binder for connecting the separately compiled modules, and an interactive debugger for debugging the Mesa programs. Optionally, a set of procedures called the Mesa run-time may be used as a base upon which to build experimental systems.

The language defines two types of modules: *definitions* modules and *implementation* modules. Both of these are compiled into binary (object) form. A definitions module describes an interface to a function by providing a bundle of procedure and data declarations which can be referenced by *client programs (clients)*. Declarations are fully type specified so that the compiler can carry out *strong type checking* between clients and implementation modules. The relevant type information is supplied to the clients (and checked against the implementations) by reading the object modules which resulted from previous compilation(s) of the relevent definitions module(s). The implementing modules contain the procedural description of one or more of the functions defined in some definitions module. Since an implementing module can be seen only through some definitions module, a wide variety of implementations and/or versions is possible without their being functionally detectable by the clients. Thus Mesa enforces a form of information hiding [Parnas, 1972].

The Mesa binder [Mitchell *et al*, 1978] defines another language, called C/Mesa, which is capable of defining *configurations*. These assemble a set of modules and/or sub-configurations into a new conglomerate entity which has the characteristics of a single module. Configurations may be nested and used to describe a tree of modules. Configurations were used in the Pilot project as a management tool to precisely define the resultant output of a contributing development sub-team.

Another software tool is the Librarian. It is designed specifically to index and track the history of the thousands of files created during the project. In addition to its indexing, tracking, and status reporting functions, the Librarian is constructed to adjudicate the frequent conflicts arising between programmers attempting to access and update the same module.

Organization, Division, and Control of the Development Effort

The size of the Pilot development team (itself mandated by schedule considerations) posed the usual organizational and management challenges. With 20 developers, a multi-level management structure was necessary despite the concomitant human communication and coordination problems.

As described below, we chose to use the modularization power of the Mesa system to address these problems, rather than primarily providing the capability for rapid interface change as reported in [Mitchell, 1978]. The resultant methodology worked well for the larger Pilot team. We believe that this methodology will extrapolate to organizations at least another factor of five larger and one management level deeper. A description and evaluation of this methodology are the topics of this section.

Another aspect of our approach was the use of a single person called the Program Secretary, a person not unlike the program librarian described by Harlan Mills [Mills, 1970] in his chief programmer team approach. As we shall describe, the Secretary performed a number of functions which would have been very difficult to distribute *in our environment.* This person allowed us to control and make tolerable a number of problems, described below, which for lack of time or insight we were not able to solve directly.

The Pilot Configuration Tree

We organized Pilot into a tree of configurations isomorphic to the corresponding people tree of teams and sub-teams. The nodes of the Pilot tree are C/Mesa configuration descriptions and the leaves (at the bottom of the tree) are Mesa implementation modules. By strictly controlling the scope (see below) of interfaces (through use of the facilities of the configuration language C/Mesa), different branches of the tree were developed independently. The configuration tree was three to four layers deep everywhere. The top level configuration implements Pilot itself. Each node of the next level down maps to each of the major Pilot development teams, and the next lower level to sub-teams. At the lowest level, the modules themselves were usually the responsibility of one person. This technique of dividing the labor in correspondence with the configuration tree proved to be a viable management technique and was supported effectively by Mesa.

Management Of Interfaces

It quickly became apparent that the *scope* of an interface was an important concept. It is important because it measures the number of development teams that might be impacted by a change to that interface. The *scope* of an interface is defined as the least configuration within which all clients of that interface are confined. This configuration corresponds to the lowest C/Mesa source module which does *not* export the interface to a containing configuration. Thus the scope of a module may be inferred from the C/Mesa sources. The impact of a change to an interface is confined to the development organization or team that corresponds to the node which is the scope of the interface. Thus the scope directly identifies the impacted organization and its sub-organizations.

The higher the scope of an interface, the more rigorously it must be (and was) controlled and the less frequently it was altered since changes to high scope interfaces impact broader organizations. Changing a high level interface was a management decision requiring careful project planning and longer lead times, while a lowest-level interface could be modified at the whim of the (usually) individual developer responsible for it. In general, changing an interface required project planning at the organizational level corresponding to its scope. In particular, misunderstandings between development sub-teams about interface specifications were identified early at design time rather than being discovered late at system integration time. Also obviated were dependencies of one

team on another team's volatile implementation details. The result of all of this was 1) the elimination of schedule slips during system integration by the elimination of nasty interface incompatibility surprises and, even stronger, 2) the reduction of system integration to a pro-forma exercise by the (thus enabled) introduction of *incremental integration* (see below).

