THE MONADS ARCHITECTURE MOTIVATION AND IMPLEMENTATION

Dr John Rosenberg and Dr. David Abramson

Department of Computer Science,
Monash University, Wellington Road,
Clayton, Victoria, Australia,3168.

This paper explains the motivation behind the MONADS project, which was established in 1976 at Monash Univeristy. The paper shows that the decomposition of sytems into information-hiding modules has many advantages. The implementation of modules on conventional hardware is discussed and some serious problems associated with the underlying architecture of conventional hardware and operating systems are highlighted. The solution adopted by the MONADS system is described and future directions for the project are discussed.

Computing Review Categories and Subject Description:

C.1.3 (Processor Architectures) Capability Architectures, C.0 (General) Hardware/Software Interfaces, D.2.0 (Software Engineering) Protection Mechanisms.

Additional Keywords: Information-Hiding, Modules, Computer Architecture.

1. INTRODUCTION

The MONADS project was established in 1976 at Monash University, with the main aim of investigating improved techniques for designing and developing large and complex software systems. The first phase of the project was the design and implementation of an operating system for a locally modified minicomputer, known as MONADS I (Keedy, 1978; Georgiades, Richards & Keedy, 1978; Ramamohanarao & Keedy, 1978; Richards & Keedy, 1978; Rosenberg & Keedy, 1978; Rosenberg, 1979; Wallace, 1978). This gave us a test-bed on which to base further investigations.

The MONADS I project was followed by two other projects involving modifications to existing computers (MONADS II and II/2 [Abramson, 1982a&b]). The three projects convinced us that in order to provide a good environment for software engineering a computer with a radically different architecture was required. In 1980-82 a new machine called MONADS-III (Rosenberg, Rowe & Keedy, 1982; Keedy, Abramson, Rosenberg & Rowe, 1982; Rosenberg, 1982; Rowe, 1982) was designed, however it was never completed due to several staff resignations. In 1984 the MONADS project regrouped and began the design of MONADS-PC, a new workstation specifically oriented towards the support of good software engineering principles in hardware (Rosenberg & Abramson, 1985).

From the above and from most of the publications relating to the MONADS project, one may get the impression that the project is very much about hardware, computer architecture and operating systems. This is not the case. The central theme of the project has always been and still is investigation of software engineering techniques for development of large 'real world' applications. Our investigations to date have been chiefly based around building a computer architecture and operating system environment to provide appropriate support for software engineering, and this is why the external appearance of the project has been rather technical.

This paper is an attempt to explain the motivation behind the MONADS architecture. The paper begins by describing a simplified design for a commercial application. It is shown, via this example, that the decomposition of systems into information-hiding modules has many advantages and the implementation of such modules on conventional hardware is discussed. Some serious problems associated with the underlying arcbitecture of conventional hardware and operating systems are highlighted. The solution adopted by the MONADS system is described and future directions for the project are discussed.

2. A SIMPLE EXAMPLE

The example that will be used throughout this paper is that of a simple accounting system. The system is to support a number of accounts, each of which has an account number. Transactions may be entered on each account and each transaction has a transaction number, transaction type, an effective date and an amount.

The following end-user operations are to be provided:

1. Enter Transactions
2. Display Account Balance
3. Delete Transactions
4. Print Trial Balance

Enter transactions allows new transactions to be added for a specified account. Display account balance will compute and display the balance of an

account as at a specified date. Delete transactions will remove an existing transaction. Print Trial Balance will print a report listing the balances for each account up to a specified date.

2.1. System Design

The simplest design for the system has only one file. The record structure using COBOL syntax is shown in Figure 1. All of the individual transactions are maintained on file rather than maintaining a current balance, allowing the balance of an account to be calculated up to an arbitrary date.

It should be noted that the tasks have different access requirements on the file and these are summarised below.

Task	Access Required
Enter Transactions	Write
Display Account Balance	Read
Delete Transactions	Read,Delete
Print Trial Balance	Read

The algorithms for enter and delete transactions are obvious. Both display account balance and print trial balance must compute the balance of an account. This is achieved by reading all transactions for a particular account up to the specified date and summing the amount field of these records.

