

PROJECT ADMINISTRATIVE DATA SHEET

ORIGINAL REVISION NO. _____

Project No. G-36-621 STRI/GTRX DATE 10/5/84

Project Director: R.J. LeBlanc School/DEPT ICS

Sponsor: Maryland Procurement Office
9800 Savage Road Fort George G. Meade Md. 20755

Type Agreement: Contract MDA904-84-C-6035

Award Period: From 9/11/84 To 9/30/85 (Performance) 10/10/85 (Reports)

Sponsor Amount: This Change Total to Date
Estimated: \$ _____ \$ _____
Funded: \$ 49,960 \$ 49,960

Cost Sharing Amount: \$ _____ Cost Sharing No: _____

Title: Research on Reliable Distributed Computing

ADMINISTRATIVE DATA

OCA Contact Ralph Grede X4820
1) Sponsor Technical Contact: Director National Security Agency
Chief, Central Security Service
Attn: (See block 14, DD form 1423)
Fort George G. Meade, Md. 20755-6000
M/F Contract MDA904-84-C-6035
2) Sponsor Admin/Contractual Matters:
Mr. Lawrence C. Tarbell, Jr.
Mr. Kurt J. Schmucker
Maryland Procurement Office
9800 Savage Road
Fort George G. Meade, Md. 20755

Defense Priority Rating: D0-A7 Military Security Classification: none
(or) Company/Industrial Proprietary: n/a

RESTRICTIONS

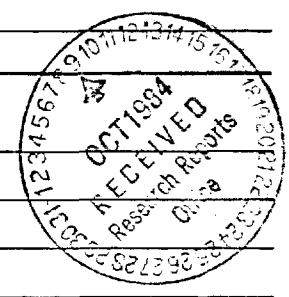
See Attached Contract Supplemental Information Sheet for Additional Requirements.
Travel: Foreign travel must have prior approval - Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of \$500 or 125% of approved proposal budget category.
Equipment: Title vests with GIT - However, none proposed.

COMMENTS:

GSA Supply Sources - Incorporated by Reference
Travel cost NTE - \$2,500
Surplus cost NTE - \$1,500
Disclosure restrictions - See Section H.1
Sub Contracting must be approved.

COPIES TO:

Project Director Procurement/EES Supply Services GTRI
Research Administrative Network Research Security Services Library
Research Property Management Reports Coordinator (OCA) Project File
Accounting Research Communications (2) Other Newton-Legal/Jones



SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

Date 2/3/86

Project No. G-36-621 (R-5840-0A0) School ~~XXX~~ ICS

Includes Subproject No.(s) None Indicated

Project Director(s) R. J. LeBlanc GTRI / ~~XXX~~

Sponsor Maryland Procurement Office

Title Research on Reliable Distributed Computing

Effective Completion Date: 9/30/85 (Performance) 11/12/85 (Reports)

Grant/Contract Closeout Actions Remaining:

- None
- Final Invoice or Final Fiscal Report
- Closing Documents
- Final Report of Inventions
- Govt. Property Inventory & Related Certificate
- Classified Material Certificate
- Other _____

Continues Project No. None Indicated Continued by Project No. None Indicated

COPIES TO:

Project Director
 Research Administrative Network
 Research Property Management
 Accounting
 Procurement/EES Supply Services
 Research Security Services
 Reports Coordination (OCA)
 Legal Services

Library
 GTRI
 Research Communications (2)
 Project File
 Other A. Jones; M. Heyser; R. Embry

QUARTERLY FUNDS EXPENDITURE REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT # MDA 904-84-C-6035
REPORTING PERIOD: 11 SEP 84 -- 31 DEC 84

Column A				Column B	Column C	Column D			Column E	Column F
ORIGINAL PROPOSAL				Latest Accepted Revised Proposal	Reporting Quarter Expenditures	Cumulative Expenditures to Date			Cost to Complete Estimate	Latest Cost Estimate
Type	Number of Hours	Hourly Rate	Dollar Total		Total Man Hours	Dollar Value	Pct. Dollar Value			
1. Direct Labor										
PI	350	23.77	\$8320.00	\$2773.32	117	\$2773.32	33.3%	\$5546.68	\$8320.00	
GRA	1300	11.41	\$14833.00	\$2971.98	260	\$2971.98	20.0%	\$11861.02	\$14833.00	
Clerical	175	6.74	\$1180.00	\$0.00	0	\$0.00	0.0%	\$1180.00	\$1180.00	
Total Direct Labor			\$24333.00	\$5745.30	377	\$5745.30	23.6%	\$18587.70	\$24333.00	
Burden @ 24.6% (Excluding GRA Labor)			\$2337.00	\$682.24		\$682.24	29.2%	\$1654.76	\$2337.00	
Total Direct Labor and Burden			\$26670.00			\$6427.54	24.1%	\$20242.46	\$26670.00	
2. TRAVEL EXPENSE			\$2500.00	\$0.00		\$0.00	0.0%	\$2500.00	\$2500.00	
3. GENERAL & ADMINISTRATIVE EXPENSE			\$1500.00	\$57.00		\$57.00	3.8%	\$1443.00	\$1500.00	
4. COMPUTING CHARGES			\$1500.00	\$375.00		\$375.00	25.0%	\$1125.00	\$1500.00	
TOTAL DIRECT COSTS			\$32170.00	\$6859.54		\$6859.54	21.3%	\$25310.46	\$32170.00	
5. INDIRECT COSTS @ 55.3%			\$17790.00	\$3793.32		\$3793.32	21.3%	\$13996.68	\$17790.00	
TOTAL CONTRACT PRICE			\$49960.00						\$49960.00	
TOTAL COMMITMENTS AND EXPENDITURES				\$10652.86		\$10652.86	21.3%			

FIGURE 1

FUNDS EXPENDITURE GRAPH

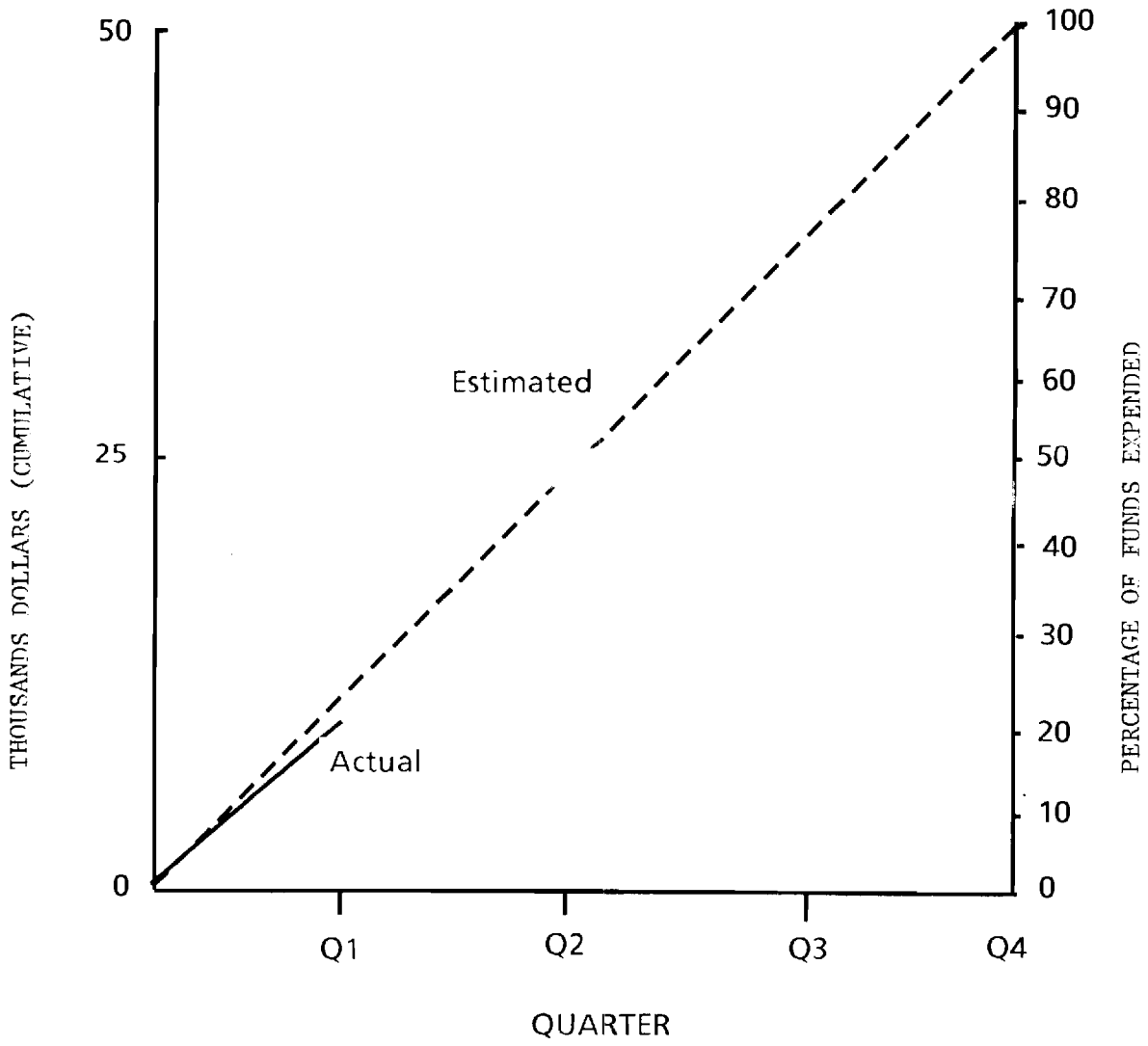


Figure 2

Column A				Column B	Column C	Column D			Column E	Column F
ORIGINAL PROPOSAL				Latest Accepted Revised Proposal	Reporting Quarter Expenditures	Cumulative Expenditures to Date			Cost to Complete Estimate	Latest Cost Estimate
						Total Man Hours	Dollar Value	Pct. Dollar Value		
1. Direct Labor										
Type	Number of Hours	Hourly Rate	Dollar Total							
PI	350	23.77	\$8320.00	\$2781.26		234	\$5562.51	66.9%	\$2757.49	\$8320.00
GRA	1300	11.41	\$14833.00	\$5066.04		704	\$8032.64	54.2%	\$6800.36	\$14833.00
Clerical	175	6.74	\$1180.00	\$0.00		0	\$0.00	0.0%	\$1180.00	\$1180.00
Total Direct Labor			\$24333.00	\$7847.30		938	\$13595.15	55.9%	\$10737.85	\$24333.00
Burden @ 24.6% (Excluding GRA Labor)			\$2337.00	\$684.19			\$1368.38	58.6%	\$968.62	\$2337.00
Total Direct Labor and Burden			\$26670.00	\$8531.49			\$14963.53	56.1%	\$11706.47	\$26670.00
2. TRAVEL EXPENSE			\$2500.00	\$795.00			\$795.00	31.8%	\$1705.00	\$2500.00
3. GENERAL & ADMINISTRATIVE EXPENSE			\$1500.00	\$65.00			\$132.00	8.8%	\$1368.00	\$1500.00
4. COMPUTING CHARGES			\$1500.00	\$375.00			\$750.00	50.0%	\$750.00	\$1500.00
TOTAL DIRECT COSTS			\$32170.00	\$9766.49			\$16640.53	51.7%	\$15529.47	\$32170.00
5. INDIRECT COSTS @ 55.3%			\$17790.00	\$5400.86			\$9202.21	51.7%	\$8587.79	\$17790.00
TOTAL CONTRACT PRICE			\$49960.00							\$49960.00
TOTAL COMMITMENTS AND EXPENDITURES				\$15167.35			\$25842.74	51.7%		

FUNDS EXPENDITURE GRAPH

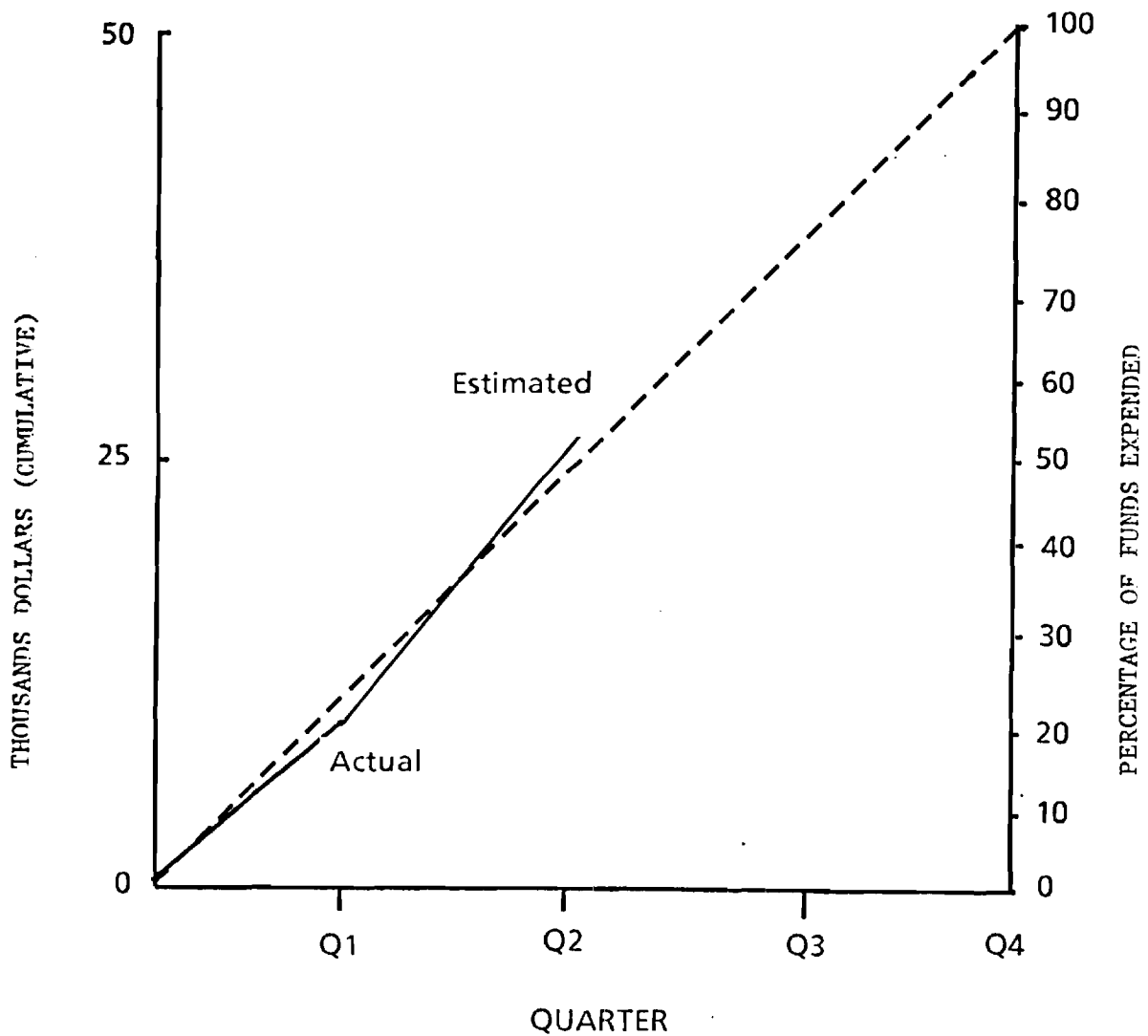


Figure 2

Prepared July 16, 1985
 Contract No. MDA 904-84-C-6035
 Contractor Georgia Tech Research Corp.

Summary/Work Package Title
 Research on Reliable Distributed
 Computing
 Report Month 4/1/85 - 6/31/85

FUNDS EXPENDITURE REPORT

Column A	Column B	Column C	Column D	Column E	Column F
-----	-----	-----	-----	-----	-----
ORIGINAL PROPOSAL	Latest Accepted Revised Proposal	Reporting Quarter Expenditures	Cumulative Expenditures to Date Total Man Hours Dollar Value Pct. Dollar Value	Cost to Complete Estimate	Latest Cost Estimate
Direct Labor					
Type	Number of Hours	Hourly Rate	Dollar Total		
PI	350	23.77	\$8320.00	\$1759.09	308 \$7321.60 88.0% \$998.40 \$8320.00
GRA	1300	11.41	\$14833.00	\$5066.04	1148 \$13098.68 88.3% \$1734.32 \$14833.00
Clerical	175	6.74	\$1180.00	\$0.00	0 \$0.00 0.0% \$1180.00 \$1180.00
Total Direct Labor			\$24333.00	\$6825.13	938 \$20420.28 83.9% \$3912.72 \$24333.00
Burden @ 24.6% (Excluding GRA Labor)			\$2337.00	\$432.74	\$1801.11 77.1% \$535.89 \$2337.00
Total Direct Labor and Burden			\$26670.00	\$7257.86	\$22221.39 83.3% \$4448.61 \$26670.00
TRAVEL EXPENSE			\$2500.00	\$611.45	\$1406.45 56.3% \$1093.55 \$2500.00
GENERAL & ADMINISTRATIVE EXPENSE			\$1500.00	\$146.46	\$278.46 18.6% \$1221.54 \$1500.00
COMPUTING CHARGES			\$1500.00	\$441.90	\$1191.90 79.5% \$308.10 \$1500.00
TOTAL DIRECT COSTS			\$32170.00	\$8457.67	\$25098.20 78.0% \$7071.80 \$32170.00
INDIRECT COSTS @ 55.3%			\$17790.00	\$4677.09	\$13879.30 78.0% \$3910.70 \$17790.00
CONTRACT PRICE			\$49960.00		\$49960.00
UNCOMMITTED COMMITMENTS AND EXPENDITURES				\$13134.76	\$38977.50 78.0%

FUNDS EXPENDITURE GRAPH

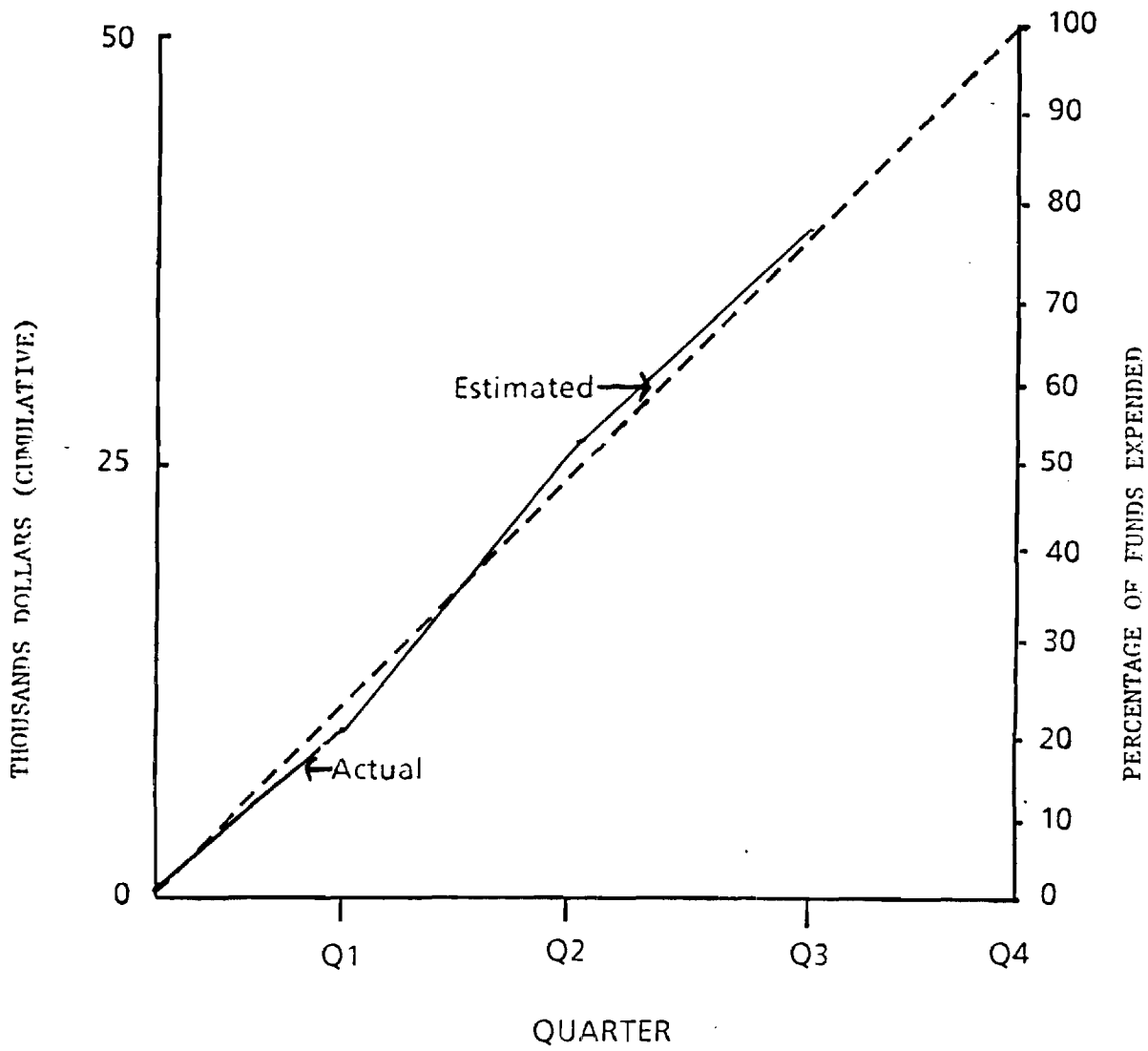


Figure 2

FUNDS EXPENDITURE REPORT

Column A ----- ORIGINAL PROPOSAL	Column B ----- Latest Accepted Revised Proposal	Column C ----- Reporting Quarter Expenditures	Column D ----- Cumulative Expenditures to Date Total Man Hours Dollar Value Pct. Dollar Value	Column E ----- Cost to Complete Estimate	Column F ----- Latest Cost Estimate
Direct Labor					
Number of Hours	Number of Hours	Dollar Total	Number of Hours	Dollar Total	
Hourly Rate					
350	430	\$8320.00	430	\$10220.96	\$2899.36
23.77				\$10220.96	100.0%
\$8320.00	\$10220.96			\$0.00	\$10220.96
1300	1648	\$14833.00	1648	\$18802.35	\$5703.67
11.41				\$18802.35	100.0%
\$14833.00	\$18802.35			\$0.00	\$18802.35
175	0	\$1180.00	0	\$0.00	\$0.00
6.74				\$0.00	100.0%
\$1180.00	\$0.00			\$0.00	\$0.00
Total Direct Labor		\$24333.00	2078	\$29023.31	\$8603.03
Burden @ 24.6%		\$2337.00		\$2387.93	\$756.40
(21.0% starting 7/1/85)				\$2387.93	100.0%
(Excluding BRA Labor)				\$0.00	\$29023.31
Direct Labor and Burden		\$26670.00		\$31411.24	\$9359.43
				\$31411.24	100.0%
		\$2500.00		\$1193.74	\$0.00
				\$1193.74	100.0%
		\$1500.00		\$413.46	\$135.00
				\$413.46	100.0%
		\$1500.00		\$1565.18	\$373.28
				\$1565.18	100.0%
TOTAL DIRECT COSTS		\$32170.00		\$34583.62	\$9867.71
				\$34583.62	100.0%
DIRECT COSTS @ 55.3%		\$17790.00		\$19959.38	\$7276.49
(63.5% starting 7/1/85)				\$19959.38	100.0%
CONTRACT PRICE		\$49960.00			\$54543.00
					\$54543.00
COMMITMENTS AND EXPENDITURES			\$17144.20	\$54543.00	100.0%

FUNDS EXPENDITURE GRAPH

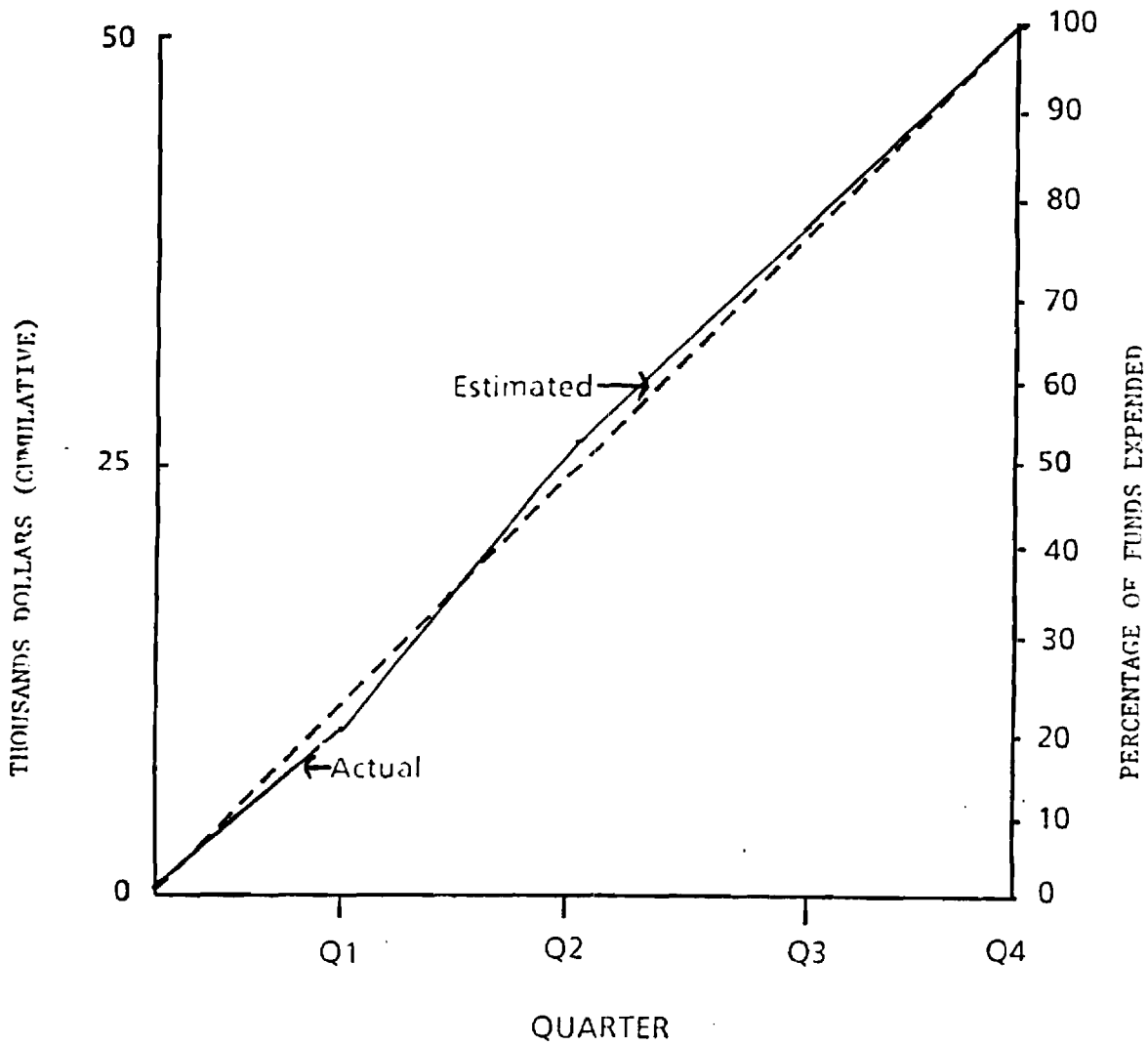


Figure 2

QUARTERLY PROGRESS REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT # MDA 904-84-C-6035
REPORTING PERIOD: 11 SEP 84 - 31 DEC 84

1. Project Status

During the initial quarter of this project, work has begun on each of the two tasks called for by the statement of work. As was planned, these efforts are closely related to work previously in progress within the Clouds Project, a major research effort in the area of reliable distributed computing. Under the Distributed File Systems task, a Ph.D. thesis proposal has been developed, entitled "Storage Management for an Action-based Operating System." A summary of that proposal follows within this report and the entire proposal is attached as an appendix. Under the Language Support for Robust Distributed Programs task, work has been done on refining a programming language called Aeolus, which is intended for programming the Clouds system. A summary of Aeolus appears in a later section of this report, and a recently written paper on Aeolus is attached as an appendix.

The work on the tasks of this project is proceeding on schedule. Future plans include a continuation of the two investigations presented here, including the development of another Ph.D. thesis proposal based on the Aeolus work.

2. Storage Management for an Action-based Operating System

The Clouds Project is an effort to provide support for a distributed computing systems, which achieves performance improvements (over conventional computing systems) through the parallelism possible in a multi-computer environment, and reliability improvements through the redundancy available in processing resources and data storage. In order to achieve such improvements, the system must ensure the proper coordination of processes on various machines in the system and synchronize the use of shared data. The system as a whole must be able to deal with failures of one of its component machines, determining what processes on the failed machine are necessary for the continuation of some larger task. A distributed system must be able to ensure the consistency of data in the presence of machine failures, taking into account that data may be replicated.

The initial goal of the Clouds project is to produce an operating system kernel that supplies the mechanisms needed by a reliable distributed computing system. In supporting these mechanisms, the Clouds kernel must support other conventional mechanisms such as virtual memory, process control and secondary storage management. The action and object support must be integrated with the conventional kernel functions so that support for a reliable distributed computing system is available through a well-defined kernel interface, and the implementation of the kernel is efficient and compact.

One subtask currently in progress is the design and investigation of a portion of the Clouds kernel: the file (storage management) system. An attached Ph.D. research proposal by David Pitts describes the work on this subtask. In addition to describing how such a system can be built, it also considers the interaction of the storage management system with other parts of the kernel, particularly its interaction with the virtual memory system.

The purpose of the proposed research is to design a kernel-level file system that supports a reliable distributed computing system. The file system will manage the secondary storage available on the system. Specifically, the design presents the structures and mechanisms necessary to support the file system. The design will include support for both recoverable and non-recoverable objects. Mechanisms to create, delete, write and read objects on disks are defined. For recoverable objects the additional protocols and structures that will ensure recoverability of objects in the presence of machine failures and action aborts are detailed. The design also discusses the interaction of the file system and the virtual memory system. This portion of the design specifies the structures and mechanisms required for virtual memory. The design also defines the support required for action management and object recovery. Finally a facility for the location of segments on secondary storage must be provided.

The design of the file system will have two phases. Phase one will be a design of essential features for the system. The end-result will be an implementation for the Clouds kernel that will serve as a test-bed for further research. An analysis of the design and implementation will be done to determine the correctness and effectiveness of the design. The results of the

analysis may have an effect on phase two. This phase of the design will include modifications and refinements to the original design. In general, phase two will include features not absolutely necessary for the file system, but which may be desirable later as the system is put to use as a research device. Feedback from the analysis of the original design may suggest some of the changes found in phase two. Phase two is not intended for immediate implementation.

The Clouds kernel will provide support for three basic mechanisms which will be important to later discussion: processes, objects and actions. Processes are the active agents of the system; to initiate and perform any work requires a process. The kernel has a process manager which handles all book-keeping associated with creating, dispatching, and destroying processes.

Objects, on the other hand, are passive entities. Objects are typed collections of data. The type of an object determines what operations may be performed on the data, as well as how the data is organized. Object data can only be manipulated through these operations, and then only by a process which has a proper capability for the object. A capability is a unique name for an object along with a list of operations which are permissible for use by the possessor. The object manager handles the overhead of verifying capabilities and performing operation calls.

Objects will be the organizational units of the system. By using objects, a programmer has a means for abstraction and isolation of data. The kernel also provides a mechanism for organizing sets of operations into a unit. This mechanism is the action. Actions are atomic. The set of operations comprising an action appears to execute completely or not at all. Also, the atomicity of actions prevents the execution of one action interfering with the execution of another. Actions provide a mechanism for making the effects of a set of operations consistent and recoverable.

Actions are also managed by the object manager. Actions require processes in order to perform any task. An action may have several processes or one process executing on its behalf. A single process could be used by several actions at various times.

The kernel provides processes, objects, and actions as efficiently as possible. Particularly, because objects have different types and possible operations, the kernel needs access to objects in a manner which is consistent and convenient. For this reason, all objects have a secondary type, called the segment type. The segment type is a sequence of bytes with primitive operations such as read a page, copy the object from one place to another, or append some data onto the end of the object. The segment is accessible only by the kernel.