[Mitchell 1978] reported good success with changing Mesa interface specifications, followed by corresponding revisions in the implementing modules and a virtually bug-free re-integration. While we also found this to be a valid and valuable technique for low-level interfaces (the scope of which corresponded to a three-to-five-person development sub-team), the project planning required to change high-level interfaces affecting the entire body of developers was obviously much greater as was the requirement for stability of such interfaces. It should be noted that the experience reported by [Mitchell 1978] refers to a team of less than a half dozen developers.

Thus, we chose to use the precise interface definition capabilities and strong type checking of the Mesa system differently for the high-level interfaces than for the low-level ones. High-level interfaces were changed only very reluctantly, and were frozen several weeks prior to system integration. This methodology served to decouple one development team from another since each team was assured that they would not be affected by the on going implementation changes made by another developer. Each could be dependent only on the shared definitions modules, and these were controlled quite carefully and kept very stable [Lauer *et al*, 1976].

The Master List

As the system grew, it became painfully obvious that we had no single master description of what constituted the system. Instead we had a number of overlapping descriptions, each of which had to be maintained independently.

One such description was the *working directory* on the file server. Its subdirectory structure was a representation of the Pilot tree. Another description of this same tree was embodied in the librarian data base which indexed the file server. Yet another description was implicit in the C/Mesa configuration files. Early in the project we found it necessary to create a set of command files for compiling and binding the system from source; these files contained still another description of the Pilot tree.

The addition of a module implied manually updating each of these related files and data bases; it was a tedious and error prone process. In fact, not until the end of the project were all of these descriptions made consistent.

We never did effect a good solution to this problem. We dealt with it in an *ad hoc* fashion by establishing a rudimentary data base called the Master List. This data base was fundamental in the sense that all other descriptions and enumerations were required to conform to it. A program was written to generate from the Master List some of the above files and some of the required data base changes.

A proper solution to this problem requires merging the various lists into a single, coherent data base. This implies that each tool take direction from such a data base and properly update the data base. Since many of the tools were constructed apart from such a system, they would all require modification. Thus the implementation of a coherent and effective data base is a large task in our environment.

Incidentally, this problem was one of those controlled by our Program Secretary. It is quite clear what chaos would have resulted if the updating of the numerous lists described above had not been concentrated in the hands of a single developer.

Pilot Update Cycle

In this section we will examine some of the interesting software engineering aspects of the inner loop of Pilot development. This inner loop occurs after design is complete and after a skeletal system is in place. The typical event consists of making a coordinated set of changes or additions to a small number of modules.

In our environment, a set of modules is fetched from the working directory on the file server to the disk on the developers personal machine. Measures must be taken to ensure that no one changes these modules without coordinating these modifications with the other developers. Usually edits are made to the source modules; the changed modules (and perhaps some others) are recompiled; and a trial Pilot system is built by binding the new object modules to older object modules and configurations. The resulting system is then debugged and tested using the symbolic Mesa debugger and test programs which have been fetched from the working directory. When the system is operating again (usually a few days later), the result is integrated with the current contents of the working directory on the file server, and the changed modules are stored back onto the working directory.

A number of interesting problems arise during this cyclic process:

Consistent Update Of Files

Pilot has been implemented in the context of a distributed computing network. The master copies of the Mesa source modules and object modules for Pilot are kept in directories on a file server on the network. In order to make a coordinated batch of changes to a set of Pilot source files, the developer transfers the current copies of the files from the file server to his local disk, edits, compiles, integrates, and tests them, and then copies them back to the file server.

This simple process has a number or risks. Two developers could try to change the same file simultaneously. A developer could forget to fetch the source, and he would then be editing an old copy on his local disk. He could fetch the correct source but forget to write the updated version back to the file server.

All of these risks were addressed (after the project had begun) by the introduction of the program librarian server. This server indexes the files in the file server and adjudicates access to them via a checkin/checkout mechanism. To guarantee consistency between local and remote copies of files, it provides atomic operations for "checkout and fetch the file" and "checkin and store the file". In the latter case, it also deletes the file from the local disk, thus removing the possibility of changing it without having it checked out (n.b. check-in is prevented unless the developer has the module currently checked out).

Consistent Compilation

Each Mesa object file is identified by its name and the time at which it was created; it contains a list of the identifications of all the other object modules used in its creation (e.g., the definitions module it is implementing). The Mesa compiler will not compile a module in the presence of definitions modules which are not *consistent*, nor will the the binder bind a set of inconsistent object modules. *Consistent* is loosely defined to mean that, in the set of all object modules referenced directly or indirectly, there is no case of more than one version of a particular object module. Each recompilation of a source module generates a new version.