A problem with this simple design is that the display and print operations would be quite slow, because the balance calculation requires the entire account history to be read. A possible improvement is the addition of a second file with the record structure shown in Figure 2.

```
01  ACCOUNT-RECORD.
    03  PRIMARY-KEY.
        05  ACCOUNT-NUMBER          PIC 9(6).
        05  TRANSACTION-NUMBER      PIC 9(6).
    03  TRANSACTION-TYPE            PIC X.
        88  CREDIT-TRANSACTION              VALUE "C".
        88  DEBIT-TRANSACTION               VALUE "D".
    03  EFFECTIVE-DATE              PIC 9(6).
    03  AMOUNT                      PIC 9(7)V99.
```

Figure 1 - COBOL Record Structure for Accounts file

```
01  MONTHLY-BALANCE-RECORD.
    03  PRIMARY-KEY.
        05  ACCOUNT-NUMBER          PIC 9(6).
        05  YEAR-MONTH              PIC 9(4).
    03  AMOUNT                      PIC 9(7)V99.
```

Figure 2 - COBOL Record Structure for Monthly-Balances file

To take account of this improvement the system can be modified so that a monthly-balance-record exists for every month and account in which there is at least one transaction. When a transaction is inserted into the accounts file the amount is added to the amount in the appropriate monthly-balance-record. If a record does not already exist for the account/month combination a new record is created. When a transaction is deleted the amount is subtracted from the monthly-balance-record. Thus an account balance as at date DD/MM/YY may now be computed by summing the monthly-balance records for the account up to the month before MM/YY and adding to this sub-total the sum of all records on the accounts file for the specified account for dates from 01/MM/YY to DD/MM/YY inclusive. Such an algorithm is clearly more efficient because of the reduction in the number of file accesses.

2.2. System Implementation

There are a number of different software engineering techniques for building the system described above. The most primitive approach is to construct one large program to perform all of the the required end-user operations. This has long been recognized as disastrous because the complexity of such a program makes construction and maintenance error-prone and difficult.

A more common approach would be to provide each operation as a separate program (possibly driven by a menu system). Each program is then of a reasonable size and this simplifies the programming task. A major problem with this scheme is that it does not recognize that there are some basic semantic operations on the file which must be performed by several programs. For example the calculation of an account balance is performed in both display-account-balance and print-trial-balance. If these programs are written by different people then they will each have to derive and code the same algorithm (possibly with different results!).

A more serious problem can be illustrated by considering the modification suggested earlier. The addition of a monthly-balances file will require modification to all of the programs. Whilst this may not appear to be a significant problem in our simple example, in a 'real' system many programs may need to be changed. It is quite likely that at least one of these changes will be incorrect and/or one of the programs will be inadvertently forgotten, leading to an unreliable system.

This problem has been recognized for many years and one of the most widely accepted solutions is based on the use of information-hiding modules (Parnas, 1971&72; Keedy, 1978; Keedy & Rosenberg, 1981; Rosenberg & Keedy, 1981a). With this technique key data structures are encapsulated by a collection of code procedures, each of which provides an interface for accessing the data. Such modules are usually known as abstract data types, information-hiding modules or type managers. Any program requiring access to the data can only do so via the procedures of the module. As well as the obvious 'put-data' and 'get-data' operations it is also possible to provide procedures for higher-level operations. The major advantage of this scheme is that the information about the internal structure of the data is completely contained within the one module.

The use of modules can be illustrated by the previous example. The accounts file can be encapsulated in an accounts-module (Figure 3) which provides procedures for the following operations:

```
read_transaction
insert_transaction
delete_transaction
compute_account_balance
```

The four programs would then make use of the appropriate procedures in order to perform the required operation. All detail relating to the structure of the database has been 'hidden' within the module. If we now consider the addition of the monthly-balances file, only the accounts-module need be modified, thus reducing the chance of new errors being introduced.

3. IMPLEMENTATION OF MODULES ON CONVENTIONAL SYSTEMS

Information-hiding modules are used by many software developers to construct large systems. This is in spite of the fact that most existing machines and software development environments provide virtually no support for modules. It is possible, however, to use facilities provided for other purposes to implement data abstractions.