2.1. Storage Management

The Clouds file system is managed as a set of partitions. Each partition is an autonomous logical device, having its own device driver which manages requests to the partition. The partition driver passes requests to a physical device driver. A partition resides completely on one physical device and consists of a contiguous set of bytes on disk. The partition driver requires three structures to manage partition storage. First is a partition header, which holds information concerning the partition such as its size, whether it provides support for recoverability, a list of bad disk records for the partition, and other such facts. The header should be duplicated to reduce the risk of its destruction by a media failure or other such disastrous error. The header is placed at a known location in the partition. Before a partition can be accessed by the kernel, it must be mounted on the system. This process consists of initiating the driver for the partition, and giving it the starting location of the partition.

Each partition also maintains a directory, contains a mapping of sysnames (for objects) onto partition record addresses. Note that partition directories contain mappings only for their own segments. Redundancy should also be insured for this structure. The partition driver also knows the location of the directory.

The third partition structure is a record map, which is a bit-map showing allocation of records for the partition. The driver uses the record map to determine which records are in use by segments and which can be allocated. Once again, the record map is an important structure which should be duplicated to

prevent its loss after a media crash. The remainder of the partition is available for the storage of object data, or as the file system treats objects, the storage of segment data.

2.2. Recovery Management and Virtual Memory

Segment recovery is accomplished via a shadowing scheme. That is, segments on which actions are operating will have shadow versions that the actions will actually see. The scheme is pessimistic, so that no modifications are made to a permanent version until the action making the modifications commits. The goals of the recovery scheme are, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few disks accesses as possible. Shadowing, then, will be minimal. That is, only those parts of the segment actually modified are shadowed.

The file system becomes involved in recovery only when an action precommits and the shadow version of the segment on which the action is operating is created. Prior to precommit, all write operations are done in main memory. An active segment is mapped into virtual memory by the virtual memory system. An object's address space contains a block of permanent data and a block of volatile data. The permanent data block contains data which will survive a crash. This is basically the object data. The volatile data block's contents will not survive a machine crash and generally consists of such structures as locks and semaphores for the object. Also contained in the volatile data block is much of the information maintained by the action management system.

When an action operates on a segment, the action management system maintains versions of any modified recoverable parts of the segment in the volatile data block. There may be any number of versions due to the nesting of actions and actions sharing the segment. When a top-level action precommits, data must be moved from the volatile data block to the permanent data block, prior to shadowing the segment on secondary storage. To simplify the precommit procedure, we allow only one action per segment to pass the precommit point. If actions A and B are both operating on object O and A precommits, B is prevented from precommitting. If B attempts to precommit, the action management system blocks the action. B still may access the object.

During the time precommit and commit are taking place, the virtual memory system must insure that modified pages of the permanent data block remain in memory and undisturbed. The virtual memory system can do this by physically locking the pages in memory, making them read-only. Then the pages can be flushed to disk to build the shadow version of the permanent segment.

Because the permanent data block is not modified until precommit, paging of object data can be performed using the permanent segment on disk. However, paging surfaces must be provided for the volatile data block. A partition for temporary data can be created on disk for this purpose. Since all the data concerned is volatile, no recovery is necessary for this partition.

3. OVERVIEW OF AEOLUS

The major design goal of Aeolus is to make possible access to the features of the Clouds system from a powerful systems programming language which supplies those features -- such as strong typing -- which aid in the quick development of error-free programs, yet allows those features to be explicitly circumvented when necessary.

The major structuring features in Aeolus are processes and objects. Objects have two purposes in Aeolus: to provide support for data abstraction, and to reflect the recoverability and synchronization capabilities provided by the Clouds kernel. It has been argued elsewhere [Allc82] that the object construct provides a powerful tool for the organization of programs for recovery, both from the standpoint of the programmer and of the system. Objects may rely on the automatic operating system / runtime system support for synchronization and recovery (*recoverable* and *autosynch* objects). Alternatively, using powerful features provided by the language and the Clouds system, the programmer may take advantage of semantic knowledge about the application to explicitly code more appropriate recoverability and synchronization. However, Aeolus objects also

provide abstraction features even when synchronization and recovery are not required. These *nonrecoverable* objects provide a logical framework for the organization of modules for separate compilation.

3.1. Features for Systems Programming

In keeping with its purpose as a systems programming language, Aeolus incorporates several features which give the programmer access to the hardware and the lower levels of the systems software, as well as "convenience" features which allow more efficient coding, including:

- a full range of assignment and bit-manipulation operators similar to those in the C language;

- features for register optimization, such as a special *index* type for loop counters and array references;

- the option of specifying *inline* expansion of a procedure;

- a facility for specifying *arbitrary* procedure argument lists of unspecified length and (predefined) types (similar to the *nospread* arglists of Interlisp);

- and the ability to specify storage addresses for variables, as well as some capabilities for setting and doing arithmetic on pointers.

However, most of the power of Aeolus as a systems programming language, aside from the access it provides to the features of the Clouds system, lies in the ability it gives the programmer to specify low-level data structures as abstract data types, and in the treatment of the underlying hardware as an object with operations on its state available from the language.

In addition to the usual structured types (records and arrays), Aeolus provides a *structure* type, which allows the programmer to specify abstract types for the manipulation of bitfields. The *structure* is similar to the *packed record* construct of Pascal, except that the programmer indicates that its fields should fit one of the addressable entities defined by the target computer (byte, word, doubleword, quadword, etc.), and this correspondence is checked by the compiler. This provides a secure mechanism allowing bit fields within a low-level data structure to be referenced by name. Aeolus also provides the *byte* and *word* types as predefined objects. These objects have operations permitting manipulations similar to those of the *bitset* type of Modula-2. The programmer may define similar objects for bit strings of other lengths.

The ability to inspect and change the state of the hardware is also important in systems programming. Access to the underlying hardware is provided by the operations of special Aeolus objects. We call such an object a *pseudo-object* since only one instance of it may exist, whereas there may be an arbitrary number of instances of a normal object. An example of a pseudo-object is *PC_System*. This pseudo-object gives access to the registers and ports of a PC's microprocessor, and through the ports to the other system components, such as the interrupt controller, device controllers, and modem registers. For example, the *IN_BYTE* and *OUT_BYTE* operations of *PC_System* allow values to be input and output from the byte ports of a PC; other *PC_System* operations provide such capabilities as access to the register set, flags, and interrupt mechanism. These operations typically compile inline to a single machine instruction. For considerations of efficiency, some operations in hardware pseudo-objects may give access to special instructions of the target machine, such as the string instructions of the PC or the polynomial instructions of the VAX.

3.2. Features for Object and Action Programming

The design of Aeolus is intended to support the recovery and synchronization capabilities of the Clouds system in a high-level systems programming language. Objects in Aeolus, besides providing an organizational tool for secure separate compilation, give access to the recovery properties of Clouds

objects. Thus, unless an Aeolus object is designated as *nonrecoverable*, the Clouds kernel mechanisms are used for invocations of its operations, allowing the system to control the recoverability properties of the object's state. In the remainder of this section, the features provided by Aeolus for accessing these features of Clouds are examined.

The code for an Aeolus object has two parts. The *definition* part is seen both by the object itself when it is being compiled, and by all other objects or programs which use that object. Compilation of a definition part produces a symbol table file which is used for type checking among these separate compilations. It can contain specifications of public types and constants defined by the object, and the interface definitions of the object's operations. Definition parts may not contain variable declarations. The *implementation* part contains the actual code of the operations, along with any needed local (private) type, constant, or procedure definitions. Local variables of an object share the lifetime of the object instance to which they belong, and thus act as "own" variables. This separation of definition and implementation provides a safe separate compilation mechanism similar to *packages* in Ada (TM) or *modules* in Modula-2.

The general syntax of object implementation parts in Aeolus is as follows:

```
implementation of [ nonrecoverable | recoverable | autosynch | epsilon ]
                  object <object id> is
  uses <id list>
  import <id list>
  action events <override list>
  <block>
end implementation.
```

(Although not shown in this example syntax, objects may be specified as being *autosynch* and *recoverable* simultaneously.) If the object is specified as being *nonrecoverable*, it is treated as being simply a separate compilation module. That is, operations in *nonrecoverable* objects are compiled using the standard preludes and postludes for procedure bodies, without special code or system calls for recovery. If the object is specified as being *recoverable*, the compiler provides a standard run-time framework for recovery by generating preludes and postludes for the object operations using Clouds object and action manager calls. Thus, the programmer may gain access to the action mechanisms of the Clouds system with a single keyword. However, the full power of the Clouds action mechanisms may be unnecessary and inefficient in some cases. For those cases, the Aeolus/Clouds system provides mechanisms which allow the user to explicitly program recovery strategies tailored to the individual requirements of the problem at hand. Therefore, if neither the *nonrecoverable* nor the *recoverable* keyword is given in an object header, it is assumed that object recovery is explicitly programmed. In this case, the programmer may provide alternate recovery procedures for recoverable variables of the object, and may also specify, in the *action events* clause, handlers other than the default system handlers for the *precommit*, *commit*, and *abort* events of the entire object. The compiler then specifies to the action and object management systems that, when one of the action events occurs, these alternate handlers are to be invoked instead of the standard, system-provided procedures.

The Aeolus language also provides access to the synchronization mechanisms of the Clouds system. When the *autosynch* object attribute is specified in an object header, it indicates that the default system synchronization procedures are to be used on the object's operations to provide concurrency atomicity. If the *autosynch* attribute is not specified, synchronization may be explicitly programmed using operations on the *lock* type provided by the language. A Clouds lock [Allc83b] is not associated with a physical object, but rather with values in the domain of the object. Thus -- for example -- a file name may be locked, even if a physical file with that name does not yet exist.

The *uses* clause allows the programmer to specify the use of system pseudo-objects, while the *import* clause allows other user-defined or system-defined object definitions to be accessed. In a *<block>*, definitions of types, constants, variables, recoverable variables, internal procedures, and operations may be written in any order (as long as their definitions appear before any uses); the *<statement part>* of the block is treated as an initialization routine to be executed upon creation of an instance of the object.

Object operations are programmed like procedures. An operation invocation looks like a procedure invocation with a prefix indicating the object instance upon which to operate:

```
<object instance id> @ <operation id> ( <actual param list> )
```

An object instance may be created by declaring a variable of that object type, and then allocating the instance's data storage on the heap using an extended version of the allocation function, or by associating the variable with a "permanent" object, much as a file variable can be associated with a physical file in Pascal.

Operations on local procedures of (recoverable) Aeolus objects may be specified to be invocable as an action. The syntax of action implementations is much like that of procedures:

```
procedure <proc id> ( <formal param list> ) is action
  <procedure block>
end procedure
```

(A <procedure block> is the same as a <block> except that it cannot contain declarations of recoverable variables.) Thus, the invocation of an action is similar to a procedure invocation; however, a unique *action-id* is created by a Clouds action manager for the invocation, which may be assigned to a variable of the invoking procedure:

```
<action-id var> := action <proc id> ( <actual param list> )
```

This *action-id* variable may be used to retrieve information from the system about the status of the action, or to abort the action, using calls to a Clouds action manager. This mechanism allows general control structures to be formulated, e.g., for the concurrent invocation of actions.

Storage Management for an Action-based Operating System

David Pitts

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

As computing systems enter more and more human activities, providing various sorts of services, it becomes increasingly important that these systems be reliable. The Clouds project is research into how reliable computing systems can be built. Such a system is supported at a low level by the Clouds kernel, currently under development. The kernel supports user-defined objects and atomic actions, with which the kernel can provide recoverability in the presence of machine and system failures.

An important component of the kernel is a reliable file system. This proposal describes a plan for building such a file system. An overview of the preliminary design is given, discussing the structure of the system and techniques for insuring recoverability of the file system. The proposal also discusses the interaction of the file system with the virtual memory system. Some tentative ideas are presented for the design of the virtual memory system. The proposal also suggests some areas for evaluation of the design.

Introduction

Many potential advantages have been suggested for distributed computing systems, most notably performance improvements through the parallelism possible in a multi-computer environment, and reliability improvements through the redundancy available in processing resources and data storage. In practice, these improvements have been hard to achieve. The system must ensure the proper coordination of processes on various machines in the system and synchronize the use of shared data. The system as a whole must be able to deal with failures of one of its component machines, determining what processes on the failed machine are necessary for the continuation of some larger task. A distributed system must be able to ensure the consistency of data in the presence of machine failures, taking into account that data may be replicated.

The Clouds Project is an effort to provide support for such a system. The initial goal is to produce an operating system kernel that supplies the mechanisms needed by a reliable distributed computing system. Among mechanisms supported by the Clouds kernel are those described in the action-object model of [Allchin 83] and developed further in [McKendry 84]. In supporting these mechanisms, the Clouds kernel must support other conventional mechanisms such as virtual memory, process control and secondary storage management. The action and object support must be integrated with the conventional kernel functions so that support for a reliable distributed computing system is available through a well defined kernel interface, and the implementation of the kernel is efficient and compact.

This proposal describes a plan for the design and investigation of a portion of the Clouds kernel: the file (storage management) system. In addition to describing how such a system can be built, we look at the interaction of the storage management system with other parts of the kernel, particularly its interaction with the virtual memory system. The system described in this proposal is a reduced design, in that it provides the functionality required by the kernel but not much more. Therefore, we also present some ideas for future expansion and refinement of the storage system, and examine how such a system can be used.

The second chapter presents the problem we are addressing, and also provides some background concerning the environment in which we are working. The third chapter discusses the design and implementation of the file system. A set of protocols for segment recovery is presented along with a discussion of the structure of the file system. The requirements and structure of the virtual memory system are also discussed. The fourth chapter discusses related work. The fifth chapter presents an outline of the thesis.

Problem Statement and Background

The purpose of the proposed research is to design a kernel level file system that supports a reliable distributed computing system. The file system will manage the secondary storage available on the system. Specifically, the design presents the structures and mechanisms necessary to support the file system. The design will include support for both recoverable and non-recoverable objects. Mechanisms to create, delete, write and read objects on disks are defined. For recoverable objects the additional protocols and structures that will ensure recoverability of objects in the presence of machine failures and action aborts are detailed. The design also discusses the interaction of the file system and the virtual memory system. This portion of the design specifies the structures and mechanisms required for virtual memory. The design also defines the support required for action management and object recovery. Finally a facility for the location of segments on secondary storage must be provided.

The design of the file system will have two phases. Phase one will be a design of essential features for the system. The end-result will be an implementation for the Clouds kernel that will serve as a test-bed for further research. An analysis of the design and implementation will be done to determine the correctness and effectiveness of the design. The results of the analysis may have an effect on phase two. This phase of the design will include modifications and refinements to the original design. In general, phase two will include features not absolutely necessary for the file system, but which may be desirable later as the system is put to use as a research device. Feedback from the analysis of the original design may suggest some of the changes found in phase two. Phase two is not intended for immediate implementation.

The next section will present an overview of the first phase design, plus some tentative ideas for the second phase. The remainder of this section provides some context for the proposed file system.

An overview of the Clouds kernel is given in [Spafford 84]. The report describes the components of the kernel, what services or mechanisms the components provide, brief overviews as to how the components provide the services, and how the components interact. The kernel provides three basic mechanisms which will be important to later discussion.

The Clouds kernel supports processes. Processes are the active agents of the system; to initiate and perform any work requires a process. The kernel has a process manager which handles all bookkeeping associated with creating, dispatching, and destroying processes.

Objects, on the other hand, are passive entities. Objects are typed collections of data. The type of an object determines what operations may be performed on the data, as well as how the data is organized. Object data can only be manipulated through these operations, and then only by a process which has a proper capability for the object. A capability is a unique name for an object along with a list of operations which are permissible for use by the possessor. The object manager handles the overhead of verifying capabilities and performing operation calls.

Objects will be the organizational units of the system. By using objects, a programmer has a means for abstraction and isolation of data. The kernel also provides a mechanism for organizing sets of operations into a unit. This mechanism is the action. Actions are atomic. The set of operations comprising an action appears to execute completely or not at all. Also, the atomicity of actions prevents the execution of one action interfering with the execution of another. Actions provide a mechanism for making the effects of a set of operations consistent and recoverable.

Actions are also managed by the object manager. Actions require processes in order to perform any task. An action may have several processes or one process executing on its behalf. A single process could be used by several actions at various times.

The kernel provides processes, objects, and actions as efficiently as possible. Particularly, because objects have different types and possible operations, the kernel needs access to objects in a manner which is consistent and convenient. For this reason, all objects have a secondary type, called the segment type. The segment type is a sequence of bytes with primitive operations such as read a page, copy the object from one place to another, or append some data onto the end of the object. The segment is accessible only by the kernel.

The Proposed Research

In this section we sketch a tentative first phase design of the file system. Included in this design is a discussion of the interaction of the virtual memory system with the file system. Our discussion starts at a point presented in [Spafford 84]. In this report, an overview of the Clouds kernel design is presented. In particular, structures and general techniques are sketched. The purpose of phase one is to refine and expand this overview into a design which can be implemented.

Storage Management

The Clouds file system is managed as a set of partitions. Each partition is an autonomous logical device, having its own device driver which manages requests to the partition. The partition driver passes requests to a physical device driver. A partition resides completely on one physical device and consists of a contiguous set of bytes on disk. The partition driver requires three structures to manage partition storage. First is a partition header, which holds information concerning the partition such as its size, whether it provides support for recoverability, a list of bad disk records for the partition, and other such facts. The header should be duplicated to reduce the risk of its destruction by a media failure or other such disastrous error. The header is placed at a known location in the partition. Before a partition can be accessed by the kernel, it must be mounted on the system. This process consists of initiating the driver for the partition, and giving it the starting location of the partition.

Each partition also maintains a directory, contains a mapping of sysnames (for objects) onto partition record addresses. Note that partition directories contain mappings only for their own segments. Redundancy should also be insured for this structure. The partition driver also knows the location of the directory.

The third partition structure is a record map, which is a bit-map showing allocation of records for the partition. The driver uses the record map to determine which records are in use by segments and which can be allocated. Once again, the record map is an important structure which should be duplicated to prevent its loss after a media crash. The record map's use will be discussed further in the presentation of the recovery protocols.

The remainder of the partition is available for the storage of object data, or as the file system treats objects, the storage of segment data. Figure 1. illustrates a simple partition consisting of one segment.

Segments are located through the partition directory. The sysname for the object is mapped to the partition record where

Partition

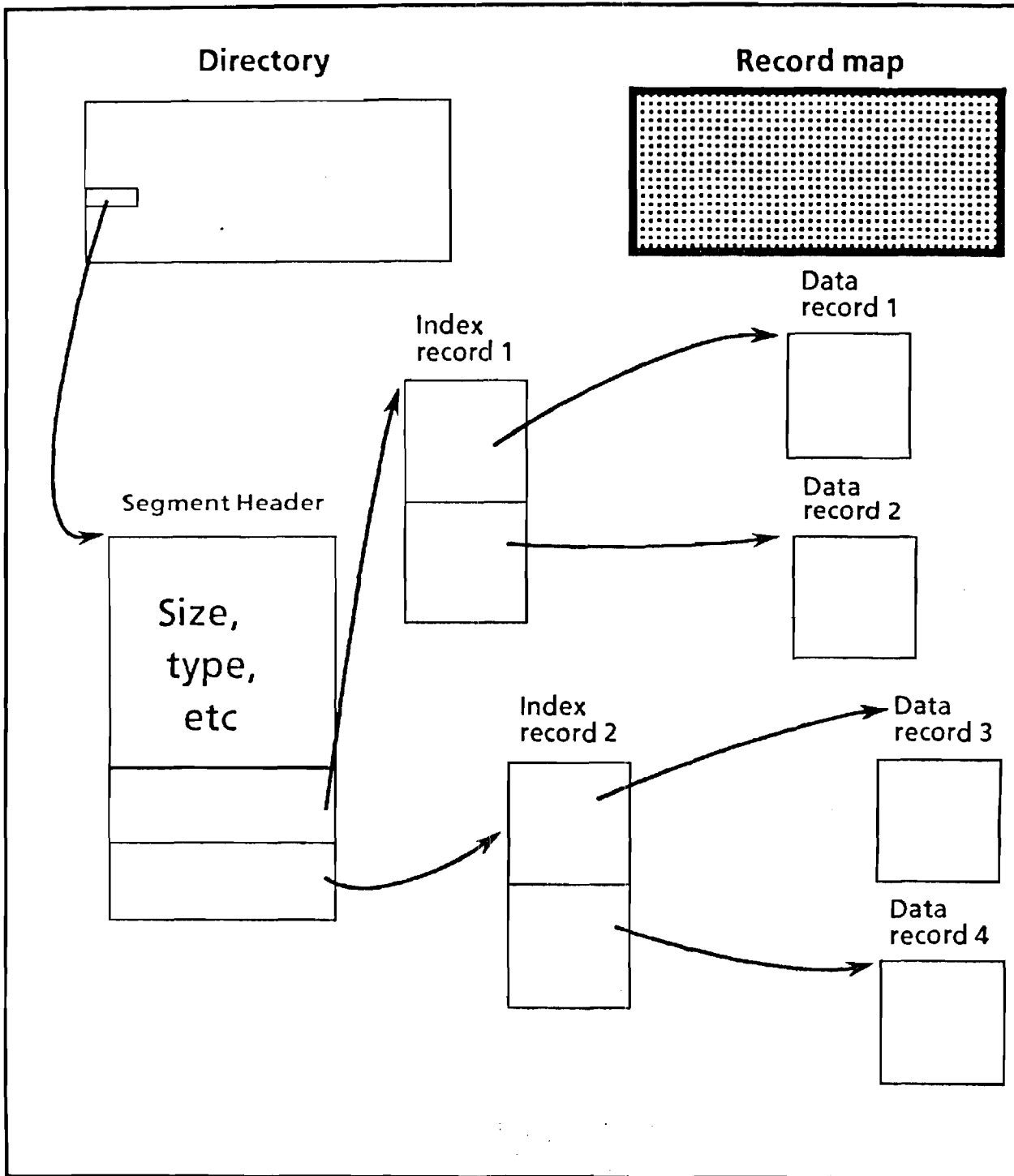


Figure 1.
A partition with one segment shown

the segment resides. This record contains the segment header. Like the partition header, the segment header contains information about the segment, such as its size (in bytes), its (object) type, and partition management information (mainly for recovery). In addition, the header contains a set of pointers to other partition records. These records may contain segment data or may be another set of pointers, called an index record. The pointers in the index record may point to yet other index records. The exact structure of the segment depends upon its size. The implementation will limit segments to having two levels of index records. The leaf nodes of the tree of records for a segment will be data records. Thus access to a segment requires first locating on which partition the segment resides and mapping the sysname for the segment in the partition directory. This gives a pointer to the segment header. Access of the particular segment record requires a look at the size of the segment and following the appropriate path of pointers. For small segment this may take only one access. For larger segments, up to three accesses may be necessary.

Recovery Management

Segment recovery is accomplished via a shadowing scheme. That is, segments on which actions are operating will have shadow versions that the actions will actually see. The scheme is pessimistic, so that no modifications are made to a permanent version until the action making the modifications commits. The goals of the recovery scheme are, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few disks accesses as possible. Shadowing, then, will be minimal. That is, only those parts of the segment actually modified are shadowed.

The recovery scheme consists of a set of protocols that dictate what the file system must do for segment states and action events. To describe the recovery scheme we will consider these states and actions one-by-one and present the protocol for each. For the purpose of the discussion, suppose that an action A is started which eventually references a recoverable segment S (Figure 2. illustrates the partition on which S resides). Initially, nothing need be done as far as the file system is concerned as long as all references to the disk segment are read references. It is only when the action makes a modification to the disk segment that any overhead need be incurred. The time that A makes such a modification to S is when A precommits. At this time A flushes all modified pages to disk so that they will be in permanent storage, allowing the action to commit safely, even if the commit process is interrupted by a machine crash. However, the precommit must be carefully performed, because it is subject to being interrupted as well and we do not want the file system to be in an inconsistent state as a result.

To insure that precommit is done correctly, the file system will follow two protocols. The first is a header shadow protocol,

which performs the following operations:

- HS1) Storage must be allocated for a shadow version of the segment header.
- HS2) The segment header on disk must be set to indicate that the segment is being shadowed. Note that in Figure 2. this field is initially labelled PERMANENT. When the object is being shadowed, this field is set to SHADOWED as shown in Figure 3. The shadow pointer field should be set to point to the shadow segment. Originally, the field is set to NULL, as shown in Figure 2.
- HS3) The data from the permanent segment header is copied to the shadow version.

The results of this protocol for segment S can be seen in Figure 3. Once the shadow is established, the modified page can be shadowed on disk using the following page shadow protocol:

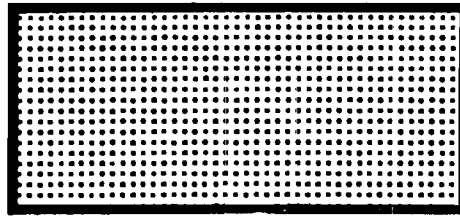
- PS1) Allocate storage for a shadow version of the modified record and a shadow version of any index record needed to access the modified page.
- PS2) The data for the modified page, and any index record are copied to the shadow versions. Pointers in the segment header shadow and any other shadowed index record should be modified at this time.
- PS3) The permanent segment header should have its shadow pointer set to the address of the shadow segment header.

Two notes here. Because our scheme is pessimistic none of action A's effects are allowed to be permanent until A commits. In particular, the permanent page map cannot be changed to show that shadow records have been allocated. Instead, all such allocations occur on a temporary page map, which could be kept in volatile memory. Eventually, upon commit of the action, the allocations will have to be made part of the permanent record map. A similar set of protocols must be enforced for this update as well.

Secondly, note that steps PSP, PS3, and HS3 require the writing of at least three disk records altogether. We are assuming that these writes are atomic. The proposed research will include a detailed analysis of the protocols with respect to this assumption. The results will allow conclusions as to the necessity of atomic writes for the protocols, when the assumption of having atomic writes can be relaxed, and the costs of not having atomic writes (the damage to recoverability).

Once this protocol has been carried out, all future references by A (either read or write references) refer to the shadow version of segment S. However at this point most of segment S is

Volatile Record map



Partition

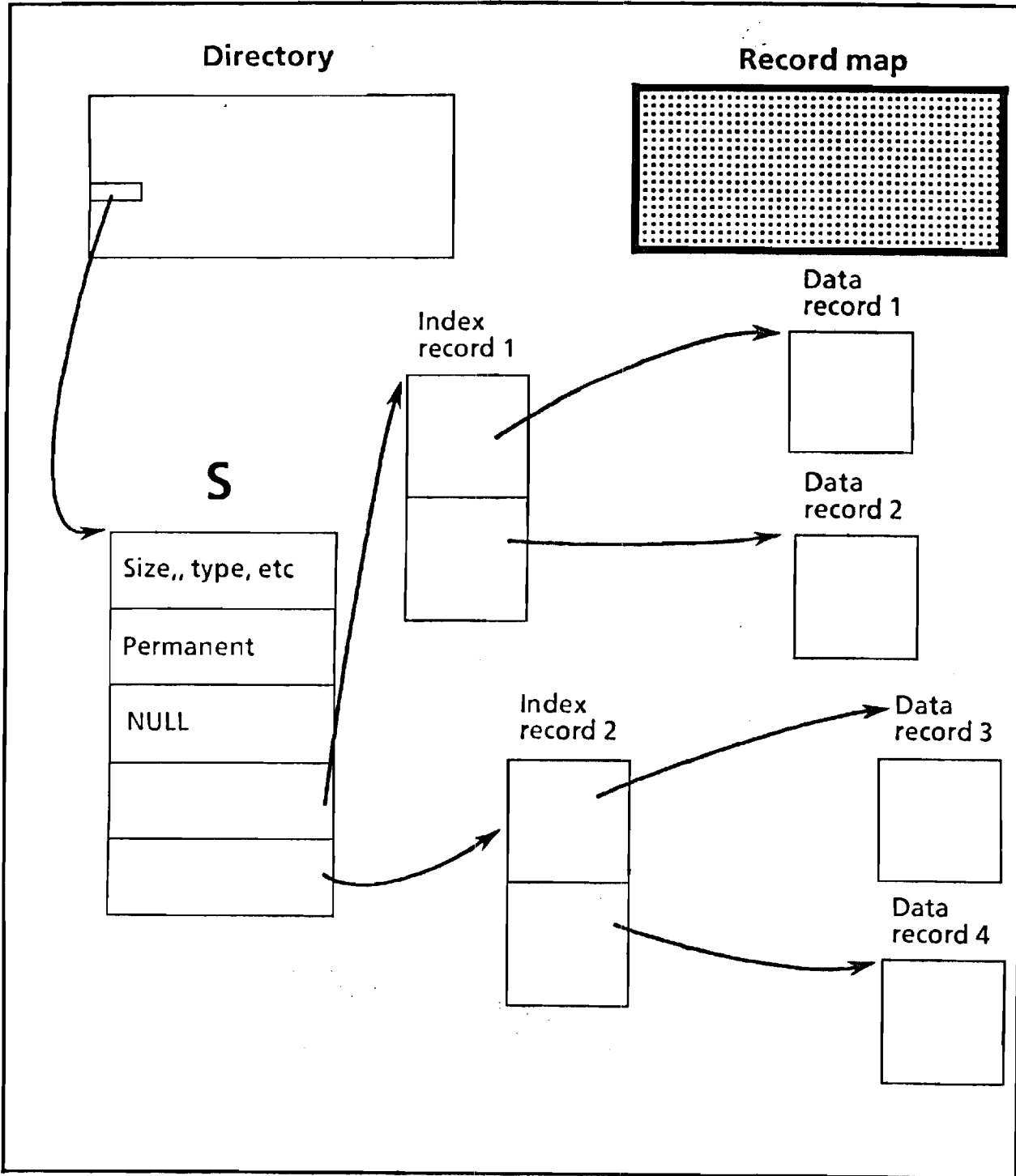
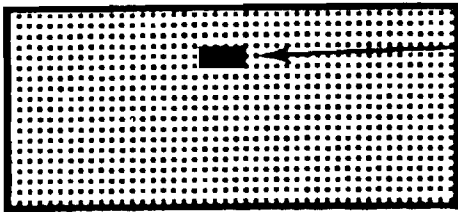


Figure 2.
Segment S

Volatile Record map



Storage allocated for segment header shadow

Partition

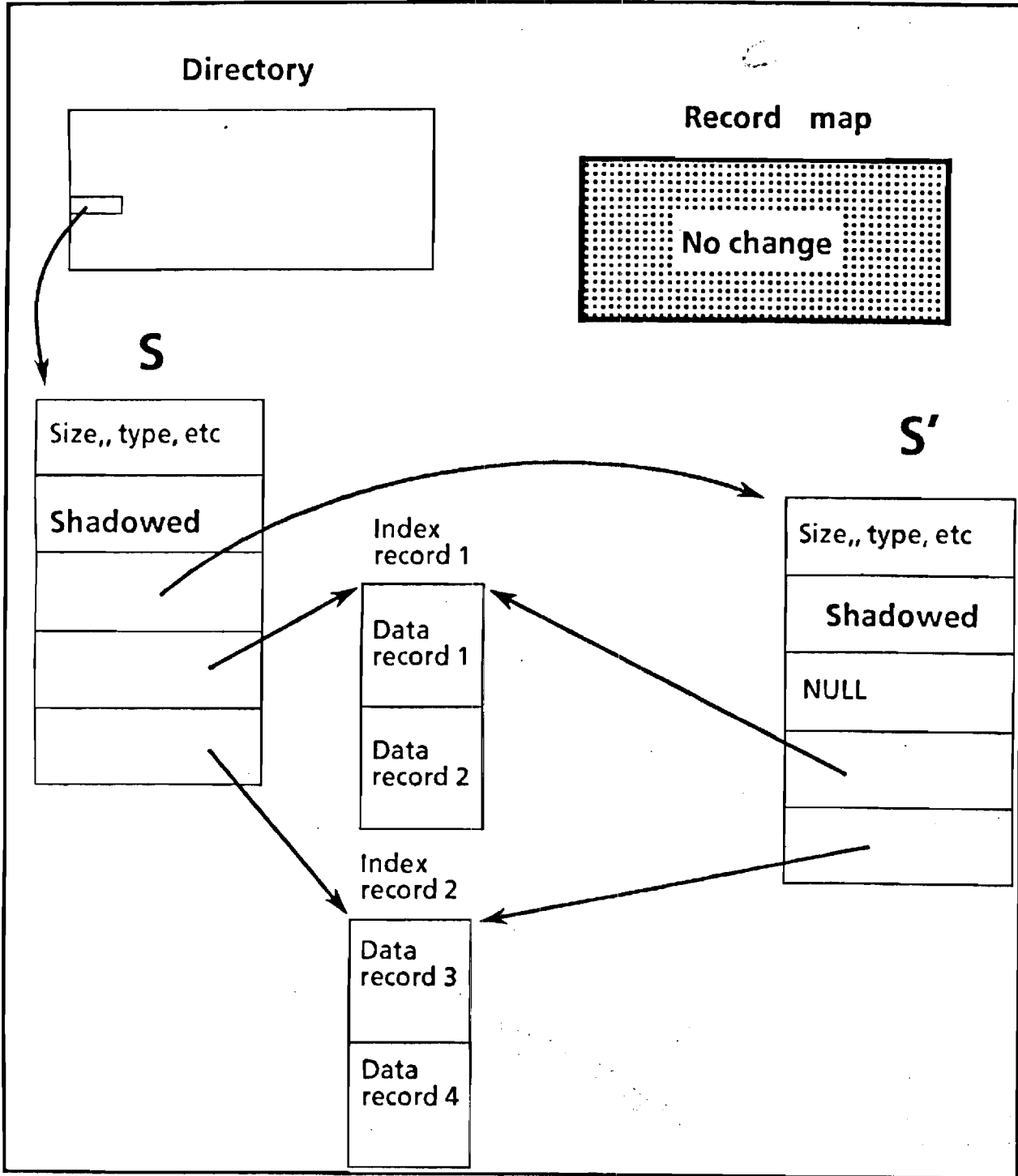


Figure 3.
After shadowing the segment header

unshadowed and read references to pages other than record x will refer to permanent records. Any modifications to some other record y will cause steps 2 and 3 to be performed for the modified record and any index records that are required to access y.