For example, module A may use definitions modules B and C, and definitions module B may also refer to C. It can easily happen that we compile B using the original compilation of C, then we

edit the source for C "slightly" and recompile, and then we attempt to compile A using C (the new version) and using B (which utilized the original version of C). The compiler has no way of knowing whether the "slight" edit has created compatibility problems, so it "plays safe" and announces a consistency error.

Thus, editing a source module implies that it recompile not only itself, but also all of those modules which include either a direct or an indirect reference to it. Correctly determining the list of modules to be recompiled and an order in which they are to be recompiled is the *consistent compilation* problem.

This "problem" is, in fact, not a problem at all but rather an aid enabled by the strong type checking of Mesa. In previous systems the developer made the decision as to whether an incompatibility had been introduced by a "slight" change. Subtle errors due to the indirect implications of the change often manifested themselves only during system integration or system testing. With Mesa, recompilation is forced via the Mesa systems auditing and judging the compatability of all such changes, thus eliminating this source of subtle problems.

A consistent compilation order for a system (such as Pilot) having a configuration tree can be determined largely by the following analysis:

1) As a direct consequence of the consistency requirement, two modules cannot reference each other, nor can any other cyclical dependencies exist; otherwise the set cannot be compiled. This implies the existence of a well-defined order of compilation.

2) Pilot implementation modules may not refer to each other but must refer only to definitions modules. Therefore only those implementation modules which import recompiled definitions modules need themselves be recompiled. Such implementation modules are recompiled in any order after the recompilation of the definitions modules.

3) An individual definitions module can have compilation dependencies only on modules having the same or a higher scope (from the definition of *scope*). The proper compilation order for definitions modules with different scopes is thus determined by the C/Mesa configuration sources (compile the one with the higher scope first). The Pilot tree of configurations thus imposes a global and fairly restrictive partial ordering on the compilation order of definitions modules. The set of "difficult" compilation dependencies are hence limited and localized to definitions modules of the same scope and described in the same C/Mesa source module.

4) By point 1) there exists a well-defined order of compilation among interfaces possessing the same scope. The compilation order of such sets of interfaces was determined at design time, and, as a matter of policy, the interfaces were not often modified so as to change this ordering.

As an aside, it is clear that it is possible to build a tool which, given that a specified module has been changed, will examine the source modules of the system, determine which modules must be recompiled, and give the order of their recompilation. This is a *Consistent Compilation Tool.* A practical consistent compilation tool need not be omniscient, and it could occasionally cause a module to be compiled when this was not really necessary. Our attempts to build such a tool have been less than completely successful.

Consistent compilation and the design of associated tools is one of those topics which requires a separate paper for a complete treatment.

System Building

As already mentioned, the nodes of the Pilot tree are C/Mesa configuration descriptions. Associated with each is an object module built by binding all associated modules and configurations below the node in the Pilot tree. If a module changes, the system is rebound bottom up through the tree. First, the changed module is bound with its siblings in its parent configuration. Next, the parent is bound with its siblings in its parent's configuration, and so on.

Since the binding must be done on the developers personal computer and the object modules are stored in the file server, it is necessary to fetch from the file server the object modules involved in the binding and to store (after testing [see below]) the newly bound replacements back onto the file server.

The process of fetching (from the file server) the correct siblings for each level of binding is somewhat tedious and error prone. It was not automated except by individual developers using command files. Clearly this information should have been derived automatically from the Master List or from the hypothesized data base.

Each rebinding yields a new version of the object module. The Mesa Binder enforces *consistent binding* by ensuring that only one version of a module or sub-configuration is used either directly or indirectly in a bind. This situation has a number of similarities to the consistent compilation issue. The subtleties of consistent binding also merit treatment in a separate paper.

Integration and Testing

A key software engineering technique which we implemented for the Pilot project was that of *incremental integration.* This kept Pilot integrated and tested in a state which was no more than a few days behind the lead developers.

Each developer integrated and tested changes as he made them. Bugs arose incrementally and were usually restricted to the last set of changes; there was always a current working version of the system. This technique was particularly useful in the early stages of development, when the various teams were quite dependent on what the other teams were doing (i.e., they needed new functions as soon as they were implemented).

Substantial payoff was realized at the time of release. Final *systems integration* and *systems test* proved to be almost trivial; essentially no bugs showed up at this stage. (In many projects it is during this phase that project failure occurs [often with no prior warning]). We were also required to designate several system integrations as internal releases. This provided a continuing sequence of milestones by which progress could be measured.