3.1. Separate Compilation as a means of Implementing Modules

Most existing high-level languages have some form of subroutine facility which may be coupled with separate compilation to implement a module-like structure. For example, using the COBOL-74 LINKAGE-SECTION it is possible to separately compile a module containing the FD for the accounts-file. This module could then contain sections of code to implement the interface functions. There are two main problems with this approach. First, all interface functions have to share a common linkage-section, despite the fact that the parameters may be quite different. Second, all interface functions will enter the code at the one point and an extra parameter must examined to determine the function to be performed. Some of these problems are alleviated by COBOL-80, however, there is still no explicit support for information-hiding modules.

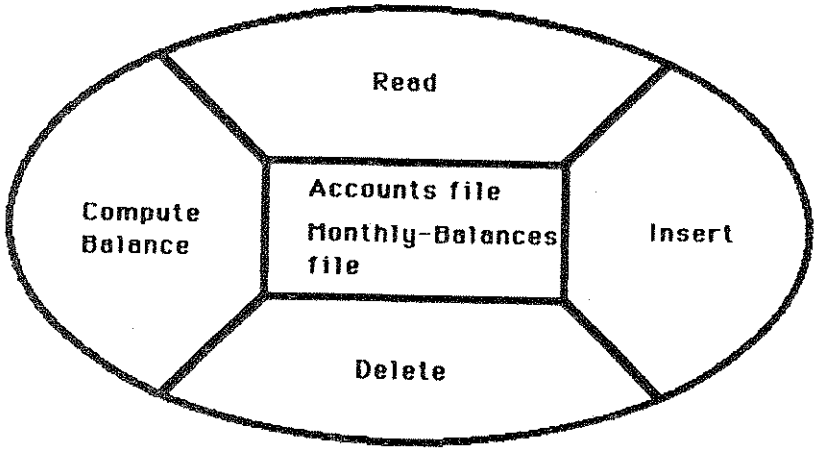

Figure 3 - The Accounts Module

A module-like structure can be achieved with Fortran by including all data for a module in a named common block and only declaring this common block in the procedures of the module. This approach relies on the programmer setting up the correct structure initially and maintaining this structure during later modifications. There is nothing to stop another module including the common block declaration and thereby gaining access to the data.

Some of the newer languages (e.g. MODULA 2, ADA) do support the decomposition of programs into modules, but usually such modules are merged into one large program before run-time. The problems with this will be discussed later.

The design of module-based languages is an interesting problem which is outside of the scope of this paper. Our initial concern is to provide architectural and operating system support for modules. However, other members of the MONADS project team have completed the design of a new language (LEIBNIZ) which is specifically designed to complement the MONADS architecture.

3.2. What is a Module?

In order to consider support for modules it is necessary to first examine what constitutes a module. An information-hiding module basically consists of code and data. Ideally the data should only be accessed by the code of the associated module. This has two main advantages. First, knowledge of the structure of the data is confined to the module, thus simplifying system maintenance. Second, it guarantees that the data will not be corrupted by any other module.

It is possible (and desirable) to view many computing structures as modules (Richards, 1983). The most common use of modules is as data abstractions in programs (e.g. a stack or a queue)(Hoare, 1972; Liskov & Zilles, 1974; Wulf, London & Shaw, 1976). A less obvious application is files (Keedy & Richards, 1982). A file may be viewed as a module whose data is the records of the file and whose procedures provide semantic operations (e.g. compute-account-balance). Another example of the use of modules is a subroutine library. This is a module with no permanent data (other than constants), but simply a collection of procedures. Thus, modules may be seen as a unifying factor in the design of systems.

Ideally a module consists of four different types of data in addition to the code procedures. The first type of data is code-related data. This is data which is created at compile time and is permanently bound to the code of the module (cf const data in Pascal). Code-related data is thus shared by all users of the code procedures. The second type of data is local data. Local data is created on entry to a procedure and is used for temporary storage within the procedure. Local data is destroyed on exit from the procedure (cf Pascal local data). Each user of a module must have its own local data.