The precommit protocol uses the shadow protocol as follows:

P1) Perform the header shadow protocol.

P2) Perform the shadow protocol for each page modified by A.

P3) Set the shadow flag of the segment header shadow to precommitted.

Figure 4. shows the precommitted segment S. Once this protocol has completed, all modified pages in the permanent segment are shadowed and the commit procedure can proceed.

So this brings us to the commit protocol. The protocol consists of the following:

C1) Update the permanent page map on disk. This requires that all addresses for shadow records be allocated in the page map and all modified records of the segment including the segment header be deallocated in the page map.

C2) The shadow segment header is set so that it is now the permanent segment header.

C3) The partition directory is set so that it points to the new segment header for segment S.

Once this protocol is complete, any references to the segment will refer to the new version of the segment as can be seen in Figure 5. The new segment is a merging of old unmodified records and new records.

Actions can also abort for one reason or another and a file system protocol is required for this event as well. The protocol simply rids the file system of any trace of action A's work as follows:

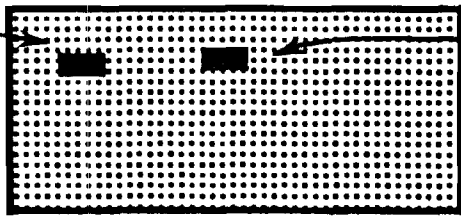
A1) The volatile page map is updated to removed allocations for A's shadow records of segment S.

A2) The permanent segment header for S is set to show that S is unshadowed and the shadow pointer is set to null.

Figure 6. shows the segment after an abort of action A. Note that this time the permanent record map is unchanged, while the volatile map is updated.

Volatile Record map

Storage allocated for index record 2'



Storage allocated for data record 4'

Partition

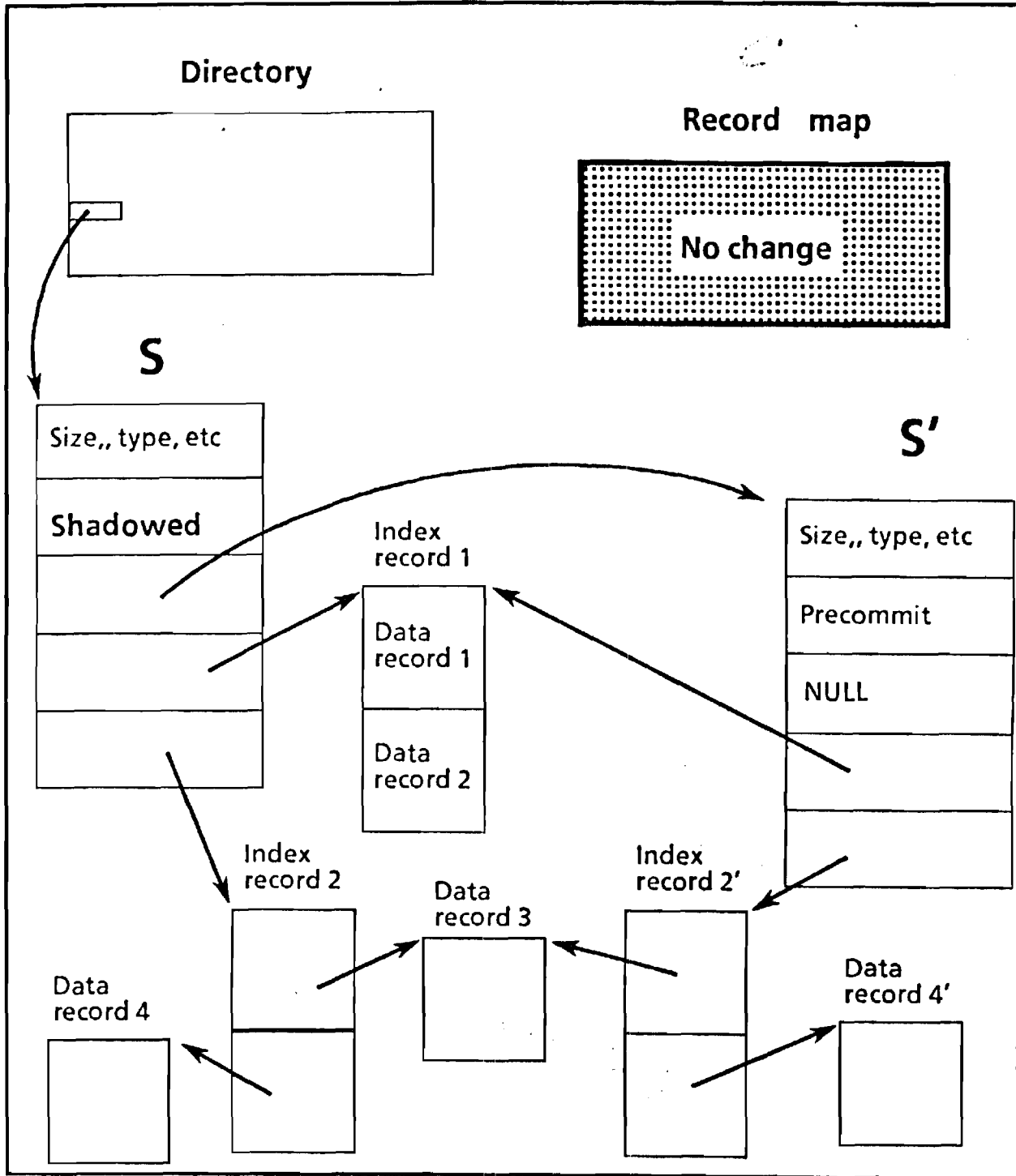
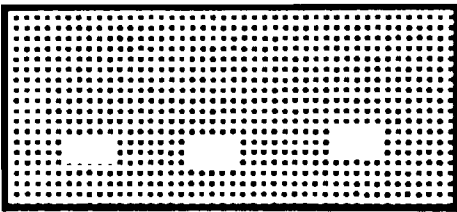


Figure 4.
After precommit



Note that the white blocks in the record maps represent deallocations.

Partition

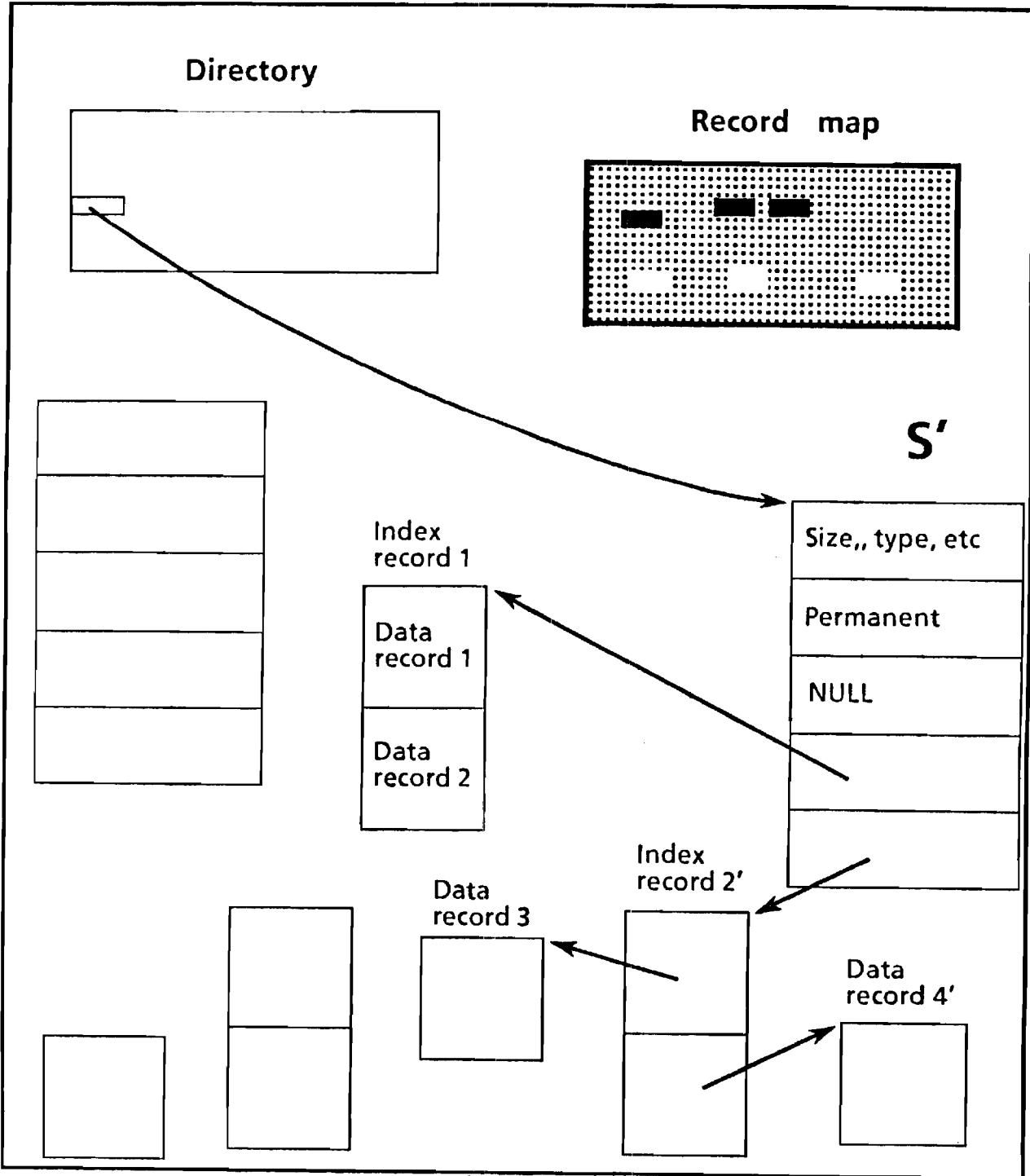
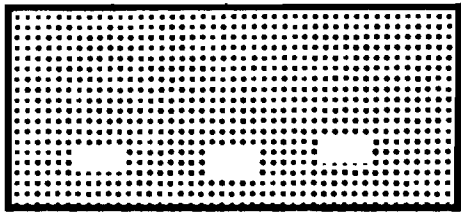


Figure 5. After commit

Volatile Record map



Note that the white blocks on the record map represent deallocations.

Partition

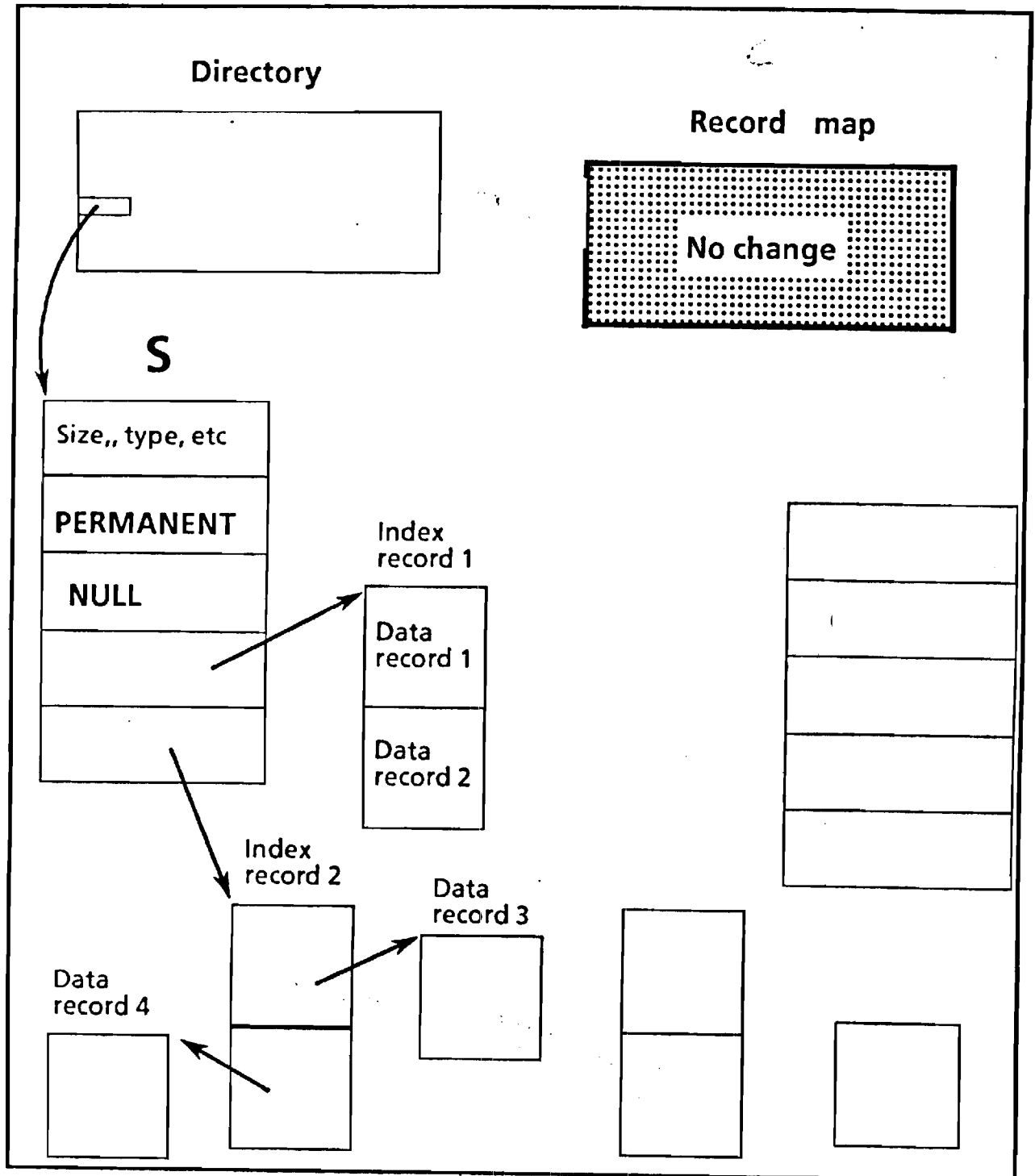


Figure 6.
After abort

One final event must be considered. That is how does the system recover from a machine crash? The protocol for this event requires:

CR1) A new volatile page map is created.

CR2) Every segment modified by an action is inspected and categorized. In our example, S is inspected to determine if A had precommitted. Depending on the result, one of step 3 or step 4 will take place.

CR3) If A had precommitted, then the commit or abort protocol is completed as appropriate.

CR4) If A had not completed, the abort protocol is performed.

The above protocols are carried out by sending requests to the partition driver, which converts the requests to the appropriate disk operations and queues the operations at the physical device driver. Note that the driver can reschedule operations. It does not necessarily perform them in the order received. The operations are scheduled to optimize access to the physical device, and in general, we do not want to interfere with the scheduling. However, the protocols require that pages be flushed to disk at precommit to insure that the file system can be recovered properly. Therefore, we must specify when the operations can be flushed and when the driver should be left to handle scheduling.

Virtual Memory

The file system becomes involved in recovery only when an action precommits and the shadow version of the segment on which the action is operating is created. Prior to precommit, all write operations are done in main memory. An active segment is mapped into virtual memory by the virtual memory system. An object's address space contains a block of permanent data and a block of volatile data. The permanent data block contains data which will survive a crash. This is basically the object data. The volatile data block's contents will not survive a machine crash and generally consists of such structures as locks and semaphores for the object. Also contained in the volatile data block is much of the information maintained by the action management system.

When an action operates on a segment, the action management system maintains versions of any modified recoverable parts of the segment in the volatile data block. There may be any number of versions due to the nesting of actions and actions sharing the segment. When a top-level action precommits, data must be moved from the volatile data block to the permanent data block, prior to shadowing the segment on secondary storage. To simplify the precommit procedure, we allow only one action per segment to pass the precommit point. If actions A and B are both operating on

object O and A precommits, B is prevented from precommitting. If B attempts to precommit, the action management system blocks the action. B still may access the object.

During the time precommit and commit are taking place, the virtual memory system must insure that modified pages of the permanent data block remain in memory and undisturbed. The virtual memory system can do this by physically locking the pages in memory, making them read-only. Then the pages can be flushed to disk to build the shadow version of the permanent segment.

Because the permanent data block is not modified until precommit, paging of object data can be performed using the permanent segment on disk. However, paging surfaces must be provided for the volatile data block. A partition for temporary data can be created on disk for this purpose. Since all the data concerned is volatile, no recovery is necessary for this partition.

Each object and process in the system has its own address space. To provide the mapping for the address spaces, the virtual memory system uses virtual address maps [Spafford 84]. Each virtual address map maps the contents of a different address space. A virtual memory map holds a pointer to the page tables for the address space in addition to a set of entries which specify the data that is mapped into the address space. This mapping is provided by segment control blocks. A segment control block indicates which portion of the segment is being mapped and the state of the pages making up this mapping. The entries in the virtual address map have a pointer to the segment control block, plus an indication of where in the address space the segment can be found. For instance, a process P's map may have an entry mapping file F into P's address space. The entry points to a segment control block for F. Since F could be very large, P might not want to map the entire file into its address space, so the segment control block indicates that only byte 64 to 2048 are being mapped. P's virtual address map entry for F indicates that this block of data is located at addresses 2000 through 3984 of P's address space. See Figure 7.

Additionally, there are object control blocks and process control blocks which point to virtual address maps for the object or process. Using the virtual memory facilities provided, the kernel can share and restrict access of address spaces. Recall that an object's address space contains the volatile data block. For recoverable objects part of this block is used to maintain versions of the recoverable data. By proper mapping of address spaces in virtual memory, the action management system and the object itself can insure that actions see versions they are using and prohibit other actions from seeing these versions.

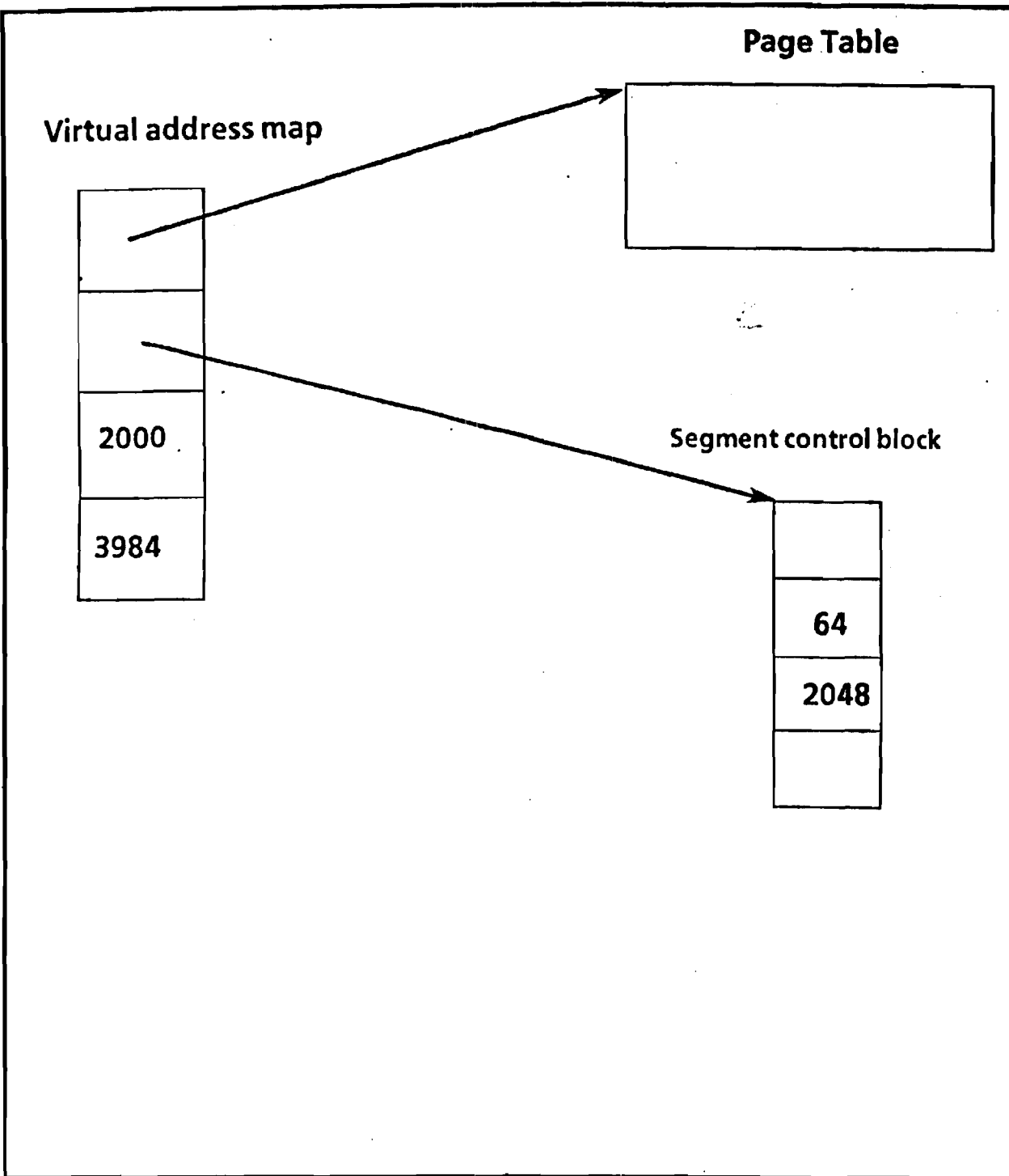


Figure 7.
Virtual Memory.

Related Work

The Clouds kernel file system provides reliable storage management. Object data is recoverable automatically in the presence of machine crashes. Most current file systems for operating systems do not support similar facilities, but instead rely on a scavenger approaches entailing intervention by an operator. Database management systems have supported automatic crash recovery and transactions (similar to our atomic actions) [Gray 79, Gray 81, Lindsay 84]. [Verhofstad 78] presents a survey of recovery techniques used by database management systems. However, those models of computation differ from ours. A database transaction may consist of reading values, processing those values, and then rewriting some of the values. Operating systems tasks are more general than this and involve more complex data. An operating system running on the Clouds kernel would work with actions consisting of sets of object operation calls.

[Stonebraker 81] describes incompatibilities between the services provided by operating systems and the functionality required by database management systems. This study interests us because some database systems support facilities similar to those required of the Clouds kernel. Database systems which run on top of operating systems must either adapt the existing services to their needs or duplicate services, when such services either do not exist or can not be adapted. Both of these alternatives can lead to performance degradation. We will examine some of problems [Stonebraker 81] mentions, explaining how they are addressed by the Clouds kernel. We also look at some existing systems.

A weakness of many operating systems is the handling of I/O. Most current system queue I/O requests at a device driver. The device driver schedules the requests based on device status and where the information is on the device. The information is transferred in and out of a collection of buffers. I/O operations do not necessarily take place when a task issues them. Suppose a task requests that a file be written to disk. This request is broken up into a number of page write requests, which are queued at the device driver. The task then terminates. If the writes are all actually performed there is no problem. However, the machine could crash before the writes are done, or when some writes have been performed and some have not. The file data, then, is either incorrect (no writes are done) or inconsistent (some writes are done some are not). Pages can be left in inconsistent states if the machine crashes in the middle of a page write.

The DMERT operating system [Wallace 83], which uses a basically UNIX-like file system, addresses some these problems by providing a protocol for writing files. The protocols preserve the consistency of the file system as a whole. However, since DMERT apparently does not force out buffered writes and writes the data

in place (no shadowing), the consistency of data inside files can be destroyed. The Clouds kernel maintains consistency through the support of atomic actions at a low level and by supporting the actions with the set of protocols provided by the file system.

[Stonebraker 81] also notes that I/O transfers from system buffers can be expensive. Many database systems prefer to maintain user buffers for this purpose. Note that since the Clouds kernel maps objects into the address space of the processes' referencing the objects, each user effectively has his own buffer area. Further, requests for data not currently in memory are handled by the page fault mechanism of virtual memory. It is expected that this approach will be very efficient. The Pilot operating system also takes this approach [Redell 80].

A problem with mapping objects into virtual memory arises when large objects are referenced. The larger the object being mapped is, the more significant the size of the page tables associated with this mapping become. With very large objects, the system may page out the page tables for the object. A page fault for a missing data page could then generate two faults. Our solution is to provide partial mappings, as noted earlier. This reduces the number pages that are mapped, which should reduce page faults. Stonebraker points out two problems with this solution. The portions of the object being mapped must be determined, and any number of mappings may be required over the duration of the processing.

As noted, the Clouds kernel supports a mechanism similar to the transactions found in database management systems such as System R [Gray 81]. System R supports transactions differently. Whereas the Clouds kernel relies on pessimistic recovery, System R uses an optimistic approach. Transaction modifications are done immediately and must be undone if the transaction aborts. Part of the rationale for using pessimistic recovery arises from the fact that undo procedures are not required.

Data recovery in System R is supported by both a logging technique and shadowing. A log is kept of all transactions. On abort, this log can be used to undo a transaction. The shadowing technique used by System R is quite different from that used by Clouds, since it supports optimistic recovery. There is slightly more overhead associated with System R shadowing, because separate directory entries are maintained for shadows.

Although we support techniques used by other systems, the Clouds file system is unique in many ways. The involvement of the virtual memory system in action management and the file system is different. We expect that this design will increase the efficiency of the implementation. Our approach to segment and partition recoverability is a comprehensive one. Recovery of both is handled at the lowest level of the system. The shadowing technique used by the file system is minimal as far as storage is concerned. Furthermore, shadows exist only for the last part of an action's duration, reducing storage cost still more. Our

method of handling I/O requests relieves processes from having to do any special scheduling to insure the I/O operations are flushed properly.

Outline of Thesis

We present here a tentative outline for the thesis resulting from the proposed research.

Introduction. This section presents an introduction to the area of research. Included here will be background and terminology. An overview of the kernel is given to provide the context in which the research was done. The goals and general plan of the research is presented.

Phase One Design. This section details the designs of the file system and virtual memory system. We will present a detailed description of the structures and mechanisms employed. We will also discuss the implementation of the design. The reasoning behind the decisions reached will be presented as well.

Analysis of Phase One. We provide here an analysis of the design and implementation. Included here will be the analysis of the conditions under which atomic writes are required by the recovery protocols. The correctness and general efficiency of the design will be discussed.

Phase Two design. The modifications to the phase one design are presented. We discussed the motives behind such modifications and examine how the modifications are integrated into the original design

Related Work. We compare the results of our research with other similar work.

Further Research. We present other research that might grow from this work. These may include ideas too ambitious to be included in phase two.

Conclusion. A summary of the work done is presented along with some conclusions about the research.

Bibliography

[Allchin83]Allchin, Jim, An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1983.

[Gray79]Gray, J. N., "Notes on Data Base Operating Systems." Operating Systems: An Advanced Course, Ed. by R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, Germany, 393-481, 1979.

[Gray81]Gray, J. N., Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tim Price, Franco Putzolu, Irving Traiger, "The Recovery Manager of the System R Database Manager." Computing Surveys, Vol. 13, No. 2, pp. 223-242, June, 1981.

[Lindsay84]Lindsay, Bruce, Laura Haas, C. Mohen, Paul F. Wilms, Robert A. Yost, "Computation and Communication in R*: A Distributed Database Manager." ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 24-38, February, 1984.

[McKendry84]McKendry, Martin, Ordering Actions for Visibility, Technical Report GIT-ICS-84/05, Georgia Institute of Technology, Atlanta, Georgia, 1984.

[Redell80]Redell, D., "Pilot: An Operating System for a Personal Computer." Communications of the ACM, Vol. 23, No. 2, pp. 81-92, February, 1980.

[Spafford84]Spafford, Eugene, Martin Mckendry, Kernel Structures for Clouds, Technical Report GIT-ICS-84/09, Georgia Institute of Technology, Atlanta, Georgia, 1984.

[Stonebraker81]Stonebraker, Michael, "Operating System Support for Database Management." Communications of the ACM, Vol. 24, No. 7, pp. 412-417, July, 1981.

[Verhofstad78]Verhofstad, J. S. M., "Recovery Techniques for Database Systems." Computing Surveys, Vol. 10, No. 2, pp. 167-196, June, 1978.

[Wallace83]Wallace, J., "DMERT: A Crash Resistant File System." Software - Practice and Experience, Vol. 13, No. 4, pp. 385-387, April, 1983.

Appendix B

Systems Programming with Objects and Actions

Richard J. LeBlanc and C. Thomas Wilkes

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

The goal of the *Clouds* project at Georgia Tech is the implementation of a fault-tolerant distributed operating system based on the notions of *objects* and *actions*, which will provide an environment for the construction of reliable applications. As part of the *Clouds* project, we are designing and implementing a high-level language in which those levels of the *Clouds* system above the kernel level will be implemented. The *Aeolus* language provides access to the synchronization and recovery features of *Clouds*. It also provides a framework with which to study programming methodologies suitable for action-object systems such as *Clouds*.

1. INTRODUCTION

In recent years, numerous research groups have been investigating the potentials of distributed computing systems. Among the benefits promised by advocates of distributed computing are improvements in system fault tolerance and reliability, increased availability of data and services, and faster response through use of distributed programs. Interest in reliability has grown as distributed systems have come to be used in an ever widening set of applications, including critical control systems. In the past, fault tolerance has principally been the concern of hardware designers, who mainly used redundancy as a solution. More recently, it has been realized that maintaining the integrity of distributed data is a crucial concern in providing the benefits listed above. Accordingly, there has been a growth in research interest in techniques for providing the required data integrity in the presence of hardware failures and concurrently executing processes.

Early attempts to provide tools to develop such systems have taken a variety of approaches. The Eden system, developed at the University of Washington, is an "integrated distributed" computing system which is built on support for object-based programming [Alme83]. The principal feature of Eden for supporting fault tolerance is the capability to checkpoint objects. The Pronet language project, a previous effort at Georgia Tech, reflects a distinctly different basic approach [Macc82a, Macc82b]. Pronet allows distributed programs to be constructed of processes which interact via messages transmitted through ports. Fault tolerance is supported through features which allow processes to be checkpointed and networks of processes to be reconfigured. It is interesting to note that even though Eden and Pronet are quite different, they both rely on checkpointing as a crucial feature.

While checkpointing can be used to construct processes or objects with an acceptable degree of fault tolerance, it has an essential weakness in that it is oriented toward dealing with a single object or a single process at a time. Maintaining data integrity in the presence of interactions between processes or objects is the essential question and checkpointing does not deal well with such interactions. Thus a number of researchers have recently proposed that reliability in a distributed system be based on *atomic actions*, an extension of the transaction concept used in distributed database work. Work in this area includes the Argus project at MIT [Weih83, Lisk83], the *Clouds* project at Georgia Tech [Allc82, Allc83a, Allc83b], the Archons project [Jens82] and Spector's work [Schw83, Dani83] at CMU, the Enchere system in France [Bana83] and the work of Birman and Skeen at Cornell [Birm84], among others.