Key to meeting this objective of incremental integration is the requirement to maintain consistency among the sources and objects in the working directory on the file server. In this case consistent means that the stored modules are consistently compiled and consistently bound and that the resultant *Pilot* object module has been system tested using regression-test programs also stored consistently in this same working directory.

When the Pilot object module had been constructed as described above, the test modules were fetched from the working directory and executed. Nothing was to be stored in the working directory until these tests had been passed. We referred to this whole process as *incremental integration.* (It is intended that the update performed in an incremental integration require only a small amount of work, [i.e., a few man-days]).

The steps in storing a change to Pilot onto the working directory were as follows: 1) test the change on a private version of Pilot in one's local environment. 2) fetch the latest object modules from the working directory, rebuild the system, and test again. 3) via the librarian, acquire sole

right to update the master copy. 4) again fetch the latest object modules, rebuild the system and test. 5) write the source and new object modules back onto the working directory. 6) relinquish sole right to update the master copy of the object modules via the librarian.

Steps 3-6 are, of course, necessary to resolve the "store race" which sometimes results from two developers performing incremental integrations in parallel. This procedure permits such parallel incremental integrations provided that they are independent updates and that the order in which they are performed matters not. Step 2) minimizes the time that the universal directory lock is held. Note that if independent and parallel incremental integrations are, in fact, taking place, the modules fetched at step 4) may very well be different than those fetched at step 2). Unless there is a subtle interaction error between the changes of the two concurrent incremental integrations, the test at step 4) will not fail.

While this procedure was effective in managing parallel incremental integrations, its implementation was not very satisfactory. The procedure was executed manually, introducing the potential for error. The fetching and storing were accomplished by command files derived from the Master List rather than from an integrated data base. This situation could be considerably improved by a tool flexing off the appropriate data base. While the overhead of our incremental integration procedure was considerable, the payoff more than justified it.

It should be pointed out that certain classes of changes could not be made as small increments to the current version of Pilot. For example, the changing of high-level interfaces usually had system wide repercussions. These changes were coordinated via internal releases (described below).

Releases

Internal Releases

Internal releases of Pilot were generated when major interface changes were required and also periodically to serve as milestones for the measurement of progress. Internal releases are also useful to assure the consistency of the source and object modules in the directory. In our environment it is possible (through human error) for the source and object modules to be inconsistent with each other due to the lack of unique version identification (e.g., a timestamp) in each source module. (Source modules may be updated and checked back in without being recompiled and rebound.) Ultimately, the only way to guarantee that the sources and objects are consistent is to recompile the source.

To make an internal release, the working directory was write-locked and the system was brought to a guaranteed consistent state by completely recompiling and rebuilding it from source files. The working directory was then tested and finally backed-up to an archive directory. This was all done by the Program Secretary using command files generated from the Master List. Any outstanding changes to high level interfaces were made and frozen several weeks prior to the internal release.

External Releases

An external release is accomplished simply by moving a completed internal release from the working directory to a public test directory. Substantial testing must take place and documentation must be created. At the completion of the testing period, the release should be moved from the public test directory to the proper public release directory.

The execution of this activity was another of the Program Secretary's duties.

Forking

Forking is defined to be the creation of a copy of a system followed by the development of that copy in a fashion inconsistent with the continuing development of the original. This usually means that there is at least one module in which changes must be made which are incompatible between the two branch systems of the fork. We forked at one point early in the development, and found it sufficiently unmanagable that we did not try it again. The extra complexity of maintaining two development paths and the problems of making parallel bug fixes were the major shortcomings of forking. The software engineering procedures described in this paper o not address the problems of forking.

File Management

All of these machinations create file and directory logistics problems. In addition to the main working directory, we also have a public test and a public release directory for the previous external release. Additionally, each external release and each internal release (four or five per external release) are captured on a structured archive directory.

By the end of the project, there were 600 current versions of files stored on just the working directory. This included almost 200 source files, their corresponding object files and symbols files (for the symbolic Mesa Debugger), and a number of other files, including about 150 associated with the test programs. With snapshots of past releases of the system on the archive directory, the actual number of online files approached 5000. The time spent keeping this data base up to date and backed up was very significant. The Master List and command files generated therefrom helped alleviate some of the logistics problems.

Conclusion

What is the upshot of all of this? In short, most of the development environment and control concepts which we used worked well. Of even more interest is the catalog of newly discovered issues which are the ones now constraining our performance. Our systems are never fast enough, particularly in switching from one major task to another. Many tasks which we perform manually cry out to be automated, to have their speed of execution improved but, more important, to have their accuracy increased. The automation of these tasks generally requires a much more integrated data base than is easily constructed in concert with our unintegrated tools.