The third type of data is called permanent data. This is data associated with a module, which exists until the module is deleted (cf conventional data files). There may be several versions of such data, all associated with same the code procedures. These versions are usually called instances of the module type and may be likened to several data files having the same structure (COBOL FD), with the difference that the data is permanently bound to the access routines. Instances of modules may be concurrently shared between users in the same sense as files may be concurrently open by several users.

The fourth type of data is called retained data. This is data associated with a particular invocation (by a user program) of an instance of a module. Retained data is created on the first call to a module (cf opening a file) and persists between calls to the module. Retained data may be destroyed explicitly and is automatically removed when the program terminates (cf closing a file).

The above types of data can be illustrated by the accounts-module described earlier. The code-related data would be constants used by the procedures of the module (e.g. record-size). The local data would include temporary variables used in the modules procedures (e.g. a running total used to calculate the current balance). The permanent data would be the actual records of the file as well as any data required to access the file (e.g. index records, end-of-file pointer). The retained data could be used to remember the type and key of the last record access. For example, it may be sensible to insist that a record is read before being deleted. Thus delete would require no parameters and could check that the last operation was a read. It should be noted that retained data is not the same as permanent data. All concurrent users of the module instance would want to share the permanent data, but have different retained data.

3.3. Support on Conventional Systems

3.3.1. Code

Most existing machines provide no explicit support for the separation of the code of different modules. Instead, all modules of a program are linked together to form one large executable image, which is then loaded into a single contiguous address space. Because such an image is often very large many systems allow copies to be physically shared in main memory. In most cases sharing is achieved via page translation hardware which forces shared code to be aligned to a page boundary (a notable exception is the B6700 (Organick, 1973) which supports a segmented memory). This scheme is acceptable if an entire executable image is shared, however, problems arise when such an image consists of many (potentially shared) modules.

First, all code in the modules must be position-independent because the module may appear in different parts of the address space of different images. Second, each module must be aligned to a page boundary so that the sharing may be implemented, creating serious memory fragmentation problems. The third, and most serious, problem is that an entire module must be included in the address space of a program only requiring a limited set of the available procedures. This leaves the door open for the program (either accidently or maliciously) to call one of the other procedures. For example, an error in the display-account-balance program above, could result in the delete procedure of the accounts-module being called. This could not have happened if only the required procedure (compute-account-balance) had been included in the executable image.

Another problem with statically merging all of the modules required into a single program is that it is not always possible to know the names of all of the modules to be used. For example, if files are implemented as modules then it would not be possible to write a program to print an arbitrary file.

3.3.2. Code-related Data

Most systems provide some support for code-related data. It is usually stored in the same area of the address space as the code and consequently is

read-only. However, in the general case, code-related data should be modifiable and any changes should be permanent. An example of the use of modifiable code-related data would be a count of the number of calls to a particular procedure for fine-tuning of a system. Note that this count cannot be placed in permanent data as it must be shared amongst all instances of the module. Existing systems do not support this type of code-related data.

3.3.3. Local Data

The conventional implementation of local data uses a procedural stack. On each procedure call a new stack frame for local variables is created at the top of the stack, and on return the frame is removed. Each process has its own stack and thus processes are protected from each other. However, there is usually no protection within the process and any procedure has access to all of the previous stack frames, including those created on entry to procedures of other modules. Consequently, it is possible to corrupt local data or read sensitive information belonging to procedures both in the current module and any other modules in the call history. Thus, in spite of constructing a system as information-hiding modules, there is no guarantee that errors will be confined to an offending module.

3.3.4. Permanent Data

The only support for permanent data on conventional systems is provided by the file system. A typical file system interface takes the form of OPEN, READ_BLOCK, WRITE_BLOCK and CLOSE (ignoring access methods), in contrast to the interface to other types of data which is simply an address within an instruction. Thus a procedure must explicitly be written either to access permanent data (i.e. a file) or local data, making it very difficult to share common algorithms for similar data structures. For example, hash tables are used both as file structures and temporary data structures, however, different routines would be required for each of these applications.