2. OVERVIEW OF THE CLOUDS PROJECT

The goal of the *Clouds* project is to allow the construction of reliable application systems on unreliable hardware. The basic approach is to exploit

the redundancy available in distributed systems which consist of multiple computers connected by high-speed local area networks. We refer to such a system as a *multicomputer* or *computer cluster*. We use the notion of an *object* to represent system components, such as directories or queues. A set of changes to objects is grouped into an *action*. The underlying support system ensures that, even if the objects extend across multiple machines, the changes will occur in totality or not at all. At this level, the support system, known as the *Clouds kernel*, is maintaining the consistency of the objects. It is ensuring that objects either reflect the effects of an action totally or not at all -- no intermediate states are possible. This guarantee of an action's totality permits us to characterize the effects of hardware component failures: they cause actions to fail. Since a failed action is guaranteed to have had no effects on the objects with which it interacted, we can restart the action without concern for potential inconsistencies it might have created.

The actions in Clouds go beyond the related notion of transactions in a database system. Rather than modelling all access to objects as simple reads or writes, the Clouds approach supports arbitrary operations on objects and allows a programmer to take advantage of operation semantics to increase concurrency, and thereby performance. Through appropriate use of encapsulation, concurrent actions can be allowed to change objects without breaching serializability. Even breaches to serializability can be allowed, when it is semantically appropriate and it is necessary to improve performance.

The capabilities of Clouds described thus far comprise an architecture which will support an operating system for a multicomputer. This structure contains three major components: objects, actions and processes.

Objects

An object consists of data and a set of operations (procedures to operate) on that data which maintain a set of associated invariants. For example a queue object might contain the actual queue as data. Operations might be GET to remove an entry and PUT to add an entry to the queue. A typical invariant for a queue might be that no more entries will be removed from the queue than have been added to it. Maintaining such an invariant involves using synchronization mechanisms to delay processes that attempt to remove entries before they are entered. Only the operations on an object can manipulate the data of the object. This property of *encapsulation* assists in maintaining invariants. We exploit encapsulation in our system to incorporate recovery. By making critical objects recoverable, we can guarantee the consistency of the system after failures. We can restart computations on other nodes without fear of partially completed computations disrupting consistency.

Actions

An action is a unit of work. During execution, an action evolves as a partial order of operations on objects. An action appears to be primitive to its surrounding environment. In particular, an action appears to be *atomic* to other actions. Once begun, an action either completes by *committing* or fails by *aborting*. If an action aborts, it has no effect on its environment. This is achieved through recovery mechanisms which maintain the state of the objects involved in aborted actions. Actions fail when they interact illegally with other actions (e.g., deadlock) or when they are explicitly aborted. In our implementation, actions can be *nested* to improve failure containment characteristics.

Processes

In the Clouds architecture, objects are passive. Thus processes are used to provide activity in the system. A process may be used to represent a single top-level action or a nested action, in which case it will terminate when the action completes.

Thus objects, actions and processes are fundamental concepts supported by the architecture. To support these concepts, recovery and consistency are incorporated into the basic virtual memory mechanism. Synchronization mechanisms to control the interactions of actions are also provided. It is with these capabilities that Clouds is meant to support the data integrity required for the implementation of reliable, distributed application programs.

The mechanisms developed for the support of transactions in database systems, as well as the traditional operating system synchronization mechanisms, have been found to be insufficient for the support of the action-object approach in operating systems. In particular, the problems of ordering and atomicity for nested actions, and several simplifications which apply to many operating systems problems, are discussed in [McKe84]. The expediciencies made possible by these simplifications make the use of the action-object approach in the Clouds system viable.

In addition to the basic action support, other mechanisms are needed to help ensure that a system tolerates faults such as component failures. One requirement is a representation of *work*. Work is represented in Clouds by *action networks*, which are described using a Petri-net notation; transitions in the Petri-nets correspond to action executions. A state of such a net is called a *job*. A second requirement is that continuity of job execution be maintained through failures. Thus, we use *job schedulers* to assign actions to machines. A *primary scheduler* may be supported by a set of *backup schedulers*, which may reside on different machines, to provide backup if a coordinator fails during action commit.

These mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects (*servers*) which in combination provide a reliable environment for applications.

3. THE NEED FOR AN ACTION-BASED PROGRAMMING METHODOLOGY

Actions are the key feature for guaranteeing data consistency. The "all-or-nothing" nature of actions really solves two problems. When an action fails, its effects are automatically undone; so, actions which fail due to machine failures cannot leave objects in an inconsistent state. Additionally, the required serializability of actions provides a coarse-grained synchronization among them. (Other features may be used to provide more concurrency by supporting synchronization at a lower level.) Actions which are aborted for logical reasons (e.g., deadlock) again can have no visible effects on the state of any object. Thus the action concept successfully broadens the recovery viewpoint provided by checkpoints, since it encompasses all the changes to any number of objects made by an arbitrarily complex action.

Actions alone do not provide all of the generally desired capabilities, since they do not address the question of the resiliency of individual objects. That is, they do not contribute toward the recovery of objects located on machines that fail. Rather, they guarantee the integrity of surviving objects. Both Argus and Clouds support resilience through use of *stable storage*. (Stable storage has the property that information entrusted to it is extremely unlikely to be lost.) Various features are provided which cause the object support system to record sufficient information on stable storage so that the state of an object (*guardian* in Argus) may be recovered after a hardware failure. Note that for a combination of consistency and resilience, the state of an object must be written to stable storage whenever an action which modified the object commits (presuming that pessimistic recovery is being used).

Writing the state of an object to stable storage is, of course, just checkpointing. Indeed, the concept of stable storage is an implicit or explicit part of any checkpointing scheme (i.e., Eden and Pronet). It is the coupling with the action mechanism which makes checkpointing of objects effective. That is, part of the implementation of a commit is a checkpoint of all affected objects. Thus checkpointing is made an effective means for providing consistent, resilient objects.

The mechanism for specifying just what must be written is one way in which Argus, Clouds and other proposals differ. In Argus, all *mutex* objects within a guardian are written. As suggested by the name, *mutex* objects also have certain synchronization properties, relating to their accessibility to concurrently executing actions. Clouds, on the other hand, allows an entire object or any data object within it to be specified as *recoverable*. As would be expected, if the entire object is recoverable, then all of its contained data objects are written to stable storage when a relevant commit occurs. Both of these approaches exemplify implicit specifications of what must be

saved for recovery purposes. Yet another approach would be to require the programmer who defines an object to provide an explicit *write-to-stable-storage* operation to be invoked by the object management system at appropriate times. This variety of proposals reflects the need for study of a programming methodology based on use of objects and actions, so that we can determine just what kinds of features are most effective.

The Clouds architecture goes beyond others in that it can support actions that involve objects on more than one machine. In other words, a remote procedure call can be done without creation of a nested action. Allchin's work [Allc83b] provides a definition of the basic capabilities supported by the Clouds architecture and a design for their implementation. Now that that implementation is in progress, we are studying how these capabilities may be applied. In particular, we wish to study a programming methodology for systems like Clouds.

As part of the Clouds project, we are designing and implementing high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology). An overview of the *Aeolus* language is presented in the next section. *Aeolus* gives the programmer access to the features of the Clouds system discussed above. However, we also intend to use *Aeolus* as a framework for studying the sort of programming methodology appropriate to Clouds. This study should lead to the design of high-level language features to support that methodology.

4. OVERVIEW OF AEOLUS

The major design goal of *Aeolus* is to make possible access to the features of the Clouds system from a powerful systems programming language which supplies those features -- such as strong typing -- which aid in the quick development of error-free programs, yet allows those features to be explicitly circumvented when necessary.

The major structuring features in *Aeolus* are processes and objects. Objects have two purposes in *Aeolus*: to provide support for data abstraction, and to reflect the recoverability and synchronization capabilities provided by the Clouds kernel. It has been argued elsewhere [Allc82] that the object construct provides a powerful tool for the organization of programs for recovery, both from the standpoint of the programmer and of the system. Objects may rely on the automatic operating system / runtime system support for synchronization and recovery (*recoverable* and *autosynch* objects). Alternatively, using powerful features provided by the language and the Clouds system, the programmer may take advantage of semantic knowledge about the application to explicitly code more appropriate recoverability and synchronization. However, *Aeolus* objects also provide abstraction features even when synchronization and recovery are not required. These *nonrecoverable* objects provide a logical framework for the organization of modules for separate compilation. Example 1 shows the structural outline of the COM_Q nonrecoverable object, which provides buffered access to the asynchronous communications ports of the IBM Personal Computer. (This example has been elided for reasons of space, especially concerning the text of its operations. A sample operation is shown in example 3.)

4.1. Features for Systems Programming

In keeping with its purpose as a systems programming language, *Aeolus* incorporates several features which give the programmer access to the hardware and the lower levels of the systems software, as well as "convenience" features which allow more efficient coding, including:

- a full range of assignment and bit-manipulation operators similar to those in the C language;

- features for register optimization, such as a special *index* type for loop counters and array references;

- the option of specifying *inline* expansion of a procedure;

- a facility for specifying *arbitrary* procedure argument lists of unspeci-

fied length and (predefined) types (similar to the *nospread* arglists of Interlisp);

and the ability to specify storage addresses for variables, as well as some capabilities for setting and doing arithmetic on pointers.

However, most of the power of Aeolus as a systems programming language, aside from the access it provides to the features of the Clouds system, lies in the ability it gives the programmer to specify low-level data structures as abstract data types, and in the treatment of the underlying hardware as an object with operations on its state available from the language.

In addition to the usual structured types (records and arrays), Aeolus provides a *structure* type, which allows the programmer to specify abstract types for the manipulation of bitfields. The *structure* is similar to the *packed record* construct of Pascal, except that the programmer indicates that its fields should fit one of the addressable entities defined by the target computer (byte, word, doubleword, quadword, etc.), and this correspondence is checked by the compiler. This provides a secure mechanism allowing bit fields within a low-level data structure to be referenced by name. Several examples of abstract structured byte types are given in example 1. Aeolus also provides the *byte* and *word* types as predefined objects. These objects have operations permitting manipulations similar to those of the *bitset* type of Modula-2. The programmer may define similar objects for bit strings of other lengths.

The ability to inspect and change the state of the hardware is also important in systems programming. Access to the underlying hardware is provided by the operations of special Aeolus objects. We call such an object a *pseudo-object* since only one instance of it may exist, whereas there may be an arbitrary number of instances of a normal object. An example of a pseudo-object is *PC_System*, which is used in the implementation of the *COM_Q* object in example 1. This pseudo-object gives access to the registers and ports of a PC's microprocessor, and through the ports to the other system components, such as the interrupt controller, device controllers, and modem registers. For example, the *IN_BYTE* and *OUT_BYTE* operations of *PC_System* allow values to be input and output from the byte ports of a PC; other *PC_System* operations provide such capabilities as access to the register set, flags, and interrupt mechanism. These operations typically compile inline to a single machine instruction. For considerations of efficiency, some operations in hardware pseudo-objects may give access to special instructions of the target machine, such as the string instructions of the PC or the polynomial instructions of the VAX. Example 2 shows how *PC_System* may be used to program an asynchronous communications interrupt service routine (ISR), a local procedure of the *COM_Q* object. This interrupt handler is invoked when a character is received over the communications line, and enqueues the character for later processing by the *RECEIVE* operation of the *COM_Q* object.

The operations of *PC_System* are also used in the *PUT_BYTE* operation of the *COM_Q* object, the text of which is given in example 3. This procedure waits for several conditions concerning the state of the communications line to become true (unless a timeout occurs first), and then places a data byte on the line. The line state is checked by interrogating the registers of the communications controller via port input operations, and inspecting bit fields of the register-type values obtained.

4.2. Features for Object and Action Programming

The design of Aeolus is intended to support the recovery and synchronization capabilities of the Clouds system in a high-level systems programming language. Objects in Aeolus, besides providing an organizational tool for secure separate compilation, give access to the recovery properties of Clouds objects. Thus, unless an Aeolus object is designated as *nonrecoverable*, the Clouds kernel mechanisms are used for invocations of its operations, allowing the system to control the recoverability properties of the object's state. Examples of the implementation of an *nonrecoverable* object have been shown previously. In Section IV, we shall present the development of an object which uses the system mechanisms for recovery and synchronization. In the remainder of this section, the features provided by Aeolus for accessing these features of Clouds are examined.

The code for an Aeolus object has two parts. The *definition* part is seen both by the object itself when it is being compiled, and by all other objects or programs which use that object. Compilation of a definition part produces a symbol table file which is used for type checking among these separate compilations. It can contain specifications of public types and constants defined by the object, and the interface definitions of the object's operations. Definition parts may not contain variable declarations. The *implementation* part contains the actual code of the operations, along with any needed local (private) type, constant, or procedure definitions. Local variables of an object share the lifetime of the object instance to which they belong, and thus act as "own" variables. This separation of definition and implementation provides a safe separate compilation mechanism similar to *packages* in Ada (TM) or *modules* in Modula-2.

The general syntax of object implementation parts in Aeolus is shown in example 4. (Although not shown in this example syntax, objects may be specified as being *autosynch* and *recoverable* simultaneously.) If the object is specified as being *nonrecoverable*, it is treated as being simply a separate compilation module. That is, operations in *nonrecoverable* objects are compiled using the standard preludes and postludes for procedure bodies, without special code or system calls for recovery. If the object is specified as being *recoverable*, the compiler provides a standard run-time framework for recovery by generating preludes and postludes for the object operations using Clouds object and action manager calls. Thus, the programmer may gain access to the action mechanisms of the Clouds system with a single keyword. However, the full power of the Clouds action mechanisms may be unnecessary and inefficient in some cases. For those cases, the Aeolus/Clouds system provides mechanisms which allow the user to explicitly program recovery strategies tailored to the individual requirements of the problem at hand. Therefore, if neither the *nonrecoverable* nor the *recoverable* keyword is given in an object header, it is assumed that object recovery is explicitly programmed. In this case, the programmer may provide alternate recovery procedures for recoverable variables of the object, and may also specify, in the *action events* clause, handlers other than the default system handlers for the precommit, commit, and abort events of the entire object. The compiler then specifies to the action and object management systems that, when one of the action events occurs, these alternate handlers are to be invoked instead of the standard, system-provided procedures.

The Aeolus language also provides access to the synchronization mechanisms of the Clouds system. When the *autosynch* object attribute is specified in an object header, it indicates that the default system synchronization procedures are to be used on the object's operations to provide concurrency atomicity. If the *autosynch* attribute is not specified, synchronization may be explicitly programmed using operations on the *lock* type provided by the language. A Clouds lock [Allc83b] is not associated with a physical object, but rather with values in the domain of the object. Thus -- for example -- a file name may be locked, even if a physical file with that name does not yet exist. The examples in the next section demonstrate the use of locks.

The *uses* clause allows the programmer to specify the use of system pseudo-objects, while the *import* clause allows other user-defined or system-defined object definitions to be accessed. In a <block>, definitions of types, constants, variables, recoverable variables, internal procedures, and operations may be written in any order (as long as their definitions appear before any uses); the <statement part> of the block is treated as an initialization routine to be executed upon creation of an instance of the object.

Object operations are programmed like procedures. An operation invocation looks like a procedure invocation with a prefix indicating the object instance upon which to operate:

```
<object instance id> @ <operation id> ( <actual param list> )
```

An object instance may be created by declaring a variable of that object type, and then allocating the instance's data storage on the heap using an extended version of the allocation function, or by associating the variable with a "permanent" object, much as a file variable can be associated with a physical file in Pascal.

Operations or local procedures of (recoverable) Aeolus objects may be specified to be invocable as an action. The syntax of action implementations is

much like that of procedures:

```
procedure <proc id> ( <formal param list> ) is action
  <procedure block>
end procedure
```

(A <procedure block> is the same as a <block> except that it cannot contain declarations of recoverable variables.) Thus, the invocation of an action is similar to a procedure invocation; however, a unique *action-id* is created by a Clouds action manager for the invocation, which may be assigned to a variable of the invoking procedure:

```
<action-id var> := action <proc id> ( <actual param list> )
```

This *action-id* variable may be used to retrieve information from the system about the status of the action, or to abort the action, using calls to a Clouds action manager. This mechanism allows general control structures to be formulated, e.g., for the concurrent invocation of actions.

Example 1. Declarations and initialization of the COM_Q object

definition of object com_q is

```
! The definition part of the COM_Q object, which
! provides buffered read/write access to either of the IBM PC serial
! ports, COM1 or COM2. Reception from the port is interrupt-driven.
! The nomenclature of the PC Technical Reference Manual is used here.
! The definition part specifies the publicly available constants, types,
! and operation interfaces of the object to those objects which import it.
```

```
type status_type is
  (normal,      init_error,   receive_error,  buffer_overflow,
   DSR_timeout, CTS_timeout,  THRE_timeout  )
```

```
operations
  procedure init (baud_rate : 300 .. 9600, data_bits : 5 .. 8,
                 stop_bits : 1 .. 2, parity : (odd, even, none),
                 var error : boolean) is modify
  procedure finish () is modify
  procedure status () : status_type is examine
  procedure get_byte (var in_data : byte,
                    var received : boolean) is examine
  procedure put_byte (out_data : byte, var error : boolean) is modify
```

end definition.

implementation of object com_q (port: word) is

```
! The implementation part of the COM_Q object.
! The operations specified in the definition part are actually implemented
! here, and any other local constants, types, variables, or procedures are
! specified.
```

uses PC_System ! The PC CPU pseudo-object.

import queue

```
! The definition part of the QUEUE object is included here as comments for
! clarity.
```

```
! definition of object queue (elem_type : private, size : integer) is
!   operations
!     procedure enqueue (item : elem_type, var full : boolean) is modify
!     procedure dequeue (var item : elem_type,
!                      var empty : boolean) is modify
! end definition.
```

```
asynch_int : const word := 16#???? ! location of the asynch. int. vector
IMR        : const word := 16#21   ! Interrupt Mask Register n
```

! Addresses of some important register ports on the IBM Asynch board.

```
.....
MCR      : const word := 16#3fc    ! Modem Control Register
LSR      : const word := 16#3fd    ! Line Status Register
MSR      : const word := 16#3fe    ! Modem Status Register
```

```

! Define the internal structures of some of the above registers.
! This includes both single bits and bit fields.
! Fields are named from most significant to least significant bit.

```

```

....
type LSR_struct is
  structured byte
    unused,           ! = 0
    TSRE,            ! Transmitter Shift Register Empty
    THRE,            ! Transmitter Holding Register Empty
    BI,              ! Break Interrupt
    FE,              ! Framing Error
    PE,              ! Parity Error
    ORun,            ! OverRun error
    DR               : boolean ! Data Ready
  end structure

```

```

....
type MCR_struct is
  structured byte
    unused           : 0 .. 7 ! 3 bits, =0
    LOOPback,
    OUT2,
    OUT1,
    RTS,             ! Request To Send
    DTR              : boolean ! Data Terminal Ready
  end structure

```

```

type MSR_struct is
  structured byte
    RLSD,            ! Receive Line Signal Detect
    RI,              ! Ring Indicator
    DSR,             ! Data Set Ready
    CTS,             ! Clear To Send
    DRLSD,           ! Delta RLSD
    TERI,            ! Trailing Edge Ring Indicator
    DDSR,            ! Delta DSR
    DCTS             : boolean ! Delta CTS
  end structure

```

```

! Queue for buffering of characters (data bytes) input from the serial port.
! Currently allows buffering of up to 128 characters.

```

```

in_q      : queue (byte, 128)

```

```

! The current status of the COM_Q object, and variables to save information
! from the serial port registers about what went wrong.

```

```

cstatus   : status_type := normal
LSR_save  : LSR_struct := 0

```

```

....
! Old state of interrupt vectors and modem control register.

```

```

old_rs232_isrvc : doubleword
old_MCR         : MCR_struct

```

```

....
begin      ! Initialization section
  new (in_q)           ! Create instance of the QUEUE object
  old_MCR := in_byte (MCR)
  init (1200, 8, 1, none, error) ! Standard parameters at first
  if error then
    status := init_error
  else
    init_isrvc (asynch_int, rs232_isr, old_rs232_isrvc) ! Set int. vector
    out_byte (IMR, 16#ac) ! Enable diskette, comm, keyboard, timer
    out_byte (MCR, 2#00001000) ! Enable OUT2 in modem control register
  end if
end implementation.

```

Example 2. Interrupt service routine of the COM_Q object

procedure rs232_isr () is

```
! The interrupt service routine handles the Data Ready interrupts
! for input from the serial port. The character is placed in
! the input queue.

! The LSR_errors constant definition provides an example of the use of
! constructors. This constant represents error conditions in the
! line status register for which we're on the lookout.
! This constant could also have been written as
!   const byte := 2#00011010
! depending on the programmer's taste.

! Note that, although interrupts are disabled by the hardware when
! the ISR is invoked, we must explicitly re-enable interrupts when
! we're done.
```

```
LSR_errors : const LSR_struct
:= LSR_struct["
```

```
FALSE : 3,
TRUE,   ! break interrupt
TRUE,   ! framing error
FALSE,
TRUE,   ! overrun error
FALSE
]
```

```
LSR_val      : LSR_struct
buffer_full  : boolean
```

```
begin
```

```
LSR_val := in_byte (LSR)           ! Get a byte from the LSR port
if LSR_val & LSR_errors then       ! Test line status reg. for errors
  cstatus := receive_error
  LSR_save := LSR_val
else
  ! Enqueue byte from the receiver reg.
  in_q & enqueue (in_byte (RBR), buffer_full)
  if buffer_full then
    cstatus := buffer_overflow
  end if
end if
enable ()                          ! Re-enable interrupts (PC_System)
```

```
end procedure ! rs232_isr !
```

Example 3. The PUT_BYTE operation of the COM_Q object

```
procedure put_byte ( ! out_data : byte, var error : boolean ! ) is
    ! Send a data byte to the serial port. If an error such as timeout
    ! has occurred, set the boolean variable "error".

MCR_init : const MCR_struct := 2#00001011    ! Set OUT2, RTS, DTR (as byte)
MSR_val  : MSR_struct
timeout  : const integer := 10
count    : integer := 0

begin
    out_byte (MCR, MCR_init)           ! Initialize the Modem Control Register
    while count <= timeout loop       ! Wait for Data Set Ready
        MSR_val := in_byte (MSR)
        if MSR_val.DSR then
            exit .
        end if
        count += 1
    end loop
    if count > timeout then
        status := DSR_timeout
        MSR_save := MSR_val
        error := TRUE
        return .
    end if

    count := 0
    while count <= timeout loop       ! Wait for Clear To Send
        MSR_val := in_byte (MSR)
        if MSR_val.CTS then
            exit .
        end if
        count += 1
    end loop
    if count > timeout then
        status := CTS_timeout
        MSR_save := MSR_val
        error := TRUE
        return .
    end if

    count := 0
    loop                               ! Wait for Transmit Holding Reg. Empty
        LSR_val := in_byte (LSR)
        if LSR_val.THRE then
            exit .
        elsif count > timeout then
            status := THRE_timeout
            LSR_save := LSR_val
            error := TRUE
            return .
        end if
        count += 1
    end loop

    out_byte (THR, out_data)         ! FINALLY send the data byte
end procedure ! put_byte !
```

Example 4. Syntax of Aeolus object implementations

```
implementation of [ nonrecoverable | recoverable | autosynch | epsilon ]
    object <object id> is
    uses <id list>
    import <id list>
    action events <override list>
    <block>
end implementation.
```

5. PROGRAMMING ACTIONS IN AEOLUS

In this section we present an example Aeolus object which illustrates the use of some of the language features which Aeolus provides for access to the action management facilities of the Clouds system. The SYMTAB object implements a simple symbol table, which uses the action mechanism to provide recovery "firewalls" around its critical operations, and uses the Aeolus/Clouds lock mechanism to specify customized synchronization rules which allow a high degree of concurrency in the use of its operations. For simplicity, the version of the SYMTAB object shown here maintains only a single copy of its state; more advanced Aeolus programming techniques will allow implementation of multiple-copy objects for availability purposes, without necessitating changes in the object interface.

The "definition part" or interface of the SYMTAB object is shown in example 5. An operation definition, as may be seen in this example, may specify (with the keyword *action*) that the operation is to be compiled so that its invocation will automatically result in the creation of an action to encapsulate its execution. (The newly created action will actually be a subaction of the action which invokes the operation. Thus the execution of the operation can be aborted without necessarily terminating the calling action.) All operations of the SYMTAB object are invoked as actions, except for the QUICK_LIST operation (more on this later).

An operation definition may also indicate that the operation might *modify* (write to) or *examine* (read) the object state; this information is used in the compilation of objects which take advantage of the automatic synchronization (*autosynch*) capabilities supported by the Clouds kernel. As would be expected, multiple concurrently executing actions are allowed to access an *autosynch* object via *examine* operations. On the other hand, an action may not execute a *modify* operation on such an object until all other actions which have touched the object have either committed or aborted, and that action will lock out all others until it commits or aborts.

SYMTAB does not use the *autosynch* feature; rather, it uses synchronization techniques which allow greater concurrency where possible. For example, among the operations defined in the SYMTAB object are two which provide listings of the object state. The QUICK_LIST operation provides only an approximate picture of the symbol table state, since it does not wait for any actions which have executed INSERT and DELETE operations on the symbol table to complete before it produces a listing. It thus can always be executed without waiting, but it effectively assumes that all changes which have been made to the working copy of the symbol table will eventually be committed. The picture of the symbol table provided by QUICK_LIST may not be and may never become a committed state. A precise picture is given by the EXACT_LIST operation, which is executed as an action. EXACT_LIST uses a lock (described below) to guarantee that all other actions which have changed the symbol table either commit or abort before the listing is produced. Thus the picture presented by this operation presents a valid logical view of the symbol table. Presumably, the user of the QUICK_LIST operation is willing to risk the possibility of an inconsistent picture in exchange for the greater speed of this operation.

The implementation of the SYMTAB object is shown in example 6. The object defines two locks for synchronizing concurrent use of its operations. The

SYMENTRY lock is used to lock individual hash buckets of the symbol table hash array, allowing a typical multiple reader / single writer protocol. This protocol is used by the INSERT and DELETE operations to exclude multiple writers on a given hash bucket. However, the granularity of this lock allows multiple writers concurrent access to disjoint buckets. This granularity of locking was, of course, selected on the basis of knowledge of the particular data structures being used to implement the hash table.

The SYMTABLE lock, on the other hand, is used to lock the entire hash array. This lock is unusual in that it allows multiple writers as well as multiple readers, although writers exclude readers and vice versa. The SYMTABLE lock is used to express the incompatibility between the EXACT_LIST operation and the INSERT and DELETE operations, since the latter two modify the symbol table state. Such a lock is a very good example of the tailoring of synchronization constraints allowed by the Aeolus lock feature.

Example 5. SYMTAB object definition part

definition of object symtab is

```
! Single-copy symbol table object using the action management
! facilities of Aeolus/Clouds for recovery firewalls and the lock
! mechanisms for synchronization.
!
! The definition part contains specifications of public constants,
! types, and operations defined by this object.
! When compiled, it produces a symbol table file which may be imported
! by other objects using this object in their implementations.
```

```
MAX_VAL_LENGTH : const integer := 80      ! or whatever
```

```
type valstring is string (MAX_VAL_LENGTH)
```

operations

```
procedure insert ( newname : valstring ) is modify action
! The INSERT operation must be invoked as an action.
! It places an entry into the symbol table,
! and locks the NEWNAME entry before the insertion.

procedure delete ( oldname : valstring ) is modify action
! The DELETE operation must be invoked as an action.
! If it finds an entry with value field = OLDNAME, it locks that
! entry and then removes the entry from the symbol table and frees
! its storage space.

procedure find ( name : valstring ) : boolean is examine action
! The FIND operation must be invoked as an action.
! It sets a READ lock on the NAME entry, and then tries to locate
! that entry with value field = NAME and returns TRUE if it succeeds.

procedure quick_list () is examine
! The QUICK_LIST operation provides a quick (dirty) listing of all
! names currently in the symbol table.

procedure exact_list () is examine action
! The EXACT_LIST operation must be invoked as an action.
! It provides a listing of the exact state of the symbol table at a
! given point in time. To do this, it locks the whole symbol table,
! thereby excluding any changes during preparation of the listing.
! Thus, although EXACT_LIST, FIND, and QUICK_LIST operations
! may execute concurrently, and INSERT and DELETE operations
! which access different hash buckets may also execute
! concurrently, INSERT and DELETE operations must block on
! EXACT_LIST operations.
```

end definition.

Example 6. SYMTAB object implementation part

implementation of object symtab is

```
! Single-copy symbol table object using the action management
! facilities of Aeolus/Clouds for recovery firewalls and the lock
! mechanisms for synchronization.
```

```
MAXBUCKET : const integer := 101    ! or whatever
```

```
type hash_range is 1 .. MAXBUCKET
```

```
type ptr_entry is -> symtable_entry
```

```
type symtable_entry is                ! just something for demo purposes
  record
    name      : valstring ,
    next      : ptr_entry
  end record
```

```
symtable      : array [hash_range] of ptr_entry
```

```
symentry_lock : lock ( write : [] ,
                      read  : [read] ) domain is hash_range
! The SYMENTRY lock allows locking of individual hash buckets in the
! symbol table. Several READ operations are allowed to proceed
! concurrently, but a WRITE operation blocks all other operations.
```

```
symtable_lock : lock ( write : [write] ,
                      read  : [read] )
! The SYMTABLE lock allows the entire symbol table to be locked.
! This lock is set in the EXACT_LIST operation for purposes of
! getting an exact listing of the state of the symbol table.
! Operations which change the state of the symbol table must wait for
! completion of any outstanding EXACT_LIST operations.
```

```
procedure hash ( name : valstring ) : hash_range is
! This HASH function is a local (nonpublic) procedure of
! the SYMTAB object.
begin
! the usual type of stuff
end procedure ! hash !
```

```
procedure insert (! newname : valstring !) is action
! The INSERT operation must be invoked as an action.
! It places an entry into the symbol table,
! and locks the NEWNAME entry before the insertion.
```

```
entry      : ptr_entry
bucket_num : hash_range
```

```
begin
  SetLock (symtable_lock, 0, write)
  bucket_num := hash (newname)
  new (entry)
  using ent := entry -> do
    ent.name := newname
    ent.next := symtable [bucket_num] -> .next
  end using
  SetLock (symentry_lock, bucket_num, write)
  region symtable [bucket_num] do
    symtable [bucket_num] := entry
  end region
end procedure ! insert !
```

```

procedure delete (! oldname: valstring !) is action
! The DELETE operation must be invoked as an action.
! If it finds an entry with value field = OLDNAME, it locks tht
! entry and then removes the entry from the symbol table and frees
! its storage space.

```

```

entry, preventry : ptr_entry
bucket_num       : hash_range

begin
  SetLock (symtable_lock, 0, write)
  bucket_num := hash (oldname)
  entry, preventry := symtable [bucket_num]
  while entry <> NIL loop
    if entry -> .name = oldname then
      SetLock (symentry_lock, bucket_num, write)
      region entry do
        preventry -> .next := entry -> .next
        dispose (entry)
      end region
      exit .
    else
      preventry := entry
      entry := entry -> .next
    end if
  end loop
end procedure ! delete !

```

```

procedure find (! name : valstring !) ! : boolean ! is action
! The FIND operation must be invoked as an action.
! It sets a READ lock on the NAME entry, and then tries to locate
! that entry with value field = NAME and returns TRUE if it succeeds.

```

```

entry : ptr_entry
bucket_num : hash_range

begin
  bucket_num := hash (name)
  SetLock (symentry_lock, bucket_num, read)
  entry := symtable [bucket_num]
  while entry <> NIL loop
    if entry -> .name = name then
      return TRUE
    else
      entry := entry -> .next
    end if
  end loop
  return FALSE ! if we get here, NAME isn't in the symbol table
end procedure ! find !

```

```

procedure quick_list () is
! The QUICK_LIST operation provides a quick (dirty) listing of
! names currently in the symbol table.

```

```

entry : ptr_entry
i : index hash_range

begin
  for i := 1 to MAXBUCKET loop
    entry := symtable [i]
    while entry <> NIL loop
      write (entry -> .name) ! or whatever
      entry := entry -> .next
    end loop ! while !
  end loop ! for !
end procedure ! list !