Successes

What worked really well? The configuration and interface definition capabilities of the Mesa language, the C/Mesa configuration language, and the Mesa Binder worked spectacularly well in allowing us to divide, organize and control our development effort. Such facilities are clearly a must in any modern systems language and implementation.

The important notion of the scope of an interface and the concept of grading and controlling the volatility of each interface according to its scope gave the project the appropriate amount of stability at each organizational level. This stability in turn was one of the enabling factors for incremental integration.

The Program Secretary was clearly a vital post in this scheme. He was instrumental in maintaining the structure and consistency of the Master List, the directories, and the many command files. He was also the prime mover in the execution of both internal and external releases. We do have some vague suspicions, however, that the Program Secretary's main value was in carrying the integrated data base in his head, as we had no automated mechanism for doing so. Certainly the implementation of an effective and integrated data base (of which the Master List would be a part) would reduce his duties considerably.

The program librarian proved its worth in dealing with the problem of updating the working directory consistently. Since this tool was introduced slightly after the beginning of the Pilot project, its impact was clearly observable. It was an important facility in the implementation of the incremental integration technique.

Last, the incremental integration technique itself, despite its largely manual implementation, was quite successful, particularly from the point of view of avoiding a monolithic system integration and test just before a scheduled release.

Deficiencies

With respect to our development environment, the relative autonomy of each of our tools reflected itself in our inability to achieve an integrated data base which would control the tools in a consistent way. It also manifested itself in the relative slowness of the system in switching from one tool to another. Something as elementary as switching from the compiler to the editor requires a fraction of a minute. This slowness raises the cost of the update cycle and effectively imposes a minimum size on a change. The resulting increased batching of changes tends to make the process more error prone.

Maintaining and updating the librarian and Master List data bases was a tedious error-prone operation. In these cases the tools are in a relatively early stage, and not all of the improvements possible to the user interaction have yet been made.

A strong requirement for some additional tools has been established. The requirement for a Consistent Compilation Tool (for determining the modules to recompile and the order of recompilation) was proposed quite some time ago by members of our staff (not participants in the Pilot project), but the necessity for such a tool was not generally accepted at that time; the requirement for a Consistent Compilation Tool is now quite clear. As a result of the Pilot experience. The requirement for a Consistent Binding Tool has been also now established, whereas before the Pilot project this was not a particularly visible requirement. A third addition which would have a large positive impact is a tool for controlling and automating the incremental integration process.

The design and implementation of such tools constitutes a major effort in itself. Central to any solution is an integrated data base.

Acknowledgements

We are particularly indebted to our colleagues Hugh Lauer, Paul McJones, Steve Purcell, Dave Redell, Ed Satterthwaite, and John Wick, who both lived through, and furnished the raw data for, the experiences related in this paper and provided encouragement and constructive criticism for the text itself. We are also indebted to Jude Costello for a her many suggestions for improvement in this paper.

References

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

Lampson, B. W., "An Open Operating System for a Single User Machine," *to be published, Proceedings - Seventh Symposium on Operating System Principles,* (Dec., 1979)

Lampson, B. W. and Redell, D. D., "Experience with Processes and Monitors in Mesa," *to be published, Proceedings - Seventh Symposium on Operating System Principles,* (Dec., 1979)

Lauer, H.C. and Satterthwaite, E.H., "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering,* 1979

Metcalfe, R. M., and Boggs, D.R., "Ethernet: Distributed Packet Switching For Local Computer Networks," *Communications of the ACM* **19** 7 (July 1976), pp. 395-404

Mills, H. D., *Chief Programmer Teams: Techniques and Procedures,* IBM Internal Report, January 1970

Mitchell, J. G., "Mesa: A Designer's User Perspective", *Spring CompCon 78* (1978), pp. 36-39

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.

Parnas, D. L., "A Technique For Software Module Specification With Examples," *Communications of the ACM* **15** 5 (May 1972), pp. 330-336

Redell D. D., Dalal, Y. K., Horsley, T. H., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C., "Pilot: An operating system for a personal computer," *to be published, Proceedings - Seventh Symposium on Operating System Principles,* (Dec., 1979)

Boggs, D., Lampson, B. W., McCreight, E., Sproull, R., and Thacker, C. P., "Alto: A Personal Computer", Technical report *to be published,* Xerox Corporation, Palo Alto Research Center, Palo Alto, California, 1979.