Another problem with using files for permanent data is that the overhead of calling the file system is incurred on each access. For small frequently used files (e.g. a file containing the company name to be displayed on all forms) this is a considerable cost.

If permanent data is implemented using a file system it is not possible to properly implement modules because the data is not bound to the appropriate code. Further, the only access protection modes provided by the file system are relatively simple (e.g. read, write, execute, delete) and take no account of the basic semantic operations on the data. In addition access modes are usually allocated on a user basis, thus any program executed by a particular user inherits the user's file access rights. A user allowed to run all of the tasks in the earlier example would therefore require read/write access to the accounts file, allowing a 'read-only' program such as compute-account-balance to accidently write to the file.

A final problem with implementing permanent data as files relates to synchronisation. Permanent data is often shared by several concurrent programs and thus these programs must synchronise with respect to their access to the data. Most file systems allow synchronisation by providing a record locking facility. In addition, the operating system usually provides an inter-task synchronisation and communication facility. However, these are usually implemented quite separately from each other and have quite different interfaces, despite the fact that they are doing basically the

same job. This duplication of mechanism is again symptomatic of the fact that permanent data is treated differently from all other types of data.

3.3.5. Retained Data

Retained data is usually 'simulated' by grouping together the retained data for all modules and creating space for this data on entry to the program. This implementation suffers all of the protection problems of local data. Also, because the size of the retained data is calculated statically at compile/link time, the solution does not cope with dynamic invocation of modules and multiple invocations of the same module (e.g. opening a file twice).

3.4. Summary

The simple example described earlier clearly demonstrates that the decomposition of programs into information-hiding modules has many advantages. It simplifies the construction, testing, debugging and maintenance of software systems by limiting the knowledge of implementation details and providing an abstract view of the underlying data. It is also clear that existing computer systems do not support the types of data required by information-hiding modules.

The main aim of the MONADS system is to provide an environment which will efficiently execute programs decomposed into information-hiding modules by supporting the appropriate data model at the hardware level.

4. IMPLEMENTATION OF MODULES IN MONADS

4.1. Uniform Virtual Memory

One of the major problems experienced with modules on conventional systems is that not all data types are treated uniformly. In particular, permanent data is maintained and accessed in an entirely different manner from the other data types. In the MONADS architecture all data is contained in one large uniform paged virtual memory, which encompasses all disk storage and main memory. Thus all of the different types of data are addressed using the same mechanism.

The virtual memory is accessed by very large addresses (60 bits on MONADS-PC), each of which identifies an individual byte. The virtual memory is divided into fixed size pages (4096 bytes on MONADS-PC) which are swapped between disk and main memory by system software in the same manner as on conventional virtual memory machines. To simplify memory management, the virtual memory is divided up into 2**32 regions (called address spaces) each of 2**28 bytes. The use of conventional address translation hardware is inappropriate for such large addresses and the MONADS architecture employs considerable hardware support to implement the virtual memory (Abramson, 1981; Rosenberg & Keedy, 1981).

Each address space is further divided into areas called segments, each of which is used to hold a logical entity (e.g. an array, a procedure). A segment is defined by a starting virtual address and a length. Segments may be of arbitrary size and need not be aligned to page boundaries (Keedy, 1980). This structure is illustrated in Figure 4.

When a new segment is created a previously unused area of virtual memory is allocated to the segment and when a segment is destroyed this addressing region (but not the physical memory or disk space) is permanently lost. Thus a given virtual address will uniquely identify a particular byte