```

```

procedure exact_list () is action
  ! The EXACT_LIST operation must be invoked as an action.
  ! It provides a listing of the exact state of the symbol table at a
  ! given point in time. To do this, it locks the whole symbol table,
  ! thereby excluding any changes during preparation of the listing.
  ! Thus, although EXACT_LIST, FIND, and QUICK_LIST operations
  ! may execute concurrently, and INSERT and DELETE operations
  ! which access different hash buckets may also execute
  ! concurrently, INSERT and DELETE operations must block on
  ! EXACT_LIST operations.
begin
  SetLock (symtable_lock, 0, read)
  quick_list ()
end procedure ! exact_list !

```

```

i : index hash_range

```

```

begin ! initialization
  for i := 1 to MAXBUCKET loop      ! symbol table is initially empty
    symtable [i] := NIL
  end loop
end implementation.

```

6. CONCLUSIONS AND FUTURE WORK

We have found Aeolus to be quite effective as a systems programming language (as represented by examples 1 through 3). In particular, the clarity of interface definitions made possible by use of pseudo-objects is extremely valuable for encapsulation of hardware details. Through our experience with developing objects like SYMTAB (examples 5 and 6), we have come to understand techniques for using subactions as "firewalls" to limit the effect of failures. We have found that Allchin's generalized lock mechanism makes it relatively easy to specify special-purpose synchronization rules dependent on object semantics (e.g., the use of the SYMTABLE lock in SYMTAB).

Among the hardest questions which need more study is how replication can most effectively be used to provide availability. Actions and resilient objects ensure that failures are not catastrophic, but they are concerned with data integrity, not with how a program reacts to failures. The availability question involves use of multiple objects on different nodes to represent a single resource, thus providing continued access to the resource in the presence of individual node failures. Algorithms for read and write access to such resources must be developed and evaluated. The recent paper by Daniels and Spector [Dani83] is one example of such an algorithm.

We must also consider possible representations of work so that it may be restarted; this is an area that has been until recently unexplored [McKe84]. As has been noted above, most of the work on actions and objects has been oriented toward protection of data from failures. The fact that processes are considered to be an important, independent component supported by the Clouds architecture gives us a point of departure for this study. McKendry's work on Petri-nets discussed above lays the groundwork for an attack on this problem within the framework of Clouds. If we view a program as a collection of processes interacting through shared objects, some features akin to the process interconnection specifications of Pronet [Macc82b] may prove to be useful.

7. REFERENCES

- [Allc82] Allchin, J. E., and M. S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982
- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1983 (also available as technical report GIT-ICS-83/23)
- [Alme83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," Technical Report 83-10-05, Department of Computer Science, University of Washington, October 1983
- [Bana83] Banatre, J. P., M. Banatre, and F. Ployette, "Construction of a Distributed System Supporting Atomic Transactions," Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems, Clearwater Beach, Florida, October 1983
- [Birm84] Birman, K., et al., "Implementing Fault-Tolerant Distributed Objects," Computer Science Technical Report 84-594, Cornell University, March 1984
- [Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [Jens82] Jensen, E. D., "Decentralized Executive Control of Computers," 3rd IEEE International Conference on Distributed Computing Systems, October 1982
- [Lisk83] Liskov, B., and M. Herlihy, "Issues in Process and Communications Structure for Distributed Programs," Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems, Clearwater Beach, Florida, October 1983
- [Macc82a] Maccabe, A. B., "Language Features for Fully Distributed Processing Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1982 (also available as Technical Report GIT-ICS-B2/12)
- [Macc82b] Maccabe, A. B., and R. J. LeBlanc, "The Design of a Programming Language Based On Connectivity Networks," Proceedings of the Third International Conference on Distributed Computing Systems, Miami / Fort Lauderdale, October 1982
- [McKe84] McKendry, M. S., "Ordering Actions for Visibility," Technical Report GIT-ICS-84/05, School of Information and Computer Science, Georgia Institute of Technology, August 1984
- [Schw83] Schwarz, P., and A. Spector, "Synchronizing Shared Abstract Types," Carnegie-Mellon University Technical Report CMU-CS-83-163, Revised November 1983
- [Weih83] Weihl, W., and B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types," Symposium on Programming Language Issues in Software Systems, June 1983

QUARTERLY PROGRESS REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT #MDA 904-84-C-6035
REPORTING PERIOD: 1 JAN 85 - 31 MAR 85

1. Project Status

During the second quarter of this project, work has continued on each of the two tasks called for by the statement of work. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing. Under the Distributed File Systems task, work has proceeded on the design of the Clouds storage manager. The main focus of this work has been the development of a firm interface specification for the storage manager. This effort is summarized below and will be documented in a forthcoming technical report "A Note on Storage Management in Clouds". Under the Language Support for Robust Distributed Programs task, a Ph.D. thesis proposal has been developed, entitled "Programming Methodologies for Resilience and Availability." This proposal is attached as Appendix A. Other work in progress under this task, involving language definition and implementation, is described in section 3.

The work on the tasks of this project is proceeding on schedule. Future plans include a continuation of the two investigations presented here. These efforts, in combination with other work in progress within the Clouds project, should lead to a working prototype system by the end of this year.

2. Storage Management -- Progress Report

This quarter's effort was devoted primarily to refining the design of the storage manager. This effort is documented in the technical report "A Note on Storage Management for Clouds" (GIT-ICS-85/02). We were interested in establishing a firm interface for the storage manager, so that efforts on other parts of the Clouds kernel that interact with the storage manager can proceed. The refined design addresses some problems that with the recovery mechanism that were ignored initially. Some initial coding of data structures has started.

We are taking an object-oriented approach to the design and have identified three object types which form the basis of the storage manager. At the lowest level is the device object type, which is responsible for the uninterpreted transfer of data to and from pages on secondary storage. Most of the functionality at this level is similar to that found in conventional device drivers. The partition object type forms the next level in the storage manager and represents a logical device. Partitions provide a mechanism for

the division of secondary storage according the intended use of the storage (recoverable object storage, non-recoverable object storage, and paging surfaces, for example). The segment object type is an alternate type for Clouds objects. This alternate view of an object allows the storage manager to manipulate object data in a uniform and convenient manner. A segment object is simply a sequence of bytes that can be manipulated by a few simple operations, such as get a page of the segment and get the status of the segment. Most of the storage manager's recovery mechanism is located in objects of this type.

We have defined operations and the major data structures for each of these object types, which collectively form the interface to the storage manager. Most accesses to the storage manager will be through a call on a segment object or some partition object operation; the device object operations form a low-level interface between the partition and segment objects and secondary storage. The segment and partition operation provide a uniform interface to the storage manager for the handling of both recoverable and non-recoverable object.

Other refinements to the design concern the recovery mechanism, particularly the recovery of partition structures such as the partition allocation map. Our concern has been the avoidance of a bottleneck during action commits. To this end, we have developed a scheme using intention lists to avoid locking out large portions of the partition map during action commits, permitting as many actions to commit concurrently as possible. This scheme differs considerably from the recovery mechanism use for object data.

The device object must be able to flush i/o requests during action commit, so to ensure that any committed changes are reflected in the permanent object data. We are in the process of developing a mechanism to allow actions to specify a set of requests which must be flushed to secondary storage before the commit is complete. Our goal with this mechanism is to interfere as little as possible with the normal scheduling of requests by the device object.

The segment level recovery protocols have undergone some polishing to make them more efficient and remove some bugs.

3. Aeolus -- Progress Report

As part of the Clouds project, we are designing and implementing a high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology) in which those levels of the Clouds system above the kernel level will be implemented. The Aeolus language, described in [Wilk85b] (in progress), provides access to the synchronization and recovery features of Clouds. It also provides a framework within which to study programming methodologies suitable for action-object systems such as Clouds. This study should lead to the design of high-level language features to support that methodology. Thus, our interest in Aeolus lies not in the language itself, but in studying the sort of programming which may be done with it.

We have found Aeolus to be effective as a systems programming language during our studies of programming systems objects such as communications handlers for the Clouds workstations. In particular, the clarity of interface definitions made possible by use of pseudo-objects is extremely valuable for encapsulation of hardware details in such hardware-dependent programming. Through our experience with developing systems objects, we have come to understand techniques for using subactions as "firewalls" to limit the effect of failures. We have found that Allchin's generalized lock mechanism makes it relatively easy to specify special-purpose synchronization rules dependent on object semantics.

A compiler for Aeolus is currently under development on one of the DEC VAX 11/750 computers of the Clouds project under Berkeley Unix (TM) Version 4.2. We are using the *Amsterdam Compiler Kit* (ACK) [Tane83] to generate code generators for Aeolus for both the Clouds VAXes and the individual work stations which the Clouds system will use to interface to the VAXes. Work on the semantic routines for Aeolus is proceeding in parallel with the development of routines to generate intermediate code for ACK. This work is being done in *Pastel*, an extended Pascal dialect developed at the Lawrence Livermore National Laboratory. Present plans call for the Aeolus compiler to be capable of interfacing with the action and object managers of the Clouds system by mid-1985.

As was mentioned above, we intend to use Aeolus as a framework within which to study programming methodologies for action-object systems. Among the hardest questions which need more study is how replication can most effectively be used to provide

availability. Actions and resilient objects ensure that failures are not catastrophic, but they are concerned with data integrity, not with how a program reacts to failures. The availability question involves use of multiple objects on different nodes to represent a single resource, thus providing continued access to the resource in the presence of individual node failures. Algorithms for read and write access to such resources must be developed and evaluated. The recent paper by Daniels and Spector [Dani83] is one example of such an algorithm.

We must also consider possible representations of work so that it may be restarted; this is an area that has been until recently unexplored [McKe84]. Most of the work on actions and objects has been oriented toward protection of data from failures. The fact that processes are considered to be an important, independent component supported by the Clouds architecture gives us a point of departure for this study. McKendry's work on Petri nets [McKe84] lays the groundwork for an attack on this problem within the framework of Clouds. If we view a program as a collection of processes interacting through shared objects, some features akin to the process interconnection specifications of Pronet [Macc82] may prove to be useful.

Our initial studies in programming methodologies for resilience and availability are described in [Wilk85a]; there, a plan is presented for determining such methodologies appropriate to the design of objects needed in the Clouds system. Examples of a replicated object exhibiting the properties of resilience and availability are given there, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the Aeolus/Clouds system to provide these properties. The issues with which we are concerned include the use of semantic knowledge of objects in the programming of replication; trade-offs between consistency and availability; the appropriateness of current programming models for replicated data; and the support needed from the operating system and language runtime system to ensure availability and forward progress of processes. As we progress with these studies, we will take advantage of our experience in the implementation of the Aeolus runtime system and its interaction with the action and object managers of the Clouds system.

[Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983

- [Macc82] Maccabe, A. B., and R. J. LeBlanc, "The Design of a Programming Language Based On Connectivity Networks," *Proceedings of the Third International Conference on Distributed Computing Systems*, Miami / Fort Lauderdale, October 1982
- [McKe84] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)
- [Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM* 26, 9, September 1983
- [Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985
- [Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, May 1985

QUARTERLY PROGRESS REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT #MDA 904-84-C-6035
REPORTING PERIOD: 1 APR 85 - 31 JUNE 85

1. Project Status

During the third quarter of this project, work has continued on each of the two tasks called for by the statement of work. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing. Under the Distributed File Systems task, work has proceeded on the integration of our storage management system with the Clouds kernel virtual memory management system. Additionally, implementation work has been done on several device drivers necessary to test the kernel and storage management system.

Under the Language Support for Robust Distributed Programs task, we have been refining the definition of our language, Aeolus, and proceeding with the implementation of the compiler. A copy of the Aeolus definition is attached as Appendix A.

The work on the tasks of this project is proceeding on schedule. Future plans include a continuation of the two investigations presented here. These efforts, in combination with other work in progress within the Clouds project, should lead to a working prototype system by the end of this year.

2. Storage Management -- Progress Report

This quarter's effort has been devoted to low-level implementation of the storage management system, namely the device drivers that Clouds will use. The Clouds kernel requires additional support beyond that normally supplied by conventional drivers. Primarily, the drivers must be able to ensure that a committing action's writes are completed before the action completes the commit procedure. A uniform interface is designed that provides access to any device on the Clouds system and successfully hides the difference between some very different devices. There are three efforts in this area. The first is the implementation of a device driver for the RA81 disk. The RA81 and UDA50 operate using the Mass Storage Control Protocol developed by DEC, which allows the development of classes of device drivers for devices supporting the protocol. We have been studying the protocol and are now working on the implementation of the driver.

The second effort is the implementation of a device driver for the RL02 removable disk device. The RL02 uses 10 Mb removable cartridges. This code is currently being tested.

The RL02 is a much simpler device than the RA81 and does not use the MSCP. The device supports flush of action requests on commit and does bad sector forwarding.

The third effort at this level is the development of a virtual disk for use by the Clouds kernel. Using a recently completed ethernet driver, a device driver is being implemented that uses the ethernet perform the disk i/o on another system under Unix, using the standard raw disk i/o routines. This will provide the Clouds kernel with use of the large RA81 disks very quickly. Also, if the disk used by the virtual disk device is dual ported, so that when the RA81 driver is available, it will have a device already using the Clouds format and the system can switch from the virtual device to an on-system device very easily.

At higher levels of the storage management system, the integration of the storage management system with virtual memory management has been examined. Data structures and routines for support of the page fault mechanism are under development. The routines will support the mapping of Clouds objects in virtual memory and the location of the disk page that satisfies a page fault.

The partition system is the next target for implementation, and preparation for that effort is underway. The major data structures for the partition object are designed and the major routines have been pseudo-coded. There have been some changes made to the overall design of the partition system, principally in the nature of the division of responsibility between the three levels of the storage management system: the device objects, the partition objects, and the segment objects. The storage management system attempts to make the action commit procedure transparent to the kernel, in that the segmentation system decides whether recovery is necessary for the segment being written. The modified design alters the interfaces between segment system and the kernel so that the segment system will have the information it needs to make these decisions. Analysis of the protocol for maintaining the partition free-page maps consistently has revealed some important simplifications that could be made to that protocol. Normally, the system uses a volatile version of the partition free-page map. Changes are made to a permanent version of the free-page map only when actions commit. This would seem potentially to require multi-page writes for each action commit. However, it is possible to put off making the allocations part of the permanent free-page map indefinitely, since it may be reconstructed from information contained in the segment system. The work of updating the free-page map could be assumed by the system initialization, reducing

some of the overhead of normal action commit. Also, a compromise solution is feasible, in which free-page map updating is done as part of the background work done by the kernel. These ideas will be explored as part of the development of the storage management system.

3. Aeolus - Progress Report

As part of the Clouds project, we are designing and implementing a high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology) in which those levels of the Clouds system above the kernel level will be implemented. *Aeolus* provides access to the synchronization and recovery features of Clouds. It also provides a framework within which to study programming methodologies suitable for action-object systems such as Clouds.

The definition of the *Aeolus* language [Wilk85b] is nearing completion. The design described in that report has undergone several iterations as the details of the language design have been filled in and as we have gained experience in the sort of programming for which the language is intended. We believe that these iterations of the design process have made the design of the language more coherent. Although we have drawn greatly on previous language designs in our work, we have tried to keep the design as simple as possible while fulfilling the design goals of *Aeolus*; thus, we have not attempted to provide such all-encompassing collections of features as are provided by some other language designs. As an example of the streamlined design of the language, the definition of type compatibility in *Aeolus* is quite simple; two entities are compatible if and only if they share the same type (by name equivalence). Thus, *Aeolus* provides no implicit type coercions. However, the language does provide powerful means of explicit type conversion, thus allowing the sort of manipulations necessary in systems programming while maintaining safety through strict typing. Among the benefits which should accrue from a simple language design are ease of learning and understanding of the language by programmers as well as ease of implementation of the compiler.

A compiler for *Aeolus* is currently under development on one of the DEC VAX 11/750 computers of the Clouds project under Berkeley Unix (TM) Version 4.2. We are using the *Amsterdam Compiler Kit* (ACK) [Tane83] to generate code generators for

Aeolus for both the Clouds VAXen and the individual work stations which the Clouds system will use to interface to the VAXen. Work on the semantic routines for Aeolus is proceeding in parallel with the development of routines to generate intermediate code for ACK. The code-generation work is progressing quite well; during the last quarter, we have been able to generate and execute code for object invocations which do not involve the facilities of the Clouds kernel or object managers (that is, code for what we call "non-Clouds objects").

Work is also progressing on the implementation of facilities for generating actual "Clouds objects." This entails the definition of the interface to the Clouds object and action managers, which will serve as an intermediary between user programs and the kernel facilities. Thus, the members of the compiler group are working with members of the kernel group on the definition and implementation of the action and object managers. We expect the Aeolus compiler to be capable of interfacing with the action and object managers of the Clouds system, and thus to be capable of invocations on actual Clouds objects, during the coming quarter.

We intend to use Aeolus as a framework within which to study programming methodologies for action-object systems. Our initial studies in programming methodologies for resilience and availability are described in [Wilk85a]; there, a plan is presented for determining such methodologies appropriate to the design of objects needed in the Clouds system. The issues with which we are concerned include the use of semantic knowledge of objects in the programming of replication; trade-offs between consistency and availability; the appropriateness of current programming models for replicated data; and the support needed from the operating system and language runtime system to ensure availability and forward progress of processes. We are currently attempting to identify data structures the study of which will be beneficial both in our work on methodologies and in the Clouds implementation effort; that is, we wish to study structures needed in the kernel and system code. This effort, and the work toward definition of the Aeolus/Clouds interface which was described above, is providing feedback in both directions, aiding both the interface design and our understanding of action-based programming methodologies. As we progress with the methodology studies, we are taking advantage of our experience in the implementation of the Aeolus runtime system and its interaction with the action and object managers of the Clouds system.

- [Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM* 26, 9, September 1983
- [Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985
- [Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, April 1985

Preliminary Aeolus Reference Manual

Technical Report

GIT-ICS-85/07

July 1985

C. Thomas Wilkes

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Table of Contents

1. Introduction	1
2. Explanation of Notation	2
3. Tokens	3
3.1. Identifiers	3
3.2. Numbers	3
3.2.1. Ints	3
3.2.2. Floats	3
3.3. Litchars and Litstrings	4
3.4. Comments and Compiler Options	4
3.5. Reserved Words	4
3.6. Operators and Delimiters	5
3.7. Other Characters	5
4. Declarations and Scopes	5
4.1. Compilation Units and Their Scopes	5
4.2. Qualified Identifiers	6
5. Constant Declarations	7
6. Type Declarations	8
6.1. Type Identifiers	8
6.2. Non-ID Types	9
6.2.1. Enumerations	9
6.2.2. Index and Pointer Types	9
6.2.3. Structured Types	9
6.2.3.1. Bitstrings	10
6.2.3.2. Strings	10

6.2.3.3. Arrays	11
6.2.3.4. Flexarrays	11
6.2.3.5. Records	12
6.2.3.6. Structures	12
6.2.3.7. Sets	12
6.2.3.8. Locks	13
6.3. Parameterized Object Types	14
7. Constraint Declarations	15
8. Variable Declarations	15
9. Expressions	16
9.1. Operands	16
9.2. Operators	18
9.2.1. Arithmetic Operators	18
9.2.2. Bitwise Operators	19
9.2.3. Address Operators	19
9.2.4. Logical Operators	20
9.2.5. Set Operators	20
9.2.6. Relational Operators	20
9.3. Type Compatibility	21
10. Statements	23
10.1. Simple Statements	23
10.1.1. Assignment Statements	23
10.1.2. Procedure Calls	24
10.1.3. Object Operation Calls	24
10.1.4. EXIT Statements	24

10.1.5. RETURN Statements	24
10.1.6. NULL Statements	25
10.2. Compound Statements	25
10.2.1. IF Statements	25
10.2.2. CASE Statements	25
10.2.3. LOOP Statements	27
10.2.4. USING Statements	28
10.2.5. REGION Statements	29
11. Procedures	29
11.1. Procedure Declarations	29
11.2. Procedure Invocations	31
12. Objects	31
12.1. Object Definition Parts	32
12.2. Object Implementation Parts	34
12.3. Object Operation Invocations	35
13. Actions	36
13.1. Action Events	36
13.2. Recoverable Variables	37
13.3. Action Invocations	38
14. Processes	39
15. REFERENCES	40
Appendix A: Systems Programming Example	A-1
Appendix B: Example of Action Programming	B-1
Appendix C: LALR(1) Grammar for Aeolus	C-1
Appendix D: Definition of the Object <i>standard</i>	D-1

Appendix E: Definition of the Clouds Action Manager E-1

1. Introduction

The goal of the *Clouds* project at Georgia Tech [Allc82, Allc83a, Allc83b] is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, which will provide an environment for the construction of reliable applications. The *Aeolus*¹ programming language developed from the need for an implementation language for those portions of the *Clouds* system above the kernel level. *Aeolus* has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the *Clouds* notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the *Clouds* system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as *Clouds* [LeBl85, Wilk86].

Thus, the main interest of *Aeolus* lies not in the language itself, but in what may be done with the language. We have avoided providing high-level features for programming actions with the intention of evolving designs for such features out of our experience with programming in *Aeolus*. These features will then be incorporated into an applications language for the *Clouds* system.

Aeolus has its roots in a long line of structured programming languages, including Simula, Pascal, Modula-2, and Ada.² Thus, many of its features should be easy to understand for those familiar with one of these languages; in particular, familiarity with Pascal or Modula-2 is assumed throughout this report, and features will often be explained in terms of the corresponding features in those languages.

The main structuring features of *Aeolus* (as of the *Clouds* system) are objects, actions, and processes. *Clouds* supports the *object* concept as a convenient structuring principle for facilitating recovery and synchronization; *Aeolus* also allows the programmer to use the object features of the language for the specification of abstract data types, without necessarily invoking the object and action management features of the *Clouds* system. Thus, *Aeolus* objects provide a separate compilation facility as well as access to the object support of *Clouds*; the separation of object specifications into *definition* and *implementation* parts (much as are *modules* in Modula-2 or *packages* in Ada) provides a safe interface to separately-compiled objects, as well as facilitating the design of large systems consisting of many objects (possibly implemented by several people) or the use of predefined objects. *Aeolus pseudo-objects* provide a means of isolating system dependencies—such as input/output or low-level machine architecture—into object-like modules which provide operations facilitating machine-level programming.

Support of the *Clouds* notion of *actions* in *Aeolus* is fairly low-level. Essentially, means are provided for specifying that an operation (procedure) of an object may be invoked as an action, or that an operation invocation is to be executed as a (oplevel or nested) action. Also, the status of action executions may be checked by means of calls to a *Clouds* action manager.

The *process* concept in *Aeolus* corresponds roughly to the *program* construct of Pascal or Modula-2. That is, a process ties together the constituent parts (objects) of a programmed system, and the invocation of a process provides activity in the *Clouds* system.

Except for the access *Aeolus* provides to the action management facilities of *Clouds* (which control recovery in the system), nothing in the language is explicitly dependent on the *Clouds* system for its implementation. In the *Clouds* implementation of *Aeolus*, the details of synchronization and recovery of objects are hidden by the interface to the *Clouds* object and

¹*Aeolus* was the king of the winds in Greek mythology.

²Ada is a registered trademark of the U.S. Government—Ada Joint Program Office

action managers; thus, for example, it is transparent to the programmer (and to the language runtime support) whether an operation invocation involves a local or remote object. Therefore, an implementation of Aeolus—without its features for recovery handling—should be possible under any operating system; only the object management need be subsumed by the language runtime support, which should be trivial for a non-distributed system.

This report is not intended to be a tutorial on the Aeolus language; rather, it strives to be a concise definition of the syntax and semantics of Aeolus, and thus should serve as a reference for programmers and implementors.

2. Explanation of Notation

The *syntax* (grammar) of a language consists of rules for arranging sequences of *terminal symbols* (also called tokens) in the vocabulary of the language (keywords, numbers, names (identifiers), and certain other characters used as punctuation to make the language more readable) into *sentences* (or sentential forms) which have meaning in the language. A syntax rule often specifies that a sequence of terminal symbols be grouped into a *nonterminal symbol*, an entity in the language which often has an intuitive meaning, such as an *expression* or a *statement*.

To describe the syntax of Aeolus in this manual, we will use a notation known as the extended Backus-Naur form (EBNF). (A complete grammar for Aeolus in LALR(1) form is presented in Appendix C.) In this notation, the so-called *metasymbols* [and] are used to enclose an Aeolus sentential form which is optional; the metasymbols { and } are used to enclose an Aeolus sentential form which may be repeated any number of times (possibly zero times). Tokens are enclosed in double quotes (“”); nonterminal symbols are enclosed in angle brackets (<>). The left-hand side of a syntax rule specifies the nonterminal which is being defined, while the right-hand side of the rule gives the sequence of terminal and nonterminal symbols which are valid for the nonterminal being defined; the two sides of the rule are separated by the metasymbol (meaning “expands into”).

Thus, for example, the syntax rule

$$\langle \text{identifier list} \rangle \rightarrow \langle \text{identifier} \rangle \{ \text{“,”} \langle \text{identifier} \rangle \}$$

specifies that the nonterminal *identifier list* consists of either a single *identifier* nonterminal, or a sequence of two or more identifiers separated by the comma token (“,”). The following are valid identifier lists:

```
foo
foo, bar
foo, bar, baz
```

Also, the rule

$$\langle \text{variable declaration} \rangle \rightarrow \langle \text{identifier list} \rangle \text{“:”} \langle \text{type} \rangle [\text{“:=”} \langle \text{expression} \rangle]$$

indicates that a *variable declaration* consists of an identifier list followed by the colon token (“:”), a specification of the *type* of the variable(s), and an optional *initialization* of the variable(s) consisting of an assignment operator token (“:=”) followed by an expression. The following are valid variable declarations:

```
foo : real
foo, bar: integer := baz + 1
```

3. Tokens

The tokens, or terminal symbols, of the Aeolus language include *identifiers*, *int* and *float* numbers, *litstrings*, and *keywords* (or reserved words) and other *delimiters* (such as arithmetic operators and other types of special characters). In this section, we will discuss rules for the formation of these tokens.

The following general rules apply: the ASCII character set is assumed; blanks must not occur within tokens (except litstrings); line breaks may not occur within any token (thus a single token may not extend over several lines); and blanks as well as line breaks are ignored except where they serve to separate consecutive tokens. Arrangement of tokens on lines may be in free format; in particular, there may be multiple statements on a line. The case of letters is ignored in keywords and identifiers; however, the case of letters in litstrings is preserved.

3.1. Identifiers

An Aeolus identifier must begin with an upper or lower case letter, which may be followed by any number of letters or digits. Also, a *separator* (the underscore character “_”) may be placed between any two characters within an identifier to improve readability; however, a separator may not occur at the beginning or end of an identifier.

<identifier> → <letter> { [<separator>] <letter or digit> }

Examples:

I am an_Aeolus_identifier As_am_I

3.2. Numbers

An Aeolus number is an “int” or “float” number, which may be specified in any base between 2 and 16 inclusive.

3.2.1. Ints

A decimal “int” starts with a digit (“0” through “9”), which may be followed by any number of digits, optionally separated by an underscore character (“_”) for readability. Ints in bases other than 10 may be specified by giving the base (a decimal number between 2 and 16 inclusive), followed by the character “#”, followed by the based number. A based number in a base greater than 10 may include the characters “A” through “F”, as appropriate to the base of the number. (Note that case is not significant for these characters.)

<num> → <digit> { [<separator>] <digit> }
 <basedit> → <digit>, “A” .. “F”
 <basednum> → <basedit> { [<separator>] <basedit> }
 <int> → <num>
 <int> → <num> “#” <basednum>

Examples:

1 32767 32_767 2#101010

8#52 16#2A 16#ff 13#42

3.2.2. Floats

A “float” number consists of a *whole part* followed by either a *fractional part* or an *exponent* or both.³ The whole part is a (possibly based) number. The fractional part consists of a fractional point “.” followed by a number with the same base as the whole part. The exponent consists of the letter “E” or “e” followed by a (possibly signed) decimal number, indicating the power of the base by which the float number should be multiplied. The base of a

³Thus, a float number must always begin with a digit.

float number is given as for an int; however, if a float number is based and has an exponent, the character “#” must appear before the exponent. If no base is given, base 10 (decimal) is assumed.

```

<exponent>  →  “E” [<sign>] <num>
<sign>      →  “+”, “-”
<float>     →  <num> “.” <num>
<float>     →  <num> [“.” <num>] <exponent>
<float>     →  <num> “#” <basednum> “.” <basednum>
<float>     →  <num> “#” <basednum> [“.” <basednum>] “#” <exponent>

```

Examples:

```
3.14159 8#7.77 0.1e32 2#1011#E-27 16#7f.a2#e+5
```

3.3. Litchars and Litstrings

A *character* is any member of the ASCII character set, including both printable characters (alphanumeric and punctuation) and control characters. Also, some systems may define extensions to the ASCII character set (for instance, graphics characters) which may be considered character tokens on those systems. A *litstring* (literal string) token is a sequence of characters enclosed in single quotes (“’”). To include a single quote as a character in a litstring, the single quote must be doubled (“’’”). A special case of the litstring token is the *litchar* (literal character) token, which is a litstring token consisting of a single character.

```

<litstring> →  “’” {<character>} “’”
<litchar>   →  “’” <character> “’”

```

Examples of LITSTRINGS:

```
'Hello, world' 'Don't be sad' 'This is a "litstring"'
```

Examples of LITCHARs:

```
'a' 'Z' '?' '!!!' 'm'
```

3.4. Comments and Compiler Options

A *comment* is explanatory text inserted into code for the reader's benefit; it is ignored by the compiler, and does not affect the meaning of the code. In Aeolus, a comment may be placed anywhere within a line where a blank may be placed. It begins with an exclamation point (“!”) and ends either at the next exclamation point or the end of the line on which the comment started, whichever comes first. Thus, comments do not extend over multiple lines.

Examples:

```
! This is an in-line comment. !      !As is this.!
```

```
! This comment goes to the end of this line.
```

A *compiler option* is used to communicate to the compiler the desired settings for various options which the compiler being used may implement, for example, whether range checks for valid variable values are to be generated. A compiler option begins with a dollar sign (“\$”) and ends either at the next dollar sign or at the end of the line on which the compiler option started, whichever comes first.

Examples:

```
$r+ $      $pagelength=84
```

3.5. Reserved Words

The following is a list of the reserved words (keywords) of Aeolus. These words *may not* be used as identifiers! Although the reserved words are shown here in upper case, upper and

lower case may be freely mixed in these words.

ACTION	FLEXARRAY	OVERRIDES
ARRAY	FOR	PROCEDURE
AUTOSYNCH	FORWARD	PROCESS
BEGIN	IF	PSEUDO
BITSTRING	IMPLEMENTATION	PURE
BY	IMPORT	RECORD
CASE	INDEX	RECOVERABLE
CONST	INLINE	REGION
CONSTRAINT	IS	RETURN
DEFINITION	LOCK	STRING
DO	LOOP	STRUCTURE
DOMAIN	MODIFY	STRUCTURED
DOWNTO	NONRECOVERABLE	THEN
ELSE	NOT	TO
ELSIF	NULL	TYPE
END	OBJECT	USES
EVENTS	OF	USING
EXAMINE	OPERATIONS	WHILE
EXIT	OTHERWISE	

3.6. Operators and Delimiters

The following are characters or groups of characters used as operators or delimiters (punctuation) in Aeolus.

()	->
{	}	@
	~	"
:=	+=	-
*=	/=	
^=	<<=	>>=
&=	%=	=
<	>	<>
<=	>=	
^	<<	>>
&	*	/
+	-	%

3.7. Other Characters

As mentioned before, blanks (except in litstrings) are ignored wherever they are not required to separate other tokens; thus, blanks may be used freely to improve the readability of code. Semicolons (“;”) are ignored in the same way as blanks; thus, semicolons may be used to separate or terminate statements if so desired, but are not required. Non-printable (control) characters are also ignored.

4. Declarations and Scopes

All identifiers in Aeolus code must be introduced by a *declaration*. In this section, the rules for ordering and extent of declarations will be presented.

4.1. Compilation Units and Their Scopes

Those sentential forms described by the Aeolus grammar which may be compiled are called *compilation units*. Compilation units include *object definition parts*, *object implementation parts*, and *processes*. As will be clarified in section 12, an object definition part serves to declare

those identifiers—constants, types, and operations—which the object makes available to other objects or processes, while the object implementation part actually provides the code for the object. Other objects or processes may *import* an object definition, and use the identifiers declared by it as if those identifiers had been declared locally.