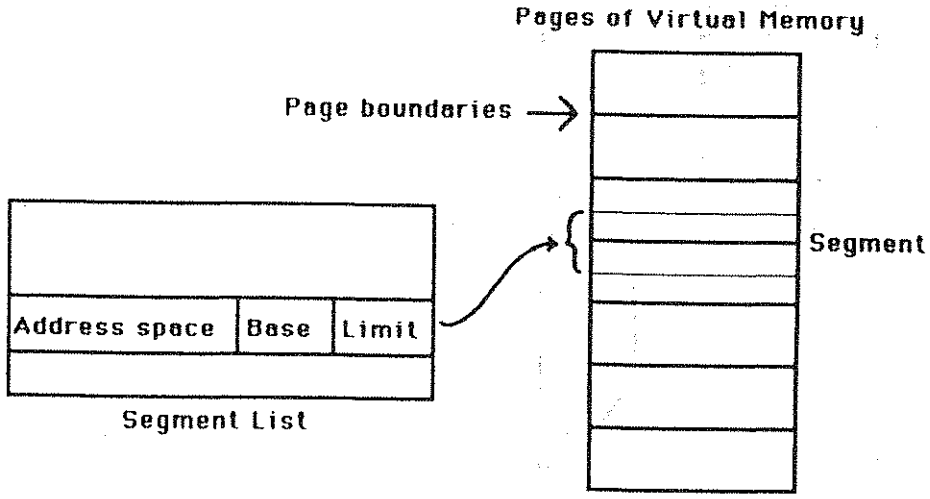

Figure 4 - Segments mapped onto Pages

of a particular segment and will never be used again to represent a different byte. Once the segment has been destroyed the use of its virtual address will cause an error.

Such unique and non-reusable addresses are called capabilities (Dennis & Van Horn, 1966). Capabilities solve many of the problems related to sharing and protection of information because they provide a unique name for an object which can be used by all users of the object (Fabry, 1974). By restricting the set of capabilities available to a program it is possible to limit the data that the program can access. Such a restriction can be implemented by not allowing programs to directly construct capabilities. Capabilities are used in the MONADS architecture to control the domain of access of modules.

4.2. Module Structure

Each process on the MONADS system has associated with it a number of base registers. These registers point to lists of capabilities (called segment lists) for segments that the process may access. The bases and segment lists may not be directly modified by user programs and are maintained by the hardware. There is at least one base associated with each of the data types required by a module as well as a special base used for passing parameters. There are several local bases to support block-structured languages providing a lexical level display similar to that on the B6700 (Organick, 1973). The only method of addressing memory is by specifying a base, segment number and offset within segment. (For efficiency a set of 16 capability registers are provided. A segment list entry may be loaded into a capability register and programs actually use addresses of the form <capability_register_number,offset>.) Thus the current set of bases define the addressing environment of the process (Figure 5).

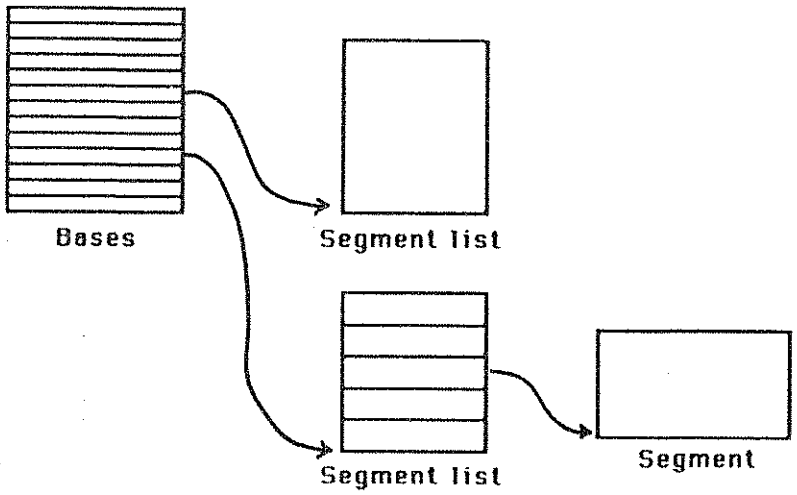

Figure 5 - A process addressing environment

The inter-module call mechanism is the key to the protection scheme used on the MONADS architecture. The call instruction not only transfers control to the new module, but also loads the bases to define the environment for the module. The permanent, constant and retained bases are pointed to the appropriate segment lists for this module, a local data segment list and associated segments are created (on a stack) and the local base for the current lexical level is pointed at this new list. In addition a parameter base is pointed to a segment list of parameters passed to the called module. Thus the called module only has access to its own data and parameters passed to it by the calling module. Any attempt to access other data (e.g. by generating an offset outside the bounds of a segment) will result in a run-time error. The return instruction removes the local segments, reinstates the calling module's environment and returns control to the caller (Gehringer, 1982; Rosenberg, 1984).