Every compilation unit implicitly imports the *standard* object, which defines various useful identifiers. (These are listed in Appendix D.) Before any other declarations are given, the compilation unit may import other objects via an *import clause* (see section 12). Then, declarations of constants, types, variables (except in object definitions), and procedures (operations) may be given in any order, as long as the declaration of any identifier used in another declaration textually precedes this use. There are, however, two exceptions to this general rule.⁴ A procedure may be declared *forward*; that is, only its header is declared, while the declaration of its body is delayed until later (see section 11). Also, a pointer may be declared to reference a type whose declaration is delayed (section 6).

After an identifier has been declared, other declarations and statements may refer to it, as long as these references occur within the *scope* of the identifier. The scope of an identifier extends from the point of its declaration to the end of the *block* in which it was declared. That is, if the identifier was declared in the the declaration part of a compilation unit, its scope extends to the end of that compilation unit; if, however, the identifier was declared in the declaration part of a procedure, its scope extends to the end of the procedure. The scope of identifiers introduced in a *using* statement (section 10) extends to the end of that statement.

The scope defined by a procedure is said to be *nested* within the scope defined by the surrounding compilation unit. As implied by the rules above, identifiers in a nested scope are not *visible* (available for reference) in the surrounding scope. An identifier in an nested scope may have the same name as an identifier in an enclosing scope; the identifier in the enclosing scope is then not visible in the nested scope. Within a scope, however, an identifier must be unique; that is, an identifier may not be declared with the same name as another identifier already declared in the same scope (see below). Procedure declarations may not be nested (within other procedure declarations); thus, the maximum *nesting level* in Aeolus is 2, where the level of a compilation unit is 1.

4.2. Qualified Identifiers

As was stated above, an identifier must be unique within the scope in which it is declared so that the entity which it represents may be correctly identified. However, it often occurs that different object definitions declare constant or type identifiers with the same name, or that different enumerated types have members with the same name,⁵ or that different objects have operations with the same name, or that different records have fields with the same name. Thus, it is sometimes necessary to *qualify* an identifier with the name of its defining type or record to ensure that it is unique.

If types or constants with the same name defined by more than one imported object type⁶ are visible in a scope, or if similarly-named members of different enumerated types are visible in a scope, these names must be qualified with the names of their defining types:

<type-qualified id> → <type id> " " <identifier>

For example,

⁴These exceptions allow more general data structures and procedural definitions to be formulated, in particular recursive structures.

⁵This problem may also occur in Pascal, which does not provide for qualification of enumerated types; thus, so-called "holes" may be left in the types.

⁶As we shall see in the next section, the names of imported object definitions may be used as the names of types. Variables declared with an object type are said to be *object instances*.

```
obj1"foo    obj2"foo
```

refer to identifiers named "foo" defined by object types "obj1" and "obj2", respectively. Also, if the enumerated types "signal_colors" and "primary_colors" are defined as follows:

```
type signal_colors is ( red, yellow, green )
```

```
type primary_colors is ( red, green, blue )
```

then references to the identifiers "red" and "green" must be qualified:

```
signal_colors"red primary_colors"red
```

```
signal_colors"green primary_colors"green
```

Different object types may define operations with the same name; however, there may also be several *instances* of the same object type visible in a scope. Object *operation invocations* must be qualified by the name of the object instance on which we wish to operate:

```
<obj op invocation> → <obj instance id> "@" <op call>
```

For example, if variable "in_queue" is an instance of an object type (say, "queue") with operation "enqueue":

```
in_queue @ enqueue (item)
```

The situation of record fields is similar to that of object operations. Declarations of record types may define fields with the same name; also, there may be several variables declared with the same record type visible in a scope. Thus, field references must be qualified by the name of the field's parent record.⁷

```
<field ref> → <parent variable> "." <field id>
```

For example, if variables "a" and "b" are both of some record type "complex," we may have:

```
a.realpart b.realpart a.imaginarypart
```

5. Constant Declarations

An identifier declared as a *constant* is associated with a value which may not be changed. Thus, a constant may not be the target of an assignment statement (see section 10). The *type* of a constant may be any valid type specification (section 6). The *value* of a constant may be specified by an expression (section 9) in which only constant terms appear. Calls to (value-returning) procedures defined by the object *standard* are also allowed to appear in such an expression.

```
<const decl> → <const id decl> ":" "const" <type> ":@" <expr>
```

⁷This qualification is often called the *field dereference* operation.

Examples:

```
i : const integer := -10
```

```
j : const integer := i + abs (2*i)
```

6. Type Declarations

The declaration of a data type specifies the set of values which variables of that type may assume. In the case of structured types, the type declaration also gives a "blueprint" of the structure of variables of that type.

The general syntax for declaration of new types is:

```
<type decl> → "type" <new type id> "is" <type>
```

As we shall see in the remainder of this section, types fall into three general classes: type identifiers (the names of previously-declared types, including non-parameterized object types), non-ID types (including enumerations and structured types), and parameterized object types. The compatibilities of types are discussed in section 9.3.

Any type may have an optional indication that variables of that type, or components (of some variable) with that type may be *shared*. This attribute is indicated by the use of the keyword *shared* before the type indication. The use of shared variables is explained in section 10.2.5.

6.1. Type Identifiers

The simplest sort of type specification is simply the name of a previously-declared type, optionally followed by a *constraint specification*:

```

<type>      → ["shared"] <constrained type id>
<constrained type id> → <type id> [<constraint spec>]
<constraint spec>   → "[" "]"
<constraint spec>   → "[" <subrange> "]"
<subrange>         → <scalar const> ".." <scalar const>

```

Constraint specifications are described in section 7. The main utility of type identifiers is in specifying the types of entities such as variables (section 8).

Several useful predefined types are provided by the object *standard*, which is automatically imported by every compiland. The definition part of *standard* is shown in Appendix D. It defines the following basic scalar types:⁸

- type *integer*, whose variables assume values between MININT and MAXINT;
- type *longint*, whose variables assume values between MINLONGINT and MAXLONGINT;
- type *unsigned*, whose variables assume values between MINUNS and MAXUNS;
- type *longuns*, whose variables assume values between MINLONGUNS and MAXLONGUNS;
- type *boolean*, whose variables assume values FALSE or TRUE;
- type *char*, whose variables assume values of the character set used by the computer on which the program is being used (that is, those values representable by *litchar* tokens); and

⁸As shown in Appendix D, the types *integer*, *longint*, *unsigned*, and *longuns* may be considered to be new types derived from constraints on an underlying *int* number "type" (which includes all numbers representable by an "int" token), while type *real* may be considered to be derived from a constraint on an underlying *float* number "type" (which includes all numbers representable by a "float" token). The types derived from "int" tokens are denoted collectively as the "int types" in this document.

- type *real*, whose variables assume real numbers as values.
- Scalar types provide the basis for the construction of structured types.

6.2. Non-ID Types

The non-ID types include enumerated types, index and pointer types, and structured types.

$$\langle \text{type} \rangle \rightarrow [\text{"shared"}] \langle \text{non id type} \rangle$$

6.2.1. Enumerations

An *enumeration* (or *enumerated type*) consists of a list of identifiers which are used as constants in the program. Variables of that enumeration type may assume *only* those identifiers as values. The sequence of the identifiers in the declaration of the enumeration defines an ordering of those identifiers; the ordinal value of the first identifier is 0.

$$\begin{aligned} \langle \text{non id type} \rangle &\rightarrow \text{"("} \langle \text{enumer id list} \rangle \text{"}")} \\ \langle \text{enumer id list} \rangle &\rightarrow \langle \text{id decl} \rangle \{ \text{","} \langle \text{id decl} \rangle \} \end{aligned}$$

Example:

```
type days is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
```

6.2.2. Index and Pointer Types

An *index* type is a scalar type, variables of which will be used as indices in *for* loops or as array indices.⁹ A variable of an index type must be declared locally to the scope within which it is used. Structures may not have components of an index type, nor may variables of an index type be passed as *var* parameters to procedures or operations. The index variable of a *for* loop must be of an index type.

$$\langle \text{non id type} \rangle \rightarrow \text{"index"} \langle \text{constrained type id} \rangle$$

Example:

```
type loopindex is index integer [1 .. 10]
```

Variables of a *pointer* type may assume as values pointers to variables of another type *t* specified in the declaration of the pointer type:

$$\langle \text{non id type} \rangle \rightarrow \text{"->"} \langle \text{id use} \rangle$$

Pointer values are generated via calls to the operation *new* defined by object *standard*. Also, a variable of any pointer type may assume the value *NIL*, which means the variable is not pointing to anything.

Example:

```
type intptr is -> integer
```

6.2.3. Structured Types

Declarations of structured types provide blueprints for arranging groups of components of scalar types or of other structured types into a single structure. Structured types provide the programmer differing levels of abstraction with which to view data, from the most primitive view—sequences of bits—through high-level abstractions such as records.

$$\langle \text{non id type} \rangle \rightarrow \langle \text{struct type} \rangle$$

⁹This declaration provides a hint to the compiler that a variable of this type would be a good candidate to be placed in a register.

The structured types include *bitstrings*, *strings*, *arrays*, *flexarrays*, *records*, *structures*, *sets*, and *locks*.

The declaration of a structured type may be associated with a constraint which gives an indication of the number of elements in an entity (variable, record field, or parameter) of that type, so that the size of the entity may be determined at the time of compilation. As explained in section 7, a constraint thus associated with a type declaration is inherited as an attribute by entities of that type, although the constraint is not considered a part of the type itself. Such a constraint may take the form of an explicit length specification (as with strings and bitstrings), or (as with arrays) may be derived from constraints on the index types. It is sometimes useful, though, to delay the specification of size constraints of a structured type which is to be used as the type of a formal parameter. The formal parameter then takes on the size constraints of an actual parameter substituted for it during a procedure call. To support this, a size constraint may be specified by a *delayed constraint* form; delayed constraints may be used only as attributes of formal parameters.

6.2.3.1. Bitstrings

A *bitstring* provides the most primitive structured abstraction of data, that of simply a sequence of bits:

$$\langle \text{struct type} \rangle \rightarrow \text{"bitstring" "(" [\langle \text{expr} \rangle] ")"}$$

The length constraint of the bitstring (in bits) may be indicated by the value of the constant expression $\langle \text{expr} \rangle$. If $\langle \text{expr} \rangle$ is not given, the type declaration is considered to be associated with a delayed length constraint attribute; the length of a bitstring with such an attribute may be obtained at runtime by use of the operation *bitlength* provided by object *standard*.

Example:

```
type nybble is bitstring (4)
```

The "system" object, defined for each computer system on which Aeolus may be compiled,¹⁰ provides declarations of several useful bitstring types. These types are referred to collectively as the *storage classes*, since they define the units of storage supported by the hardware of most computer systems: types *bit*, *byte*, *word*, *longword*, and *quadword*, with lengths BITSIZE, BYTESIZE, WORDSIZE, LONGWORDSIZE, and QUADWORDSIZE, respectively.

Another important bitstring type, *address*, is also defined by the "system" object. The address type is defined as *bitstring (ADDRESSSIZE)*. The "system" object defines a constant of type address called NIL, which was mentioned in section 6.2.2. The relationship between address types and pointer types is discussed in section 9.3.

6.2.3.2. Strings

A *string* is a sequence of components of type *char* (that is, a sequence representable by a *litstring* token), terminated by a NUL character (ASCII 0).

$$\langle \text{struct type} \rangle \rightarrow \text{"string" "(" [\langle \text{expr} \rangle] ")"}$$

The maximum length constraint of the string (excluding the NUL terminator¹¹) may be specified by the value of the constant expression $\langle \text{expr} \rangle$; this value must be a positive integer. If $\langle \text{expr} \rangle$ is omitted, the type declaration is considered to be associated with a delayed length constraint attribute; the maximum length of a string with such an attribute may

¹⁰At present, Aeolus is supported on the DEC VAX and IBM PC-XT and -AT families of computers; the system objects for these families are named *VAX_System* and *PC_System*, respectively.

¹¹The *length* operation on objects of type string—described later in this report—returns the count of characters in the string, also excluding the NUL terminator.

be obtained at runtime by use of the operation *maxstringlen* defined by object *standard*. The components of the string are stored in consecutive bytes of memory.

Example:

```
type tokenstring is string (127)
```

A variable of type "tokenstring" will take up 128 bytes of memory, including room for a terminator character for a string value of the maximum length (127 characters).

6.2.3.3. Arrays

An *array* is a sequence of a fixed number of components which are all of the same type. The individual components or *elements* of an array are specified by the element's *indices*, which are values belonging to the *index type* of the array.

```
<struct type>   →  "array" "[" [<index type list>] "]" "of" <type>
<index type list> →  <constrained type id> {"," <constrained type id> }
```

Example:

```
type smallarray is array [ integer [1 .. 10] ] of integer
```

A declaration of the form

```
array [T1, T2, ... , Tn] of T0
```

with an *index type list* of *n* index types is considered shorthand for the declaration

```
array [T1] of array [T2] of ... of array [Tn] of T0
```

The size constraint attribute associated with entities of an array type is derived from constraints associated with the index types. Should any of the index types be associated with a delayed constraint, the array type is considered to be associated with a delayed size constraint. The minimum and maximum values of the index constraints for such an array may be obtained at runtime by use of the operations *low* and *high* provided by object *standard*.

6.2.3.4. Flexarrays

A *flexarray* is an array with a flexible size constraint attribute (sometimes called a *dynamic array*):

```
<struct type>   →  "flexarray" "[" [<index type list>] "]" "of" <type>
<index type list> →  <constrained type id> {"," <constrained type id> }
```

Each index type of a flexarray must be associated with a delayed constraint. A flexarray must be initialized with a special form of the operation *new* provided by object *standard*; the lower and upper bound constraints for each dimension of the flexarray (from first to last) are given as parameters to *new*. The lower and upper bound constraints for the first dimension of a flexarray may be changed by specification of the new bounds in a call to operation *resize* provided by object *standard*. A flexarray type may be used only for the type of a variable, and thus may not be used as the type of a parameter, record field, or array element.

Example:

```
type smallflexarray is flexarray [ integer [] ] of integer
f : smallflexarray
new (f, 1, 10)
resize (f, -10, 20)
```

6.2.3.5. Records

A *record* is a sequence of a fixed number of components which are of possibly differing types. An individual component or *field* of a record is specified by its *field name*, qualified by the name of the record variable to which the field belongs.

A record type declaration specifies the names and types of each field in a variable of that record type. A record type may also have *variant fields*. The first field of a variant field is called its *tagfield*. The remainder of the variant field consists of a *variant list*, each of whose *variants* is prefaced by a *variant label list*, a list of constants whose types match that of the tagfield. The value of the tagfield selects the variant from the variant list one of whose variant labels matches that value.

As we shall see in Section 8, tagfields may be assigned only in record constructors; thus, a tagfield may be changed only if all other fields of the record are specified at the same time.

6.2.3.6. Structures

A *structure* is a special case of a record somewhat similar to the *packed record* construct of Pascal. The declaration of a structure type specifies the storage class which the structure will fit:

```
<struct type>  →  "structured" <type id>
                  <field list>
                  "end" "structure"
```

(The storage classes were discussed in section 6.2.3.3.) A field in a structure typically represents a bitstring or scalar; the fields are packed together consecutively within an object of the specified storage class (without implicit padding), with the first field specified starting at the most significant bit position in the storage class. The compiler checks that the fields declared for the structure together fit into the specified storage class.

6.2.3.7. Sets

A *set* type defines a powerset of sets of values of the specified *base type*:

```
<struct type>  →  "set" "of" <constrained type id>
```

```
<struct type>  →  "record" <field list> "end" "record"
<field list>   →  <field> {"," <field>}
<field>        →  <field id list> ":" <type>
<field>        →  <variant field>
<field id list> →  <id decl> {"," <id decl>}
<variant field> →  "case" <tagfield id> ":" <scalar type id> "of"
                  <variant list> [<variant otherwise>]
                  "end" "case"
<variant list> →  <variant> {"|" <variant>}
<variant>      →  <variant label list> ":" <field list>
<variant label list> →  <variant label> {"," <variant label>}
<variant label> →  <scalar const>
<variant label> →  <subrange>
<variant otherwise> →  "otherwise" <field list>
```

Syntax of Record Type Declarations

```

type t is
  record
    case tf1 : days of
      Monday .. Friday :
        office_no,
        work_phone : integer
      || Saturday, Sunday :
        home_phone : integer
    end case
    last_name : string (20)
    case tf2 : integer of
      3, 5 .. 7 :
        weekly_rate : integer
      || 8 .. 10 :
        monthly_rate : integer
        benefits : boolean
    otherwise
      hourly_rate : integer
      temporary : boolean
    end case
  end record

```

Example of a Record Type Definition

The base type of a set must be scalar. There is no restriction on the number of elements that the base type may have.

Example (see section 6.2.1):

```
type dayset is set of days
```

6.2.3.8. Locks

A *lock* type may be used to declare variables which in turn may be used to implement locking protocols on particular *values* in some domain.¹²

```

<struct type>   →  "lock" "(" <compat list> ")" ["domain" "is" <type>]
<compat list>  →  <compat> {",", <compat>}
<compat>       →  <id use> ":" "[" <compat id list> "]"

```

A lock declaration includes the specification of a *compatibility list*, which defines, for a given *mode* of the lock, which other modes are compatible with that mode.¹³ The presence of an identifier in a compatibility list serves as a declaration of that identifier as a mode of the lock type; the modes of a lock type may together be thought of as an enumeration. An empty compatibility list indicates that the given mode is incompatible with all other modes.

¹²Note that a lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be obtained on a file name even if that file does not yet exist. The lock structure is directly supported by the Clouds architecture.

¹³A lock may be set with a specified mode only if other modes already set, if any, are compatible with that mode. Thus, a process adhering to a protocol using that lock may wish to block until the requested mode is available. Operations are provided by object *standard* for testing, setting, and releasing locks (see Appendix D).

```

type VAX_processor_status is
  structured longword
    CM,                                     ! bit
    TP                                     ! 31: Compatibility Mode
      : boolean                            ! 30: Trace Pending
    MBZ1                                   ! 29-28: must be zero
      : bitstring (2)
    FPD,                                   ! 27: First Part Done
    IS                                     ! 26: Interrupt Stack
      : boolean
    current_mode                           ! 25-24
      : 0 .. 3
    previous_mode                          ! 23-22
      : 0 .. 3
    MBZ2                                   ! 21: must be zero
      : boolean
    IPL                                    ! 20-16: Interrupt Priority Level
      : 0 .. 16#1f
    MBZ3                                   ! 15-8: reserved (must be zero)
      : byte
    DV,                                    ! 7: Decimal overflow bit
    FU,                                    ! 6: Floating Underflow bit
    IV,                                    ! 5: Integer overflow bit
    T,                                     ! 4: Trace bit
    N,                                     ! 3: Negative condition code
    Z,                                     ! 2: Zero condition code
    V,                                     ! 1: overflow condition code
    C                                     ! 0: Carry condition code
      : boolean
  end structure

```

Example of a Structure Type Definition

The lock declaration may also specify the *domain* of values which may be locked. If the domain specification is omitted, a simple lock (i.e., one which does not lock over any particular domain) is assumed.

Examples:

```

type simple_lock is lock ( busy : [] )

type file_lock is lock ( read : [read] ,
                        write : [] ) domain is string (20)

```

The declaration of "simple_lock" above defines a lock type with a single mode "busy" which is incompatible with itself; thus, only one client may set a lock variable of type "simple_lock" at any one time. The declaration of "file_lock," on the other hand, defines a lock type over the domain of strings of length 20. Clients may set a lock variable of type "file_lock" on a given string with modes "read" or "write." The "read" mode is specified as being compatible with other settings of "read" mode; the "write" mode is incompatible with itself and with "read" mode. Thus, a client may set the lock with "read" mode on a given string even if several other clients have outstanding settings of the lock with "read" mode on that string; however, a client wishing to set the lock with "write" mode on a given string must wait for all outstanding settings of "read" mode on that string to be released.

8.3. Parameterized Object Types

Object types may be defined with one or more *object parameters*, which allow the user to instantiate so-called generic objects. These parameters typically specify sizes or element types to be assumed by abstract data types. The formal parameters of the object definition header are

replaced by actual parameters when the object type is used in a declaration:

```

      <type>      →  ["shared"] <parameterized obj id>
<parameterized obj id> →  <id use> "(" <obj actual param list> ")"
<obj actual param list> →  <expr> {"," <expr>}

```

Example:

```
queue (integer, 128)
```

If an object type *queue* has been defined with formal parameters allowing instantiation of queues with a given element type and queue size, the above parameterized object id specification will instantiate a queue of integers with maximum queue size 128.

7. Constraint Declarations

A constraint, which indicates the minimum and maximum values which a variable having that constraint may assume, may be specified for any scalar type except *real*. As was described in section 6, a constraint may be associated with a type declaration; although the constraint is not considered to be part of that type, entities of that type (variables, parameters, or record fields) inherit the constraint as an attribute. The type being constrained may have already had a constraint associated with it; the new constraint replaces any previous constraint. The effect (or lack thereof) of constraints on type compatibility is described in section 9.3.

A constraint may also be associated with a previously-defined named type, and this association may be given a name which may be used as if it were a type identifier. Such an association is called a *constraint declaration*:

```
<constraint decl> → "constraint" <new constraint id> "is" <constrained type id>
```

Entities declared with a constraint identifier in place of a type are considered to be of the type indicated by the named type specified in the constraint declaration, as if the entity had been declared to be of that named type. Thus, a constraint declaration does not create a new type. However, the entity also inherits the constraint specified in the constraint declaration as an attribute. The new constraint replaces—for entities declared with the constraint name as a type—any constraint previously associated with the named type.

Example (see section 6.2.1):

```
constraint weekdays is days [Monday .. Friday]
```

8. Variable Declarations

A *variable declaration* introduces a variable into a process or object implementation part; it associates the variable with a unique identifier and with a fixed type. All variables whose identifiers appear in the same declaration list have the same type. A variable declaration may have an optional initialization clause, which consists of a constant expression of the same type as the variable type. This expression is evaluated, and its value assigned to the variable, before the block is entered in which the variable is declared.¹⁴

```

<var decl>      →  <id decl list> ":" <new type> [":=" <expr>]
<new type>     →  <type>
<new type>     →  "recoverable" <type> {"," <override>}
<override>    →  <id decl> "overrides" <id use>

```

A variable may also be declared to be located at a specified address:

```
<var address decl> → <id decl> "[" <expr> "]" ":" <new type> [":=" <expr>]
```

¹⁴Variables declared global to a compiland are static, and may be initialized before execution (that is, at compilation or link time).

The address expression must be a constant expression of type *address*.

In the version of Aeolus under the Clouds system, variables may be declared to be *recoverable*. Recoverable variables are discussed in section 13.

Examples:

```

i, j : integer [ 1 .. 10 ] := 0
a : array [ integer [ 1 .. 10 ] ] of
    record
        realpart, imaginarypart : real
    end record

string_array : array [ integer [ 1 .. 10 ], integer [ 100 .. 200 ] ]
of -> string ( 80 )

KB_flag [ 16#0017 ] : PC_keyboard_flag
    
```

9. Expressions

The use of *expressions* allows the programmer to obtain the values of variables and to generate new values by specifying computations to be performed. An expression is constructed from *operands* and *operators*.

9.1. Operands

An operand is either a literal constant (a number, string, or *constructor* [see below]), or a variable. A variable may be designated either by a (possibly qualified) simple identifier, or, if the variable is of a structured type, by a *structured variable*, which consists of the variable name followed by *selectors*. Selectors serve to designate the desired component of a variable. A call to a value-returning object operation or procedure (function) may also be used anywhere a variable may be used; in particular, the value returned by such a call may be dereferenced with selectors, if this return value has the appropriate type.

```

<variable>    → <id use>
<variable>    → <func call>
<variable>    → <obj op invocation>
<variable>    → <structured var>
<structured var> → <variable> "." <id use>
<structured var> → <variable> "-"
<structured var> → <variable> "[" <expr> {"," <expr>} "]"
    
```

If the variable is of a pointer type, the *pointer dereference* operator (">") may be used to obtain the item referenced by the pointer. If the variable is of a record type, an individual field of the record may be obtained by use of the *field dereference* operator ("."), followed by the name of the field. An individual element of a variable of an array type may be obtained through use of an *element selector* operator, which specifies the index of the array element desired. Thus, the structured variable *a*[<expr>] designates that element of array *a* whose index is the value of the expression <expr>. The list of index expressions in an array element selector, such as

```
a [ <expr 1> , <expr 2> , ... , <expr n> ]
```

is considered shorthand for the sequence of selectors

```
a [ <expr 1> ] [ <expr 2> ] ... [ <expr n> ]
```

for an array *a* declared with *n* dimensions. The type of each element selector expression must be compatible with the type of the corresponding index type of the array (see below).

An element selector may also be applied to a variable of type bitstring or string; for purposes of element designation, these sequences may be considered to be one-dimensional arrays with element type *bit* and *char*, respectively. The index type of such a sequence is considered to be *unsigned* [1 .. LEN], where LEN is the length of the bitstring or string.

Examples of variable designations (see section 8):

```
a[5].realpart
a[i].imaginarypart
string_array [i, 110] ->
string_array [10, 150] -> [80]
```

As stated above, operands may be literal constants as well as variables. The specification of a literal constant of an integer or real number is simply a token of that type (see section 3). A constant of a structured type, however, must be built by specification of its elements in a *constructor*. Constructors for constants of structured types are built using the following syntax:

```
<constructor>  →  <type id> “ ” “ [” <con elem> { “ , ” <con elem> } “ ] ”
<con elem>    →  <expr> [ “ : ” <expr> ]
<con elem>    →  <subrange>
```

The constructor is prefaced by the name of the type to which the constant being constructed belongs. The value of each element of the constant is then specified (in the order in which the elements were declared in the relevant type declaration) by an expression which must have the same type as the corresponding element in the structured type. If a structure has several elements of the same type in sequence, the same value may be assigned to each element by specifying an optional *repetition factor* (a [positive] constant integer expression); thus, the constructor element 0:10 would specify that the value 0 be assigned to the next 10 elements in a structure.

The constructor for a constant of a set type merely lists those elements of the base type which are to be included in the set constant. An empty constructor (“[]”) for a constant of a set type implies the so-called *null set*, which is a set with no members.

Constants of bitstring and string types may also be expressed using more traditional styles of constructors for these types. The alternative constructor for a constant of a bitstring type is simply an unsigned binary number (or a number in another base with the equivalent bit pattern) with same number of bits in its representation as the length of the bitstring. We have already seen (in section 3) the alternative constructor for constants of a string type, that is, a string token with enclosing quotes. The string constructor may have no more characters than the maximum length of the string type. When the standard constructor syntax shown above is used for constants of bitstring or string type, each element need not be individually specified; rather, (bit)string constants of smaller (maximum) length may appear as constructor elements, as long as the total (maximum) length of all constructor elements matches the (maximum) length of the target (bit)string type. The individual (bit)string constants are concatenated into the resulting constant.

Constants of array, record, or structure types may be built only by using the above constructor syntax. Constructors are especially important for record or structure types with variant fields: the tagfield of a variant field may be assigned a value only in a constructor. Thus, a tagfield may not be changed without the specification of values for all other fields in the variant.¹⁵

¹⁵This restriction simplifies runtime checking of variants considerably.

Examples of constructors (see section 6 and below):

```
smallarray"[1, 2, 3:5, 4:2, 5]
word"[byte (2#1000), byte (2#0010)]
tokenstring"[Hello, world! ', 'Bye, now. ']
dayset"[Monday, Wednesday, Friday]
dayset"[]
```

9.2. Operators

The syntax of Aeolus expressions defines *precedence levels* of operators similar to those in Pascal or Modula-2. There are four levels of precedence: the logical NOT operator and the bitwise complement (“~”) operator have the highest precedence (level 1), followed by the *multiplicative operators* (level 2), then the *additive operators* (level 3), and finally the *relational operators* (level 4). When a sequence of operators has the same precedence, the sequence is executed from left to right in textual order. The order of evaluation in an expression may be changed by enclosing parts of the expression in parentheses.

The operators provided by the Aeolus language are listed below. Unless otherwise specified, these are binary operators. In certain cases, the same operator symbol has different meanings when applied to data objects of different types. The intended operation is then identified by the types of the operands.

9.2.1. Arithmetic Operators

These operators apply to compatible operands of type *integer*, *longint*, *unsigned*, *longuns*,

<factor>	→	“real”
<factor>	→	“integer”
<factor>	→	“char”
<factor>	→	“string”
<factor>	→	<constructor>
<factor>	→	<variable>
<factor>	→	“not” <factor>
<factor>	→	“~” <factor>
<factor>	→	“(” <expr> “)”
<term>	→	<factor> {“multop” <factor>}
<expr>	→	<simple expr>
<simple expr>	→	[“sign”] <term> {“addop” <term>}
<expr>	→	<rel expr>
<rel expr>	→	<simple expr> “relop” <simple expr>

Syntax of Expressions

and (except for the modulus operator) *real*:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
+	addition	3
-	subtraction	3
*	multiplication	2
/	division	2
%	modulus	2

The operators “+” and “-” may also be used as unary operators. They then denote the *sign* of a term; the “-” operator implies negation, while the “+” operator implies the identity operation. The “%” or modulus operator yields the remainder of an integer division of its (integer) operands:

$$x \% y \text{ gives the remainder of } x / y, \text{ for } y > 0.$$

The division operator (“/”), when applied to integer operands, yields the truncated quotient of its operands.

9.2.2. Bitwise Operators

The following operators may be applied to compatible operands of a bitstring type, except that the right operand of the shift operators is an expression of type *integer*:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
	bitwise OR	3
^	bitwise XOR	3
<<	left shift	3
>>	right shift	3
&	bitwise AND	2
~	bitwise complement (unary)	1

The left and right shift operators yield the value of their first operand shifted left or right (respectively) by the number of positions given by the value of their second operand; the vacated bits are zero-filled. The results of these operators are undefined if the value of the right operand is greater than the length (in bits) of the left operand. The bitwise complement operator (“~”) yields the one’s complement of its operand.

9.2.3. Address Operators

Arithmetic on pointers is not allowed in Aeolus. However, the bitstring type *address* allows the programmer to perform address computations via explicit conversions from pointer types (see section 9.3). The “system” object for the computer for which a compiland is being compiled (such as *VAX_System* or *PC_System*) defines three named operations on data of type *address*:

addr(*v*) Returns a value of type *address* representing the storage address of variable *v*, which may be a static or dynamic data item.

next(*a*, *t* [, <expr>]) Increments the address-type variable *a* by an amount equal to the product of the value of <expr> and the size in address units (bytes or words, depending on the system object being used) of the type represented by type identifier *t*. The type of <expr> must be one of the “int types.” If <expr> is omitted, the value 1 (one) is assumed for it.

prev(*a*, *t* [, <expr>]) The same as *next*, but the address-type variable *a* is decremented rather than incremented.

9.2.4. Logical Operators

The following operators apply to operands of type *boolean* and yield a *boolean* result:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
OR	logical conjunction	3
AND	logical disjunction	2
NOT	logical negation (unary)	1

9.2.5. Set Operators

The following operators apply to compatible operands of a set type and yield a value of the same type:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
+	set union	3
-	set difference	3
*	set intersection	2
/	symmetric set difference	2

The following named operations are also provided for sets by object *standard*:

- in*(elem, s) Returns TRUE if the scalar *elem* is currently a member of set *s*, FALSE otherwise. The type of *elem* must be the same as the base type of *s*.
- out*(elem, s) Returns the value of *not in*(*elem*, *s*).
- incl*(s, elem) The scalar *elem* is included in (becomes a member of) the set *s*. The type of *elem* must be the same as the base type of *s*.
- excl*(s, elem) The scalar *elem* is excluded from (is no longer a member of) the set *s*. The type of *elem* must be the same as the base type of *s*.

The following statements define the (binary) set operations:

<i>in</i> (<i>x</i> , <i>s1</i> + <i>s2</i>)	iff	<i>in</i> (<i>x</i> , <i>s1</i>) or <i>in</i> (<i>x</i> , <i>s2</i>)
<i>in</i> (<i>x</i> , <i>s1</i> - <i>s2</i>)	iff	<i>in</i> (<i>x</i> , <i>s1</i>) and <i>out</i> (<i>x</i> , <i>s2</i>)
<i>in</i> (<i>x</i> , <i>s1</i> * <i>s2</i>)	iff	<i>in</i> (<i>x</i> , <i>s1</i>) and <i>in</i> (<i>x</i> , <i>s2</i>)
<i>in</i> (<i>x</i> , <i>s1</i> / <i>s2</i>)	iff	<i>in</i> (<i>x</i> , <i>s1</i>) <> <i>in</i> (<i>x</i> , <i>s2</i>)

9.2.6. Relational Operators

The relational operators apply to compatible operands of scalar, set, and bitstring types, and yield results of type *boolean*:

<i>symbol</i>	<i>relation</i>
=	equality
<>	inequality
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The relations "<=" and ">=" denote improper inclusion when applied to sets, while the relations "<" and ">" denote proper inclusion. The relations "=" and "<>" may also be applied to operands of a pointer type. Operands of a bitstring type are considered to be unsigned values of the equivalent length for purposes of comparison. The relations "=" and "<>" are also defined for compatible operands of a pointer, string, record, or array type. All relational operators have the lowest precedence (level 4).

9.3. Type Compatibility

The operands of a binary operation are said to be *compatible* if they are of the same type, that is, if the types of the operand are *equivalent*. The types of operands are equivalent if the operands were declared with the same named type or with the same anonymous type.¹⁶ (This is known as *name equivalence* of types.) Thus, for example, given the declarations

```
type t is array [ integer [ 1 .. 10 ] ] of integer
a : t
b : t
c, d : array [ integer [ 1 .. 10 ] ] of integer
```

the variables *a* and *b* have equivalent types (and are hence compatible) since their types both derive from the named type *t*. Also, variables *c* and *d* are compatible, since they share the same anonymous type. However, variable *a* is not compatible with variable *c* since their types, although structurally equivalent, are not name equivalent, since the anonymous type of *c* is not name equivalent to the named type *t*.

The Aeolus language does not allow incompatible operands to appear in an expression; that is, there are no implicit type conversions (coercions). However, it is sometimes desirable to perform operations on operands of differing types. Thus, Aeolus provides the programmer with powerful means of explicit type conversion.

Every named scalar type definition also implicitly defines a conversion function of the same name as the scalar type. This function may accept as a parameter an operand of any other scalar type, or of a bitstring type of the same storage class. The result of the function invocation is considered to be a scalar of the named type. Thus, if we have

```
i : integer := 0
type fruits is (apples, oranges, lemons)
```

then the result of the expression *fruits(i)* is of type *fruits* and has value "apples;" also, the result of the expression *integer(oranges)* is of type *integer* and has value 1.

As mentioned in section 7, a constraint which is associated with a scalar type (by way of a constraint specification in the type's declaration, or via a constraint declaration) is not considered part of that type, but rather is an attribute which is given to an entity (variable, parameter, or record field) of that type¹⁷. Thus, a constrained entity is compatible with an entity which has the same type but a different (or no) constraint. For example, considering the declaration of type "days" in section 6.2.1 and the declaration of constraint "weekdays" in section 7, if we have the following variable declarations:

```
d1 : days    d2 : weekdays
```

then *d1* and *d2* are compatible. However, every type declaration creates a new type; thus, if we had the declarations

```
type i1 is integer    type i2 is integer
```

then entities of type "i1" are incompatible with entities of type "i2," as well as with entities of type *integer*. Also, had "weekdays" been declared as a type rather than as a constraint, e.g.,

```
type weekdays is days [Monday .. Friday]
```

¹⁶The term "anonymous type" refers to a specification of a non-ID type which appears as the type of an entity (variable, parameter, or field).

¹⁷Constraints are used for range checking (if enabled) and for determining the sizes of structures, not for type checking.

then variables *d1* and *d2* would not be compatible.

The type of a so-called *int token* (see section 3.2.1) is determined by the size of the number it represents. Such a number may be assigned to any of the "int types" *integer*, *longint*, *unsigned*, or *longuns* (see section 6.2). Since these types are constrained, however, when range checking is enabled, the number may not be greater than the maximum (or less than the minimum) value representable in the target type.¹⁸

The conversion functions *integer* and *longint* may also be applied to real expressions; if the resulting value is not too large for the given type, the result is an integer (or *longint*) value which represents the integral part of the real number (obtained by truncation). The real representation of an integer expression may be obtained by using the conversion function *real*.

The declaration of a named bitstring type implicitly defines a conversion function to that bitstring type from any scalar type representable in that length bitstring. Thus, access may be obtained to the bit representation of data in an explicit manner. This implicit conversion function will also accept a data item of another bitstring type as parameter, as long as the parameter's length (in bits) is no greater than that of the target type. If the length of the source type is less than that of the target type, the resulting value is padded on the right with zeroes to the length of the target type. Also, two named operations are defined by object *standard* for selecting parts of the *word* bitstring type. The *hbyte* and *lowbyte* functions return (as values of type *byte*) the high-order and low-order (respectively) bytes of their word parameter.

The definition of a named pointer type provides a conversion function of the same name from a variable of type *address* to that pointer type. However, the result of such a conversion may be used only for dereferences; it may not be assigned to a pointer variable. Values are assigned to address variables via the *addr* operation discussed above; values may be assigned to pointer variables only with the operation *new* defined by object *standard* (see Appendix D), or via assignment from other variables of the same pointer type. The exception to the above rules is a special constant of type *address*, called *NIL*, defined by each "system" object. The constant *NIL*, which denotes a null pointer or address reference, may be assigned to a variable of any pointer or address type.

A definition of a named string type provides a conversion function of the same name to that type from other string types with maximum lengths no greater than that of the target type. The resulting value is a null-terminated string with the same value as the source value, but with the same maximum length as the target type. A number of named operations are provided in object *standard* for other string manipulations and conversions, such as substring extraction and conversions between strings and numbers; these are listed in Appendix D.

The conversion functions discussed above are for the most part somewhat restrictive in the types of the arguments which they will accept. Also, if the range checking option is enabled during compilation, most of these functions will generate runtime range checks of their parameters. However, Aeolus provides a less restrictive (and less safe) means of type conversion. The *retype* function accepts as parameters a value of any type and a type identifier; the result is a value of the type specified by the type identifier, left-justified bitwise. For example:

```
longword_value := retype (integer_value, longword)
```

No type checking is performed; the only restriction is that the target type representation be no smaller (in bitlength) than the type of the source value. Any range checking or filling of unused space (when the target type is larger than the source type) is the responsibility of the programmer.

¹⁸This implies that negative numbers may not be assigned to *unsigned* or *longuns* variables, since the minimum value representable in those types is 0.

10. Statements

A *statement* allows the programmer to specify activities such as assignment of a value to a variable, decision branching, or repetitive execution of groups of statements. The so-called *simple statements* do not contain other statements, while the *compound statements* may contain other statements as parts. One or more statements may be grouped into a *statement list*:

$$\langle \text{stmt list} \rangle \rightarrow \langle \text{stmt} \rangle \{ \langle \text{stmt} \rangle \}$$

for use as a part of a compound statement.

10.1. Simple Statements

The simple statements include the assignment statement, procedure call, object operation call, EXIT statement, RETURN statement, and NULL statement.

$$\langle \text{stmt} \rangle \rightarrow \langle \text{simple stmt} \rangle$$

10.1.1. Assignment Statements

An assignment statement denotes the replacement of the value of the variable designated by the left-hand side with some function of the value of the expression on the right-hand side. The *assignment operator* used in an assignment statement describes what function of the value of the right-hand side is to be used. The simplest assignment operator is “:=” (pronounced “gets”), which denotes simple replacement. Other assignment operators apply some binary operator to the value of the variable designated by the left-hand side and the value of the expression on the right-hand side; the resulting value replaces the value of the designated variable. An example of such an assignment operator is “+ =” (pronounced “plus-gets”); the assignment statement “i + = 1” is equivalent to “i := i + 1”. The other binary operators (listed throughout section 9) also have corresponding assignment operators.

$\langle \text{simple stmt} \rangle$	\rightarrow	$\langle \text{assign stmt} \rangle$
$\langle \text{assign stmt} \rangle$	\rightarrow	$\langle \text{lhs} \rangle$ “assignop” $\langle \text{rhs} \rangle$
$\langle \text{lhs} \rangle$	\rightarrow	$\langle \text{lhs elem} \rangle$ { “,” $\langle \text{lhs elem} \rangle$ }
$\langle \text{lhs elem} \rangle$	\rightarrow	$\langle \text{id use} \rangle$
$\langle \text{lhs elem} \rangle$	\rightarrow	$\langle \text{structured var} \rangle$
$\langle \text{rhs} \rangle$	\rightarrow	$\langle \text{expr} \rangle$
$\langle \text{rhs} \rangle$	\rightarrow	[“toplevel”] “action” “(” $\langle \text{action invocation} \rangle$ “)”
$\langle \text{action invocation} \rangle$	\rightarrow	$\langle \text{proc call} \rangle$
$\langle \text{action invocation} \rangle$	\rightarrow	$\langle \text{obj op call} \rangle$
$\langle \text{action invocation} \rangle$	\rightarrow	$\langle \text{assign stmt} \rangle$

Besides a single variable designation, an assignment statement may also take a list of variable designations as its left-hand side; this is called a *multiple assignment*. Here, the value of the expression on the right-hand side is assigned to each of the variables designated on the left-hand side, from the right of the list to the left. For example:

$$i, j, k := m + 1$$

is equivalent to the series of assignment statements:

$$k := m + 1 \quad j := m + 1 \quad i := m + 1$$

Assignment statements with other assignment operators may also be multiple assignments. The variable designation rightmost in the variable list is used as the left operand for the binary operator. Thus:

$$i, j, k + = 1$$

is equivalent to the series of assignment statements:¹⁹

$$k := k + 1 \quad j := k + 1 \quad i := k + 1$$

An assignment statement may also take the form of an *action invocation*. Action invocations are described in section 13.

10.1.2. Procedure Calls

A procedure call statement activates a named procedure. The procedure call may have a list of actual parameters, which are substituted for the corresponding formal parameters defined by the procedure declaration:

$$\begin{aligned} \langle \text{simple stmt} \rangle &\rightarrow \langle \text{proc call} \rangle \\ \langle \text{proc call} \rangle &\rightarrow \langle \text{proc id} \rangle \text{ "(" } \langle \text{parm list} \rangle \text{ ")" } \end{aligned}$$

Procedure calls are more fully described in section 11.2.

10.1.3. Object Operation Calls

Object operation calls are similar to procedure calls. However, an object operation must in general be invoked on that instance of the object type given by the *object ID* specified in the operation call:

$$\begin{aligned} \langle \text{simple stmt} \rangle &\rightarrow \langle \text{obj op call} \rangle \\ \langle \text{obj op call} \rangle &\rightarrow \langle \text{obj id} \rangle \text{ "@" } \langle \text{obj op id} \rangle \text{ "(" } \langle \text{parm list} \rangle \text{ ")" } \\ \langle \text{obj id} \rangle &\rightarrow \langle \text{id use} \rangle \\ \langle \text{obj id} \rangle &\rightarrow \langle \text{structured var} \rangle \end{aligned}$$

Operation calls are more fully described in section 12.3.

10.1.4. EXIT Statements

An EXIT statement specifies the termination of one or more enclosing loops (see section 10.2.4). The keyword *exit* may be followed by either a period ("."), which specifies the termination of the immediately enclosing loop, or by an identifier, which specifies the termination of the enclosing loop with the same name:

$$\begin{aligned} \langle \text{simple stmt} \rangle &\rightarrow \langle \text{exit stmt} \rangle \\ \langle \text{exit stmt} \rangle &\rightarrow \text{"exit"} \langle \text{name option} \rangle \\ \langle \text{name option} \rangle &\rightarrow \text{"."} \\ \langle \text{name option} \rangle &\rightarrow \langle \text{id use} \rangle \end{aligned}$$

An EXIT statement may not appear outside a loop; however, a loop may contain several exit statements.

Examples:

```
    exit .
    exit outer_loop
```

10.1.5. RETURN Statements

A RETURN statement specifies the termination (and return from) the enclosing procedure. The keyword *return* may be followed either by a period (".") if the enclosing procedure does not return a value, or by an expression of the same type as the declared return

¹⁹This may be compared to the equivalent C statement:

```
i = j = k + = 1;
```

type if the procedure is value-returning:

```

<simple stmt>   →   <return stmt>
<return stmt>  →   "return" <value option>
<value option> →   "."
<value option> →   <expr>

```

A RETURN statement may not appear outside a procedure body; however, a procedure body may contain several RETURN statements.

Examples:

```

        return .
return 1.0 - cos (2.0*PI)

```

10.1.6. NULL Statements

A NULL statement indicates that no action is to be taken:

```

<simple stmt>   →   <null stmt>
<null stmt>    →   "null"

```

The NULL statement is useful in constructs in which a statement or statement list would ordinarily appear, but where no action is desired, for instance, in certain cases in a CASE statement or as the body of a procedure stub which is to be filled in later.

10.2. Compound Statements

The compound statements include the IF statement, CASE statement, LOOP statement, USING statement, and REGION statement.

```

<stmt> → <compound stmt>

```

10.2.1. IF Statements

The IF statement allows the programmer to construct decision control structures:

```

<compound stmt>  →   <if stmt>
<if stmt>        →   "if" <expr> "then"
                   <stmt list> {<elsif option>} [ <else option> ]
                   "end" "if"
<elsif option>   →   "elsif" <expr> "then" <stmt list>
<else option>    →   "else" <stmt list>

```

The expressions following the keywords *if* and *elsif* must be of type boolean. These expressions are evaluated in order, and the corresponding statement lists skipped, until one of the boolean expressions yield the value TRUE; the statement list following the keyword *then* after this expression is then executed, and control is then transferred to the statement following the keywords *end if*. If the optional ELSE clause is present, the statement list following the keyword *else* is executed if none of the boolean expressions evaluate to TRUE.

10.2.2. CASE Statements

The CASE statement allows the programmer to specify a multiple-branch decision struc-

the optional OTHERWISE clause is present, the statement list following the keyword *otherwise* is executed; if no OTHERWISE clause is present, control is transferred to the end of the CASE statement.

10.2.3. LOOP Statements

The LOOP statement allows the programmer to specify that a statement list be executed repeatedly, either for a specified number of iterations, or while some condition is true, or until the loop is explicitly exited:

<compound stmt>	→	<loop stmt>
<loop stmt>	→	[<iteration clause option>] <basic loop>
<loop stmt>	→	<loop id dec> ":" [<iteration clause option>] <basic loop> <loop id use>
<basic loop>	→	"loop" <stmt list> "end" "loop"
<iteration clause option>	→	"while" <expr>
<iteration clause option>	→	"for" <index id> "==" <expr> <direction> <expr> [<by clause>]
<direction>	→	"to"
<direction>	→	"downto"
<by clause>	→	"by" <expr>

The basic form of the LOOP statement, without the optional *iteration clause*, is essentially an infinite loop: the enclosed statement list is executed until the loop is explicitly exited by means of an EXIT statement (see section 10.1.4).

Two iteration clause options are available for control of the repetitive execution of the LOOP construct. The simplest of these two options is the WHILE clause, which specifies that the loop is to be continued as long as some condition is fulfilled. The expression following the keyword *while* must be of type boolean. This boolean expression is evaluated before each execution of the statement list enclosed by the LOOP construct; this repetition continues as long as the expression yields the value TRUE.

The second iteration clause option is the FOR clause, which specifies that a progression of values is to be assigned to a variable during the repetitive execution of the loop. The identifier following the keyword *for* is called the *loop index variable*; this identifier must have been declared as a variable of an index type (see section 6.2.2). The loop index variable may not be the target of an assignment statement within the statement list enclosed by the LOOP construct. The direction of the progression of values is specified by the use of one of the <direction> keywords *to* or *downto*; the former specifies an increasing progression (that is, the loop index is incremented on each iteration), while the latter specifies a decreasing progression (the loop index is decremented). The ordinal amount by which the loop index is incremented or decremented on each iteration is specified by the value of the expression following the keyword *by* in the optional BY clause; this expression must yield a positive value. If no BY clause is given, the value 1 is assumed for the increment or decrement. The starting value of the progression is given by the value of the expression following the token "==" , while the ending value of the progression is given by the value of the expression following the <direction> keyword; the types of these two expressions must be compatible with the base type of the loop index. All three expressions (starting value, ending value, and increment) are evaluated before the loop is entered. Execution of the statement list enclosed by the LOOP construct continues until the value of the loop index variable exceeds the ending value, in the sense of the direction of the progression.

A LOOP statement may optionally be qualified by a *loop identifier*. The appearance of this identifier at the start of the construct is considered to be the declaration of the loop identifier; if a loop identifier is specified, the same identifier must appear after the *end loop* keywords. The scope of the loop identifier is the extent of the LOOP statement which declared it. A loop identifier may be used in the <name option> clause of an EXIT statement (see section 10.1.4)

to specify the termination of an enclosing loop with that name.²⁰

Examples:

```

      InOut @ ReadChar (ch)
      while ch <> ' ' loop
        process_char (ch)
        InOut @ ReadChar (ch)
      end loop

for ch := 'z' downto 'a' by 2 loop
  process_char (ch)
  for i := integer (ch) to 10 * integer (ch) loop
    InOut @ WriteChar (ch)
  end loop
end loop

outer :
  loop
    loop
      InOut @ ReadChar (ch)
      if ch = '!' then
        number_of_sentences += 1
        exit outer
      elsif ch = ' ' then
        exit .
      end if
      process_char (ch)
    end loop
    number_of_words += 1
    skip_spaces ()
  end loop outer

```

10.2.4. USING Statements

The USING statement allows the programmer to “alias” parts of complicated variable designators. These “aliases” may then be used in place of those parts of the designators within the statement list enclosed by the USING construct:

```

<compound stmt> → <using stmt>
<using stmt> → “using” <use spec list> “do”
               <stmt list>
               “end” “using”
<use spec list> → <use spec> {“,” <use spec>}
<use spec> → <id decl> “for” <variable>

```

The effect of a USING statement is the creation of a nested scope for the extent of that USING statement; the identifiers on the left-hand sides of each <use spec> in the <use spec list> are considered to be declared within this scope. The effective address value yielded by the variable designation on the right-hand side of a <use spec> is assigned to the identifier on the left-hand side of that <use spec>²¹ (that is, the identifier denotes the so-called *lvalue* of the variable designation). That identifier may then be used as shorthand for the variable designation within the statement list enclosed by the USING statement. An identifier declared in a <use spec> may also be used in the variable designation of any <use spec> following it textually.

²⁰This is especially useful when the named loop is not the loop immediately enclosing the EXIT statement.

²¹This value is also considered to be a good candidate to be placed in a register.

Example 10.2.4:

```

using inOut @ WriteString; s1 for s1 [150] ->. aj for a [j] do
  inOut @ WriteString s1 [149] ->.
  inOut @ WriteString s2.
  inOut @ WriteChar (s2 [80])
  s2 := string80["thanks for all the fiche.", ' ':55]
  s2 [1] := 'T'
  aj.imaginarypart := 0.0
end using
    
```

10.2.5. REGION Statements

The REGION statement implements a *critical region* protocol for mutual exclusion on execution of a region (list of statements). In the header of the REGION statement, the programmer specifies a variable designator on which the statements enclosed by the REGION statement will operate:

```

<compound stmt>   ->   <region stmt>
<region stmt>     ->   "region" <variable> "do"
                   <stmt list>
                   "end" "region"
    
```

The type of the entity designated by <variable> must have the *shared* attribute (see section 6), which indicates that access to the entity may be safely shared among concurrent processes. To ensure safe access, a *shared* entity may appear as the target of an assignment only within a REGION statement designating that entity.

The effect of a REGION statement is to associate the enclosed statement list with a semaphore which is also associated with all other REGION statements having the same variable designator. The first process to enter the region when the semaphore is free will then gain exclusive access to the region; others attempting to enter the region will be forced to wait in a queue on the semaphore. When a process leaves the region, it signals the semaphore so that the next process in the queue gains access (in a first in—first out manner).

Example (see section 9.1):

```

region a [j] do
  a [j].realpart := 10.5
  a [j].imaginarypart := 0.2
end region
    
```

11. Procedures

The procedure construct provides a type of control abstraction known as *procedural abstraction*. A statement list may be associated with an identifier by means of a *procedure declaration*; then, the use of that identifier in a *procedure call* statement implies the activation of that statement list, with the possible substitution of *actual parameters* for *formal parameters*. Also, a procedure may be declared as *value-returning*, in which case the procedure may be activated within an expression; the *return value* of the procedure call may then be used by the expression for further computation.

11.1. Procedure Declarations

A procedure declaration consists of a *procedure header* and a list of statements enclosed by a *procedure block*. The header contains declarations of the procedure's name and (optionally) its formal parameters, return type, and procedure attributes; the block may contain, besides the statement list, any *local declarations* of constants or variables. The procedure block may be replaced in the declaration by the keyword *forward*, which indicates that the procedure block will appear in a second declaration of the procedure which must appear later within the same

<proc decl>	→	<proc hdr> "forward"
<proc decl>	→	<proc hdr> <proc block> "procedure"
<proc hdr>	→	"procedure" <proc id decl> "(" [<params>] ")" ["returns" <return type>] "is" [<proc attr>]
<params>	→	<formal parm section> {',' <formal parm section> }
<formal parm section>	→	<id decl list> ":" ["const"] <type id>
<proc attr>	→	"action"
<proc attr>	→	"inline"
<proc attr>	→	"pure"
<proc block>	→	[<proc decl pt>] <stmt pt>
<proc decl pt>	→	<proc declaration> { <proc declaration> }
<proc declaration>	→	<const or var decl>
<proc declaration>	→	<type decl>
<proc declaration>	→	<constraint decl>
<proc declaration>	→	<var address decl>
<stmt pt>	→	"begin" <stmt list> "end"

Syntax of Procedure Declarations

compiland; the specification of parameters, return type, and attributes must appear in the procedure header in the so-called *forward declaration*, and may not be repeated in the procedure header of the second declaration.

The visibility of constants and variables declared locally to a procedure, as well as the visibility within a procedure of items declared in a procedure's environment, was discussed in section 4.1. The values of locally-declared variables are undefined upon entry to the procedure unless these variables have an associated initialization clause. Note that a procedure may not be declared within the declaration of another procedure; that is, a procedure declaration may not be nested. The use of a procedure's identifier in a procedure call within its procedure block declaration denotes the recursive activation of the procedure.

The formal parameters declared in a procedure header act as "placeholders" in the procedure block for the actual parameters to be passed in a procedure call. At the time of a procedure call, the formal parameters are replaced by the corresponding actual parameters. There are two kinds of formal parameters, known as *constant* parameters and *variable* parameters. A constant parameter acts as a local constant to the procedure to which it is passed, with the value of the corresponding actual parameter as its value; a formal constant parameter may not appear as the target of an assignment statement. A formal variable parameter acts as a renaming of the corresponding actual parameter; any assignment to a formal variable parameter will be reflected in the value of the actual parameter. (This mechanism is known as *pass by reference*.) Formal parameters declared in a list prefaced by the keyword *const* in a procedure header are considered to be constant parameters; those declared without the keyword *const* are considered to be variable parameters. The type of a formal parameter may be any named type. The scope of a formal parameter is the same as that of the local variables of the procedure, that is, its scope is the extent of the procedure.

A procedure may be specified to have a return type, in which case it is called a value-returning procedure or *function*. The type of the return value may be any named type. The value to be returned must be specified by an expression in a RETURN statement (see section 10.1.5); the type of this expression must be compatible with the return type.

A procedure declaration may also specify certain *attributes* for the procedure. These include *inline*, which specifies that the compiler should insert the procedure code "inline" at the

point of the call to the procedure, rather than to compile an actual call to the procedure; and *pure*, which indicates to the compiler that the procedure does not modify any non-local variables or make any calls to other non-pure procedures.²² (Do we need the attribute *action* if we use the action-invocation semantics described in section 13?)

Example (see Appendices A and B for more examples):

```

procedure factorial ( i : const integer ) returns integer is pure
begin
  if i <= 1 then
    return 1
  else
    return i * factorial(i-1)
  end if
end procedure ! factorial !

```

11.2. Procedure Invocations

The invocation of a procedure may take place either as a *procedure call statement* (see section 10.1.2), or (if the procedure has been declared as value-returning) within an expression:

```

<proc call>   →   <proc id> “(” [ <parm list> ] “)”
<parm list>  →   <expr> { “,” <expr> }

```

The values of the actual parameters specified in a procedure call are evaluated before the call, and these values are substituted for the formal parameters within the called procedure. For constant formal parameters, the actual parameter may be an expression. An actual parameter which is substituted for a variable formal parameter must be a variable designator; the selectors for components of structured variables are evaluated before parameter substitution takes place (that is, before the procedure call). The type of each actual parameter must be compatible with that of the corresponding formal parameter, and the number of actual parameters must match the number of formal parameters for that procedure, unless a parameter has been specified as *arbitrary* (more to come on this).

Example:

```
factorial(2*j)
```

12. Objects

The object construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction.

An Aeolus object may also have parameters indicating, for instance, sizes or element types of the abstraction implemented by the object; thus, an object implementing a bounded stack abstraction may be parameterized by the element type and maximum number of elements of the stack. Then, various *instances* of the bounded stack object may be created (instantiated) with differing element types and sizes; the implementation of the object need not be concerned with details such as the element representation, and the programmer does not need to create new object types for each combination of element type and stack size. Support for such *generic objects* increases the level of abstraction available to the programmer, and makes possible the creation of libraries of reusable object types.

²²This attribute gives the compiler a hint that certain optimizations may be possible in this procedure. This attribute is used at the programmer's risk; that is, the compiler does not attempt to verify that the procedure is actually pure.

The object construct also provides a safe separate compilation mechanism. The separation of an object specification into a *definition part* and an *implementation part* allows checking across the interface to an object, as well as allowing the use of an object definition before the corresponding implementation part is finished (thus facilitating top-down design).

12.1. Object Definition Parts

The definition part of an object defines the interface of the object with other compilands. It specifies the attributes of the object itself as well as the constants, types, and operations which the object provides to other objects and to processes.

The header of an object definition part, besides declaring the name of the object, optionally specifies the *object attributes* and the *object formal parameters* for a generic object. One possible object attribute is indicated by the keyword *pseudo*, which indicates that the object being defined is a *pseudo-object*, the simplest form of object. A pseudo-object may not be instantiated, as a full-fledged object may be; its operation calls are qualified by its object type name rather than by an instance name.²³ Also, pseudo-objects may not use the action-management

<comp unit>	→	<obj def>
<obj def>	→	<obj def hdr> <obj visible decls> "end" "definition" "."
<obj def hdr>	→	"definition" "of" [<obj attrs>] "object" <obj id decl> [<generic option>] "is"
<obj attrs>	→	"pseudo"
<obj attrs>	→	"nonrecoverable"
<obj attrs>	→	<auto attr> [<auto attr>]
<auto attr>	→	"recoverable"
<auto attr>	→	"autosynch"
<generic option>	→	"(" <obj formal param list> ")"
<obj formal param list>	→	<obj formal param> {"", <obj formal param> }
<obj formal param>	→	<id decl> ":" <generic type>
<generic type>	→	<type id>
<generic type>	→	"type"
<obj visible decls>	→	[<uses option>] [<imports>] <decls&specs>
<uses option>	→	"uses" <imp list>
<imports>	→	"import" <imp list>
<imp list>	→	<misc id> {"", <misc id> }
<decls&specs>	→	[<visible decls>] [<op spec part>]
<visible decls>	→	<visible decl> {<visible decl> }
<visible decl>	→	<const decl>
<visible decl>	→	<type decl>
<visible decl>	→	<constraint decl>
<op spec part>	→	"operations" <op spec list>
<op spec list>	→	<op spec> {<op spec> }
<op spec>	→	<proc hdr> [<op effect>]
<op effect>	→	"examine"
<op effect>	→	"modify"

Syntax of Object Definition Parts

²³Pseudo-objects are thus much like *modules* in Modula-2; the calling mechanism for their operations is simpler than for the other object types.

(recovery) features of the Clouds system (see section 13). A more complex form of object is given by the attribute keyword *nonrecoverable*, which indicates that—as with a pseudo-object—the object being defined does not use the action-management (recovery) features of the Clouds system; however, unlike pseudo-objects, a nonrecoverable object may have multiple instantiations. An instantiation of a nonrecoverable object is created by a call to the operation *new* provided by object *standard*; a variable of that object type is passed as a variable parameter to *new*, and the variable may thereafter be used to qualify operation invocations on that object instance.

If neither of the *pseudo* or *nonrecoverable* attribute keywords is specified, the compiler assumes that the object makes use of the Aeolus/Clouds action and object management facilities (see sections 12.3 and 13). The simplest way in which the programmer may make use of these facilities is through specification of one or both of the “*auto*” attributes in the object definition header. Specification of the *recoverable* attribute keyword causes the compiler to generate code in the object's operations for fully automatic handling of object state recovery, while specification of the *autosynch* keyword causes code to be generated for automatic synchronization of object operation invocations based on programmer-supplied indications of operation effects (see below). Thus, the programmer may gain access to the action and object manage-

```
definition of nonrecoverable object bounded_stack
  ( size : unsigned, elem_type : type ) is
```

```
! Definition of a generic bounded stack object with size SIZE
! and elements of type ELEM_TYPE.
```

```
operations
```

```
procedure push ( elem : elem_type ) is modify
  ! Places ELEM on the top of the stack,
  ! if the stack is not full.
```

```
procedure pop () : elem_type is modify
  ! Removes the top element of the stack and returns it.
  ! The return value is undefined if the stack is empty.
```

```
procedure top_elem () : elem_type is examine
  ! Returns the top element of the stack without removing it.
  ! The return value is undefined if the stack is empty.
```

```
procedure empty () : boolean is examine
  ! Returns TRUE if the stack has no elements,
  ! FALSE otherwise.
```

```
procedure full () : boolean is examine
  ! Returns TRUE if the stack has SIZE elements,
  ! FALSE otherwise.
```

```
end definition. ! bounded_stack !
```

Example of an Object Definition

ment facilities of the Clouds system simply by specifying a few keywords.²⁴ However, in some cases the programmer may be able to use knowledge of the semantics of the object and its operations to program synchronization and recovery mechanisms more efficient than the automatic mechanisms supplied by the "auto" attributes. Automatic recovery involves checkpointing of the entire object state; automatic synchronization is based on a simple read-write model of operation interactions on entire operations. As will be discussed in section 13, Aeolus provides facilities that allow the programmer to specify which parts of the object state are to be checkpointed (recoverable variables), to access information about the states of actions and to change these states (via operations on the action manager), and to control the recovery process by specification of what is to be done during action events (action event handlers); also, the programmer may specify finer-grained locking mechanisms for greater control of synchronization (via the *lock* type; see section 6.2.3.8).

If an object is to be generic, the programmer must specify the generic formal parameters in the object definition header. Such a parameter may be of any named type, or it may be an identifier which is to be used within the object implementation as a type identifier (specified by the keyword *type* in place of a type name in the formal parameter specification). As stated above, these parameters may be replaced by actual parameters (in the form of expressions or type names) when a variable of that object type is declared; the values of the actual parameters then determine the sizes, element types, etc. of that instance of the generic object (see section 6.3).

Following the object definition header, the programmer may specify the names of other object definitions which contain constant or type specifications to be used in this object definition. The names of these objects are specified in either a *uses* clause (if the object whose definition is being imported is a pseudo-object) or in an *import* clause (for other kinds of objects). Definitions imported in an object's definition part are also available in that object's implementation part.

After any necessary imports are specified, the declarations of the object definition are given. These are called its *visible declarations* since the declarations are available publically to any object which imports the object definition. The visible declarations of an object may include specifications of constants, types, or operations, but not of variables. The specifications of the object's operations are listed following the keyword *operations*. Each specification consists of the procedure's header (see section 11.1), optionally followed by one of the operation effect keywords *examine* or *modify*, which indicate that the operation reads from or writes to the object's state, respectively. This information is used by the compiler to generate automatic read or write locking for each operation if the *autosynch* attribute is specified for the object. If no operation effect is specified, the compiler assumes that the operation neither reads nor modifies the object state, and thus no automatic locking is done for that operation.

12.2. Object Implementation Parts

The implementation part of an object provides the actual code for the operations of the object, as well as the definitions of any private constants, types, variables, or procedures needed by the object. The definition part of the object being implemented is implicitly imported by the implementation part; thus, the attributes, formal parameters, and public constant, type, and operation specifications provided by the definition part may not be repeated in the implementation part. Also, as mentioned in the previous section, any objects imported by the definition part are also available in the implementation part. The implementation part may import other objects as well via its own *uses* and *import clauses*. All constants, type definitions, and operations declared in the objects made available by any of these methods are visible in the implementation part; also, the names of these imported object types may be used as the types of variables declared in the implementation part. Such variables must be initialized by use of the operation *new* provided by object *standard*.