Each module has two standard procedures called CREATE and OPEN. The CREATE procedure makes a new instance of the module, creates the permanent data segment list and data segments and binds these to the code. The OPEN procedure is called on the first invocation of a module by a program. The OPEN procedure is divided into two halves. The first half is executed when OPEN is called and creates the retained data segment list and segments. The second half is automatically called when the program which called OPEN terminates and removes the retained segments.

4.3. Naming of Modules

The above sections have described how the environment of a module is dynamically defined by CALL and RETURN. On a call a program must identify which procedure and module instance is to be entered. Because the unique identity of a module instance is defined by its permanent data, it may be identified by the address space number of the permanent data. Each procedure within a module is given a number and thus the call instruction requires both the permanent data address space number and a procedure number. This is achieved by providing a module capability, which contains the permanent data address space number and a list of the procedures which may be called. Thus,

in order to call a particular procedure, a program must have a module capability including the required procedure number. This facility can be used to give a program access to only a limited set of procedures of a given module.

Module capabilities are stored in segments in the same way as other data, but the segment list entry indicates that the segment contains module capabilities and the hardware guarantees that these will not be modified directly by a program. Because module capabilities are stored in normal segments it is possible to have a symbolic names manager for translating symbolic module names to capabilities. This is equivalent to a conventional file directory.

It should be noted that it is not necessary to have a static linking process. Any module may call any other module provided it has an appropriate module capability which can be obtained dynamically. Thus it is easy to write general purpose programs (e.g. a program to print a file), which are simply passed a module capability as a parameter.

5. CONCLUSIONS

The central theme behind the MONADS project is the concept of the module. The decomposition of software systems into information-hiding modules has many advantages as has been demonstrated in this paper. It has also been shown that conventional hardware and software does not properly support modules. The goal of the MONADS project is to construct a computer system which is specifically designed to support modules.

Some progress has been made towards that goal. A new 32 bit machine, MONADS-PC, has been designed and constructed and is now operational. The machine is not based on any commercial microprocessor, but rather is a microcoded machine built out of standard MSI and LSI TTL components (Abramson, 1984). At present an operating system is being developed for MONADS-PC. When this has been completed, the project will continue with investigation of of the effects of such an architecture on other applications, in particular databases. Research is also continuing in the hardware design techniques, especially the application of VLSI technology to the MONADS architecture (Abramson & Rosenberg, 1985). Another area of interest concerns the implementation of a local area network of MONADS processors (Abramson & Keedy, 1985).

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contribution to this work from Professor J.L. Keedy, the instigator of the MONADS project.

REFERENCES

ABRAMSON, D.A. (1981): "Hardware Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane, pp 1-13.

ABRAMSON, D.A. (1982a): "Hardware for Capability Based Addressing", Proc. 9th. Australian Computer Conference, pp 101-115, Hobart.

ABRAMSON, D.A. (1982b): "Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory", Ph.D. Thesis, Department of Computer Science, Monash University.

ABRAMSON, D.A. (1984): "MONADS-PC Micro Architecture Manual", MONADS-PC Technical Report 2, Monash University.

ABRAMSON, D.A. & KEEDY, J. (1985): "Implementing a Large Virtual Memory in a Distributed Computing System", Proc. 18th Hawaii International Conference on System Sciences, Hawaii.

ABRAMSON, D.A. & ROSENBERG, J. (1985): "Supporting a Capability-Based Architecture with Silicon", submitted for publication.

DENNIS, J.B. & VAN HORN, E.C. (1966): "Programming Semantics for Multiprogrammed Computations", Comm. ACM, 9, 3, pp 143-155.

FABRY, R.S. (1974): "Capability Based Addressing", Comm. ACM, 17, 7, pp 403-412.

GEHRINGER, E.F. (1982): "MONADS: A Computer Architecture to Support Software Engineering", MONADS Report No. 13, Monash University.

GEORGIADES, A., RICHARDS, I. & KEEDY, J.L. (1978): "A File System for the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 547-558, Canberra.

HOARE, C.A.R. (1972): "Proof of Correctness of Data Representations", Acta Informatica, vol. 1, pp 271-281.

KEEDY, J.L. (1978): "The MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 903-910, Canberra.

KEEDY, J.L. (1980): "Paging and Small Segments: A Memory Management Model", Proc. IFIP-80, Melbourne, pp 337-342.

KEEDY, J.L., ABRAMSON, D.A., ROSENBERG, J. & ROWE, D.M. (1982): "A Comparison of the MONADS II and III Computer Systems", Proc. 9th. Australian Computer Conference, pp 581-587, Hobart.

KEEDY, J.L. & RICHARDS, I. (1982): "A Software Engineering View of Files", ACJ, 14, 2, pp 56-61.

KEEDY, J.L. & ROSENBERG, J. (1981): "Information-Hiding - A Key to Successful Software Engineering", Proc. Conference on Computers in Engineering, Melbourne, Institution of Engineers, Publication No. 8178, pp 1-5.

LISKOV, B. & ZILLES, S. (1974): "Programming with Abstract Data Types", Proc. of Conference on Very High Level Languages, ACM SIGPLAN Notices, 9, 4, pp 50-59.

ORGANICK, E.I. (1973): "Computer Systems Organisation, the B5700/6700 Series", Academic Press, New York.

PARNAS, D.L. (1971): "Information Distribution Aspects of Design Methodology", Proc. IFIP Congress, Booklet TA-3, pp 26-30.

PARNAS, D.L. (1972): "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, 15, 12, pp 1053-1058.

RAMAMOHANARAO, K. & KEEDY, J.L. (1978): "Job Management in the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 1476-1488, Canberra.

RICHARDS, I (1983): "Unifying Software Objects", Proceedings of 6th Australian Computer Science Conference, Sydney, pp. 334-345.

RICHARDS, I. & KEEDY, J.L. (1978): "Subsystem Management in the MONADS Operating System", Proc. 8th. Australian Computer Conference, pp 1520-1529, Canberra.

ROSENBERG, J. (1979): "The Concept of a Hardware Kernel and its Implementation on a Minicomputer", Ph.D. Thesis, Department of Computer Science, Monash University.

ROSENBERG, J. (1982): "The MONADS Series III Instruction Set", Proc. 9th. Australian Computer Conference, pp 704-718, Hobart.

ROSENBERG, J. (1984): "MONADS-PC System Management Instructions", MONADS-PC Technical Report 5, Monash University.

ROSENBERG, J. & KEEDY, J.L. (1978): "The MONADS Hardware Kernel", Proc. 8th. Australian Computer Conference, pp 1542-1552, Canberra.

ROSENBERG, J. & KEEDY, J.L. (1981a): "Information-Hiding - A Case Study", Proc. Conference on Computers in Engineering, Melbourne, Institution of Engineers, Publication No. 8178, pp 6-9.

ROSENBERG, J. & KEEDY, J.L. (1981b): "Software Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane, pp 173-181.

ROSENBERG, J., ROWE, D.M. & KEEDY, J.L. (1982): "An Overview of the MONADS Series III Architecture", Proc. 5th. Australian Computer Science Conference, pp 58-67, Perth.

ROSENBERG, J. & ABRAMSON, D.A. (1985): "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", Proc. 18th Hawaii International Conference on System Sciences, pp 222-231, Hawaii.

ROWE, D.M. (1982): "MONADS III Inter-unit Communication", Proc. 9th. Australian Computer Conference, pp 719-729, Hobart.

WALLACE, C.S. (1978): "Memory and Addressing Extensions to a HP2100A", Proc. 8th. Australian Computer Conference, pp 1796-1811, Canberra.

WULF, W.A., LONDON, R.L. & SHAW, M. (1976): "Abstraction and Verification in Alphard: Introduction to Language and Methodology", Department of Computer Science, Carnegie-Mellon University.

Sponsored by
Ultimate computer.

September 10-13, 1985

Royal Exhibition Building, Melbourne, Australia.

PROCEEDINGS VOL. 1