²⁴For more information on the mechanisms supplied by the Clouds system to support synchronization and recovery, see [Allc83b].

<comp unit>	→	<obj imp head> <block> <obj imp tail>
<obj imp hdr>	→	“implementation” “of” “object” <obj id> “is”
<event part>	→	“action” “events” <override list>
<override list>	→	<override> {“,” <override>}
<override>	→	<id decl> “overrides” <id use>
<block>	→	[<decls>] <stmt pt>
<decls>	→	<decl> {<decl>}
<decl>	→	<const or var decl>
<decl>	→	<type decl>
<decl>	→	<proc decl>
<obj imp tail>	→	“implementation” “.”

Syntax of Object Implementation Parts

If none of the attribute keywords *pseudo*, *nonrecoverable*, nor *recoverable* are specified in the definition header of the object being implemented, the programmer may give an optional *event part* in the object's implementation part. Event part specifications are described in section 13.2.

The <block> of an object implementation part must include full declarations of all operations specified in the object's definition part. As with the full (second) declaration of a forward-declared procedure, the parameter list of an operation is not given in its full declaration. Constants, types, or procedures declared in the <block> but not specified in the object's definition part are not visible to other compilands importing the object. Variables declared in the outer level of the <block> are global to the object, and are static (“own”) variables; that is, the values of such variables survive between calls to the object's operations. The global variables of an object are called collectively the object's *state*.

12.3. Object Operation Invocations

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself):

<obj op call>	→	<obj id> “@” <obj op id> “(” <parm list> “)”
<obj id>	→	<id use>

```
implementation of object bounded_stack
  ! ( size : unsigned, elem_type : type ) ! is
```

```
! More to come.
```

```
end implementation. ! bounded_stack !
```

Example of an Object Implementation

<obj id> → <structured var>

Invocations of pseudo-object and nonrecoverable object operations have semantics essentially like those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities, that is, all objects which are not pseudo-objects or nonrecoverable objects (the so-called "Clouds objects").²⁵ Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level. (More to come on operation invocation semantics.)

Examples (see previous two sections):

```
s1 : bounded_stack ( integer, 10 )
s2 : bounded_stack ( real, 5 )

if not s2 @ full () then
  s1 @ push (42)
elsif not s2 @ empty () then
  r := s2 @ pop () + 3.14159
end if
```

13. Actions

The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed. The rationale for the action concept and the mechanisms supporting it in the Clouds system are described in [Allc83b].

Support for actions in the Aeolus language is relatively low-level. The methodology of programming with actions is not at present well-understood compared with experience in programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable variables*, and *action invocations*.

13.1. Action Events

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action manage-

²⁵This is because the code for pseudo-objects and for nonrecoverable objects is actually linked into the code of the compiland using these objects, whereas the code for Clouds objects is physically separate from the code of the invoking compiland. This code is paged in on demand by the object manager (see [Allc83b]).

ment system. The *action events* include:

<i>event name</i>	<i>purpose</i>
BOA	beginning of action
toplevel_precommit	prepare for commit for a toplevel action
nested_precommit	prepare for commit for a nested action
commit	normal end of action (also called EOA)
abort	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described in section 12.1, by specification of the keyword *recoverable* in the header of an object definition the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires checkpoints of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if none of the attribute keywords *pseudo*, *nonrecoverable*, or *recoverable* are specified in the definition header of the object being implemented, the programmer may give an optional *event part* in the object's implementation part. Following the keywords *action events*, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override:

```

<event part>  →  "action" "events" <override list>
<override list> →  <override> {"", <override>}
<override>   →  <id decl> "overrides" <id use>

```

Thus, for example, the specification (say, in an object called "stack"):

```

action events
stack_BOA overrides BOA, stack_precommit overrides precommit

```

indicates that the default handlers for the *BOA* and *precommit* action events are to be replaced by the procedures named "stack_BOA" and "stack_precommit," respectively, for the "stack" object only.

13.2. Recoverable Variables

As mentioned in section 8, if an object being implemented is not a pseudo-object, nonrecoverable object, or (automatically) recoverable object, then some of its variables may be declared to have the *recoverable* attribute:

```

<new type>   →  "recoverable" <type> {"", <override>}
<override>   →  <id decl> "overrides" <id use>

```

The state of a recoverable variable which has been touched by an action is maintained on a *version stack* by a Clouds action manager, and is saved to permanent storage upon commit of the action which touched it. If an action which touched a recoverable variable is aborted, the version of that variable which existed before the action touched it is restored.²⁶ Thus, the use of recoverable variables allows the programmer to provide finer granularity in the specification of that part of the object state which must be checkpointed, since the use of automatic recovery on object (the *recoverable* object attribute) performs checkpoints on the entire state of the

²⁶For more information on the semantics of recoverable variables and the mechanisms to support them, see [Allc83b].

object.

The interaction with the action manager necessary to manage the states of recoverable variables is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance. However, to achieve better performance in the management of recovery for specific recoverable variables, the programmer may specify one or more alternate event handlers on a variable-by-variable basis which take advantage of semantic knowledge about the variable in question. Thus, the declaration of a recoverable variable may specify the name of an operation which *overrides* the default handler for an event. Override specifications for recoverable variables may coexist with override specifications for the entire object.

Example:

```
r : recoverable integer, r_precommit overrides nested_precommit
```

In the declaration of "r" above, the default handler for the *nested_precommit* action event is overridden (replaced) by the procedure "r_precommit" for management of the recovery of variable "r."

13.3. Action Invocations

As mentioned in section 10.1.1, the right-hand side of an assignment statement may also take the form of an *action invocation*:

```

    <rhs>      →  ["toplevel"] "action" "(" <action invocation> ")"
  <action invocation> → <proc call>
  <action invocation> → <obj op call>
  <action invocation> → <assign stmt>

```

Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. The action ID may then be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort. (The interface to the Clouds action manager is described in Appendix E.) If the keyword *toplevel* is specified, the action is created as a "top-level" action; that is, as an action with no ancestors.²⁷ Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors.

The semantics of an action invocation is as follows: the action manager operation *CreateAction* is invoked with the name of the operation to be performed as well as the list of arguments to be passed to that operation.²⁸ The action manager then invokes the BOA event handler on the object to which the operation belongs. Next, the action manager creates and dispatches a process in which the operation code runs. An attempt by the operation to return to its caller is considered an implicit attempt to commit the action, and will cause control to transfer to the *Commit* operation of the action manager, which terminates the process and invokes the precommit event handler of each object touched by the action. (An explicit invocation of the *Commit* operation has the same effect.) If precommit of the object is successful, the action manager then invokes the commit event handler of each touched object. If the action (or one of its ancestors) invokes the *Abort* operation of the action manager, the action manager terminates the process corresponding to the action and invokes the abort event

²⁷Thus, as we shall see, a top-level action cannot be affected by an abort of any ancestor of the action which created it.

²⁸The exact details of the manner in which this information is provided depends on whether the operation is a local procedure or a publicly-visible operation of the object to which it belongs.

handler of each object touched by that action.

It may sometimes occur that an object operation may be called either as an action invocation or as an ordinary object operation invocation. In the case that an operation is invoked normally (that is, not as the target of an action invocation), an invocation of the action manager operation *Commit* by the operation will cause the action manager to merely return control to the point of invocation of the original operation; thus, in this case the *Commit* call is effectively a normal procedure return. On the other hand, an invocation of the *Abort* operation by an operation invoked normally will cause the parent action of the invoker of the original operation to abort.²⁹ Thus, in the case of normally-invoked operations, a call to the *Abort* action manager operation provides a mechanism similar to an exception-handling mechanism with a single exceptional condition ("error").

14. Processes

The final structuring feature of the Aeolus language provides an abstraction of the *process* concept of the Clouds system. (The process is analogous to the *program* construct of Pascal or Modula-2.) The invocation of a process provides activity in the Clouds system; processes may be considered the "glue" which binds object operations, and possibly actions, to do useful work.

A process is introduced by a header which gives the name of the process, as well as clauses detailing any imports of object definitions necessary (see section 12.1):

```

<comp unit>   →   <prog head> <block> <prog tail>
<prog head>  →   <prog hdr> <uses option> <imports>
<prog hdr>   →   "process" <prog id> "is"
<prog tail>  →   "process" "."

```

Following any import clauses, the body (<block>) of the process is specified; the <block> has the same form as that of an object implementation part (section 12.2).

²⁹Note that all processes in the Clouds system are descendants of the top-level "universal action," which cannot be aborted.

```

process test_bounded_stack is

import bounded_stack

bs1, bs2 : bounded_stack ( 10, integer )

i : integer := 0

begin
  new (bs1)  new (bs2)
  loop
    if bs1 @ full () then
      exit .
    end if
    bs1 @ push (i)
    if (i % 3 = 0) and not (bs2 @ full () or bs1 @ empty ()) then
      bs2 @ push (bs1 @ pop ())
    end if
    i += 1
  end loop
end process. ! test_bounded_stack !

```

Example of a Process (see section 12.1)

15. REFERENCES

- [Allc82] Allchin, J. E., and M. S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982
- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also available as technical report GIT-ICS-83/23)
- [LeBl85] LeBlanc, R. J., and C. T. Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)
- [McKe84a] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," School of Information and Computer Science, Georgia Institute of Technology, May 1984
- [McKe84b] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)
- [Pitt84] Pitts, D., "Storage Management for an Action-Based Operating System," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of

Technology, November 1984 (also available as Technical Report GIT-ICS-85/02)

- [Wilk86] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, in progress

QUARTERLY PROGRESS REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT #MDA 904-84-C-6035
REPORTING PERIOD: 1 JULY 85 - 10 SEPT 85

1. Project Status

During the fourth quarter of this project, work has continued on each of the two tasks called for by the statement of work. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing. Under the Distributed File Systems task, work has proceeded on the integration of our storage management system with the Clouds kernel virtual memory management system. Additionally, implementation work has been done on several device drivers necessary to test the kernel and storage management system.

Under the Language Support for Robust Distributed Programs task, we have been refining the definition of our language, Aeolus, and proceeding with the implementation of the compiler. We have also been actively working with members of the Clouds kernel group on the definition of the interface between the Aeolus run-time system and the Clouds action and object managers.

The work on the tasks of this project has proceeded on schedule. These efforts, in combination with other work in progress within the Clouds project, have kept us on target toward our goal of having a working prototype system by the end of 1985.

2. Storage Management - Progress Report

The development of the Clouds storage manager involves the implementation of three components. These are the device object, the partition object, and the segment object. These objects are each abstractions of the disk storage available on a Clouds machine. The device object manipulates device storage as a collection of uninterpreted blocks of data, which it will transfer in and out of virtual memory. The partition object provides a mechanism for division of device storage for administrative purposes and also is involved in the location of data and the allocation of device storage. The segment object treats device storage as a collection of bytes. In fact, the segment object is just an alternate view of any Clouds object. We consider the device object the lowest level of abstraction and the segment object represents the highest level. In the paragraphs that follow, we describe the current state of the storage manager.

At the device object level, we are developing two disk objects. Clouds disk objects include not only the conventional device driver functions, but also provide necessary support for the recovery mechanisms of the storage manager. Specifically, the Clouds disk objects provide a mechanism, *flush routine*, which insures that requests scheduled by an action are actually completed before the action commits. This mechanism differs from conventional disk management schemes, where a request may remain enqueued after the process that issued it terminates. The flush routine relies on the *flush table*, a *per device* structure. The table contains an entry for each action; the entry contains a list of requests made by the action and a record of the number of requests pending and completed. The development of a RL02 disk object has been straightforward and we now have a working version integrated with the Clouds kernel. Minor changes in the way the object formats the medium are anticipated. Additionally, the object must be modified to lock physical pages for I/O transfers, because of the Clouds kernel's use of the virtual memory system as the basic I/O mechanism. The RL02 will allow us to go ahead with the development of kernel and particularly with the testing of the storage manager. The RL02 will not be the primary disk for the Clouds system, as it holds only 10 Mb on each cartridge. The primary disk for the initial Clouds implementation will be the RA81, a disk object for which is under development in parallel with the development of the RL02 object. Because the RA81 is a "smart" device, progress has been slower and the integration of the facilities required by the Clouds storage manager is more complex. Testing is currently under way on this device. We have kept the device object interface for the two devices uniform and also have attempted to reduce any side-effects so that upon completion of the RA81 object, this object can be use in the place of or along with the RL02.

The next level of abstraction for the storage manager is the partition level. Implementation at this level is just being completed. A partition provides all the structures required to support the creation and management of Clouds objects. Specifically, the partitions provide support for the location of objects and the allocation of disk storage for objects. The Clouds kernel provides for the location-independent invocation of object operations, which requires the kernel to search for the objects at each operation invocation. Object searches are network-wide and several techniques are being developed to short-circuit these searches. One concern is the necessity of going to disk in order to determine if the object is on a partition. Each partition maintains a structure called a *maybe table* which is intended to reduce the number of unnecessary

disk accesses during object search (an unnecessary access is one which shows the object is not on the partition in question). The maybe table is a small in-memory representation of the partition membership. Our maybe table is an example of a Bloom filter [Bloom70]. The table is a compressed hash table, in which several segment names may hash to the same entry. In trade-off for the reduced size of the table, only a negative response to a query is guaranteed to be correct. Responses indicating the object is on the partition may in fact be wrong and may require the partition object to access the directory on disk. We are really trading accuracy of the responses for speed since in most cases the query can be answered without an unnecessary access to disk.

The segment system forms the highest abstraction provided by the storage manager. Disk storage at this level is managed as a collection of arbitrarily sized segments, which generally represent some Clouds object. Segments provide a uniform interface through which the Clouds kernel can manipulate objects. In addition, the segment system provides a set of protocols which insures the consistency of the permanent object data when manipulated by some action. Implementation of the segment system is currently in progress. Recovery is provided using a pessimistic shadowing scheme, in which modifications are stored in a temporary version of the segment until the action making the modifications commits, making the temporary version the new permanent segment. The segment system, along with the action management and object management systems are involved in the management of virtual memory with respect to the mapping of objects. We are finalizing the extent of each system's responsibilities and influence in the virtual memory and the cooperation needed between the systems. The storage manager shares with the object manager the responsibility for mapping the on-disk version of the segment to the virtual memory version. Each segment has one or more windows which represent chunks of the segments which are actually mapped into virtual memory. This allows the mapping of portions of large objects into virtual memory, avoiding the cost of mapping the entire object. The storage manager also makes use of the virtual memory system to assist the action management system in the committing of actions. The segment system uses virtual memory structures to determine which segment pages have been modified and then, based on its own information as to the structure of the segment, decides which segment pages must be shadowed to provide the necessary recovery.

A technical report [Pitts85] which summarized all of the work which has been done on the storage manager is attached as Appendix A.

[Bloom70] Bloom, B.H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, No. 13, Vol. 7 (July 1970), pp.422-426.

[Pitts85] Pitts, D.V. and E.H. Spafford, "Notes on a Storage Manager for the Clouds Kernel" Technical Report GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, October 1985.

3. Aeolus - Progress Report

As part of the Clouds project, we are designing and implementing a high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology) in which those levels of the Clouds system above the kernel level will be implemented. *Aeolus* provides access to the synchronization and recovery features of Clouds. It also provides a framework within which to study programming methodologies suitable for action-object systems such as Clouds.

The work of the *Aeolus* group during the past quarter has been concentrated on the completion and rationalization of the language design and—in conjunction with members of the kernel group—on the definition of the interface of the *Aeolus* runtime system with the Clouds action and object managers. The design of the language has undergone a major reworking, especially those parts of the design concerned with specification of types. In view of one of the *Aeolus* design goals of providing the power necessary for systems programming without sacrificing the advantages of strong type checking, we wished to provide dynamic (flexible) data types; however, we felt that our previous design for this violated the goals of simplicity and readability. Our reworked design integrates flexible types into the language in a much cleaner manner, within the framework of general parameterized types. The changes to the design have been incorporated into the language description [Wilk85b], which is now essentially complete; the interfaces with the Clouds system object and action managers remain to be specified in the language description. The new language definition is attached to

this report as Appendix B.

Work on the implementation of a compiler for Aeolus has recently resumed following the absence over the past quarter of the team member responsible for implementation of the symbol table and code generation routines. The design changes have now been incorporated into the symbol table, and new semantic routines necessitated by the changes are being implemented. We have also taken advantage of the redesign to streamline parts of the semantic routine structure, taking into account our previous implementation experience. Work on the implementation is accelerating now that these changes are understood.

The design of the interfaces of the runtime system with the Clouds action and object managers is essentially complete. Members of the Aeolus group have been assisting members of the kernel group over the past quarter in the design of these interfaces as well as in strategies for efficient action management. Of particular importance were our designs for support of recoverable areas in Clouds objects; these constructs enable the Aeolus language (in conjunction with the action management system) to provide *view atomicity* in addition to the *failure atomicity* provided by the kernel. Each action which touches an object which has a recoverable area gets its own copy (or *version*) of that recoverable area on which it may make its changes; when a nested action commits, it propagates its version of the recoverable area to its parent. View atomicity ensures that each action in the action tree which accesses an object sees the correct version of the data in that recoverable area. We have developed a technique for implementing recoverable areas using partial replacement of the object page table entries which provides view atomicity without causing a time penalty for access to the data in the recoverable area. Rather, a small penalty is paid upon process exchange if a process is associated with an action. The action and object managers exist now in pseudo-code; the interfaces with these kernel routines will shortly be codified as appendices to [Wilk85b].

We intend to use Aeolus as a framework within which to study programming methodologies for action-object systems. Our initial studies in programming methodologies for resilience and availability are described in [Wilk85a]; there, a plan is presented for determining such methodologies appropriate to the design of objects needed in the Clouds system. The issues with which we are concerned include the use of semantic knowledge of objects in the programming of replication; trade-offs

between consistency and availability; the appropriateness of current programming models for replicated data; and the support needed from the operating system and language runtime system to ensure availability and forward progress of processes. Now that the language design is complete and our implementation effort is well underway, we plan to devote proportionally more effort to these studies in the immediate future.

[Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985.

[Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, July 1985.



GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE • ATLANTA, GEORGIA 30332 • (404) 894-3152

October 22, 1985

To whom it may concern:

The attached paper, Notes on a Storage Manager for the Clouds Kernel, by David V. Pitts and Eugene H. Spafford which was partially funded under the contract #MDA 904-84-C-6035 is being submitted for your information. This paper will be published as a Technical Report in the School of Information and Computer Science at Georgia Institute of Technology.

Sincerely,

Richard J. LeBlanc
Associate Professor
RJL/kkh

**Notes on a Storage Manager
for the
Clouds Kernel**

Technical Report GIT-ICS-85/02

*David V. Pitts
Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

CONTENTS

1. Background	2
2. Hardware and Environment	4
3. The Design Approach	6
4. Device Objects	8
4.1 Device Media	8
4.2 Device Object Structures	8
4.3 Device Object Calls	10
5. The Partition Object	12
5.1 Partition Data Structures	13
5.2 Calls on the Partition Object	16
6. The Segment Object	18
6.1 Segment Object Data Structures	18
6.2 Calls on the Segment Object	18
7. Reliable Storage Management	21
7.1 Segment level recovery	22
7.2 Partition level recovery	26
7.3 Device support for recovery	28
7.4 Summary	29
8. Conclusions	30
REFERENCES	31

LIST OF FIGURES

Figure 1. Architecture of the Clouds kernel	2
Figure 2. Clouds hardware configuration	4
Figure 3. The system device table and other device object structures	9
Figure 4. The system partition table and other partition object structures	13
Figure 5. Two implementations of a Bloom filter	15
Figure 6. Clouds kernel segment structure	19
Figure 7. Actions block on competing commits	21
Figure 8. Precommitted segment	23
Figure 9. A committed segment	24
Figure 10. An aborted segment	25

**Notes on a Storage Manager
for the
Clouds Kernel**

Technical Report GIT-ICS-85/02

*David V. Pitts
Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

Abstract: The Clouds project is research directed towards producing a reliable distributed computing system. The initial goal of the project is to produce a kernel which provides a reliable environment with which a distributed operating system can be built. The Clouds kernel consists of a set of replicated sub-kernels, each of which runs on a machine in the Clouds system. Each sub-kernel is responsible for the management of resources on its machine; the sub-kernel components communicate to provide the cooperation necessary to meld the various machines into one kernel.

This report documents a portion of that research, namely, the implementation of a kernel-level storage manager that supports reliability. The storage manager is a part of each sub-kernel and maintains the secondary storage residing at each machine in our distributed system. In addition to providing the usual data transfer services, the storage manager ensures that data being stored survives machine and system crashes, and that the secondary storage of a failed machine is recovered (made consistent) automatically when the machine is restarted. Since the storage manager is a part of the Clouds kernel, efficiency of operation is also a concern. We wish to reduce the overhead required to ensure the recoverability of secondary storage as much as possible, while adhering to the design goals associated with the storage manager.

1. Background

In this section we present an overview of the Clouds kernel and discuss the philosophy behind its development. The Clouds approach to providing reliability is through the use of actions and objects, as discussed in [1], [2], [3], [4]. The Clouds kernel provides higher level applications such as operating systems with three primitives: *processes*, *actions*, and *objects*. An *object* is a typed collection of data which is manipulated by a set of operations. The data structures and the set of operations for the object define its type. An *action* is the unit of (fault tolerant) work in the Clouds system. Actions guarantee failure atomicity of the operations performed by them: the operation appears to either occur totally or not at all. *Processes* in Clouds are similar to processes found in other systems, and represent a thread of execution and control. Actions and objects are passive, waiting for a process to activate them. The model of computation for the Clouds system is that of a set of processes making operation calls on objects to perform services required by the system. In order to make these services reliable, the object operation calls are performed under the auspices of an action.

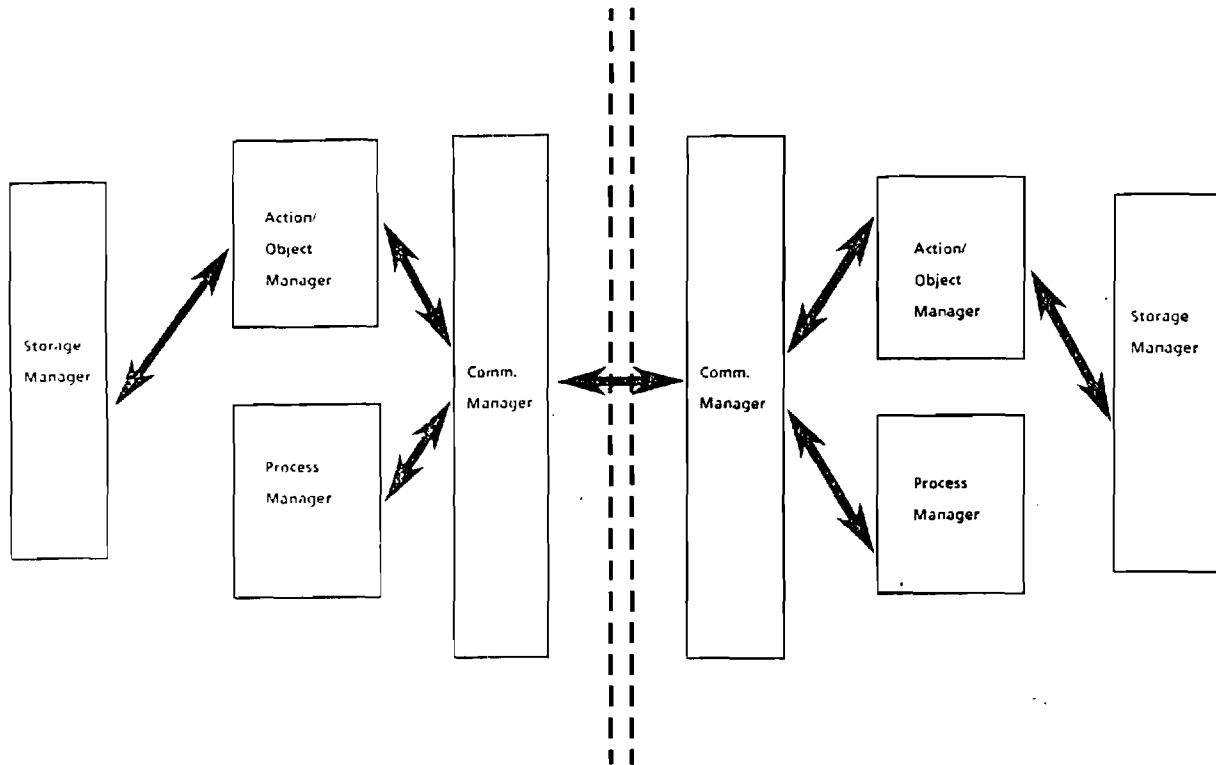


Figure 1. Architecture of the Clouds kernel

Clouds actions provide a mechanism that allows the programmer to violate the conventional notions of correctness and consistency, as defined by strict serializability, when programming reliable objects. The programmer can use any semantic knowledge about the intended activation of an object to program a customized method for providing the recovery of the object. This is done by the programmer writing new *abort* and *commit* operations for the object, which indicate how the object data must be recovered. By allowing object recovery to be customized in this way, we hope to provide increased concurrency in the execution of services compared to using the usual recovery and synchronization rules (i.e., serializability), and so improve the performance of the Clouds system.

The Clouds kernel has four major components: the object/action manager(s), the process manager, the communications manager, and the storage manager. Figure 1 depicts the architecture of the Clouds kernel for a system consisting of two nodes. The kernel is made up of two sub-kernels, one of which resides on each node that is part of the Clouds system. Each of the components of the kernel can communicate with its corresponding components on other nodes through the proper protocols.

The object manager is responsible for providing the object operation invocation mechanism. Each object is named by a unique capability comprised of a system name (called a *sysname*) and a series of rights which indicate which object operations are available to the invoking process. The object manager checks the capability provided by the operation call, locates the desired object instance, formats and maps the operation parameters, and activates the object. The object manager is involved with handling action bookkeeping, as necessary. The object manager also hides references to objects on other machines by providing a remote procedure call mechanism (RPC). The object manager makes an RPC look exactly like a local operation call.

The process manager creates, destroys, and dispatches processes. It manages local processes, as well as slave processes started to handle RPC's from other machines. The process manager is not a global scheduler; it simply manages local resources.

The communications manager is responsible for the transmission of information among the machines in the Clouds network. It maintains information about the connectivity of the network, the status of the various lines to which each machine is connected, and queues of outgoing and incoming data. The data that goes through the communications manager is uninterpreted — it might be an RPC or a part of a file that is being transmitted across the network. More detailed descriptions of the object, process, and communications managers can be found in [5] and [6].

The function of the storage manager was described above. It coordinates with the object manager to provide the correct commits and aborts of actions on object data residing on secondary storage. In the remainder of this report, storage will refer to the secondary storage (disk, tape, etc.) attached to a machine. Memory will refer to the main memory of the machine.

2. Hardware and Environment

The Clouds kernel is currently under implementation on three VAX 11/750's.¹ The machines have eight megabytes of main memory altogether and are interconnected via a 10 Mb/sec Ethernet. Also connected to the Ethernet are a set of IBM-PC's, which will serve as intelligent work stations. Future versions of the system may be connected by multiple networks of varying bandwidth.

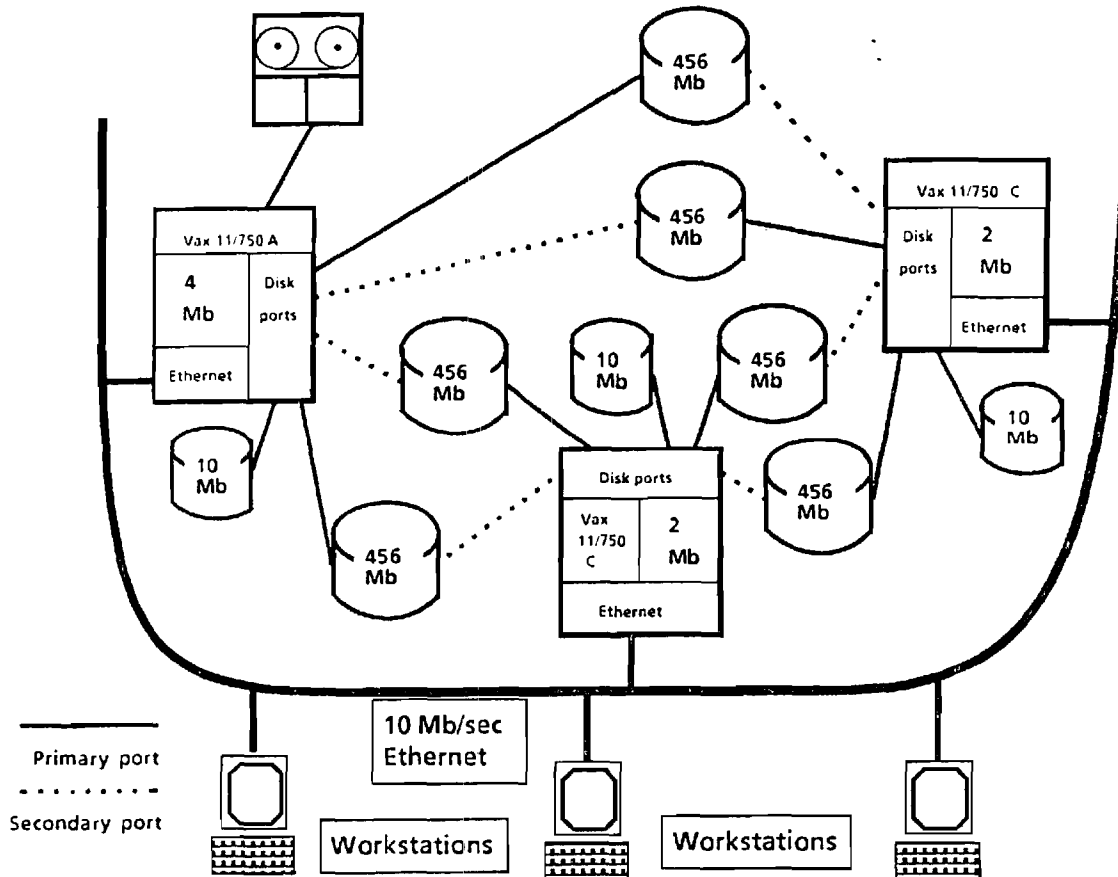


Figure 2. Clouds hardware configuration

Our prototype will have three types of storage devices available for the kernel. There may be a tape drive on one machine that will be used to archive data and perform conventional system backups. Each machine will have a RLO2 removable pack disk drive, in which each pack provides 10 Mb of storage. We expect that RLO2 media will be used as short term archive devices and boot devices. Finally, each system will also have up to four RA81 disk drives. Each such drive has a permanently mounted pack providing 456 megabytes of storage (unformatted). The RA81 drives are dual-ported; two controllers may be coupled to the drive simultaneously. However, the drive is on-line to only one of the controllers at any time. The switching of the device between controllers is done primarily by a front panel switch, but switching can be done under program control. The disks are controlled by UDA50 controllers

1. VAX is a trademark of the Digital Equipment Corporation

which use DEC's Mass Storage Control Protocol (MSCP). These devices are expected to provide the primary object storage for the Clouds system. Figure 2 shows the Clouds prototype system.

3. The Design Approach

The Clouds kernel provides user-defined objects² as the building blocks (along with atomic actions) of a reliable distributed system. The arguments for using an object-oriented approach in general, and as used in the Clouds project in particular, are presented elsewhere [7] and [3] and we will not repeat those rationales here. We feel that the kernel, in addition to supporting objects for higher levels of software, should also reflect the use of an object-oriented methodology in its design and implementation. To this end we have identified basic components of the kernel and isolated them as modules that are accessible only through a set of procedures defined for each module. These objects are then used to form the major systems of the kernel: the object manager, the process manager, the communications manager, and the storage manager.

We attempt to present kernel objects as typical Clouds objects that provide (restricted) access to functions and services provided by the kernel. However, there are differences between the objects that form the kernel and those that are supported by the kernel. The first such difference is in the implementation. User-defined objects will be created by users with an object-oriented language, such as Aeolus [8], [9]. This language enforces the use of an object-oriented methodology. Our kernel objects are currently implemented as C modules and most of the responsibility for adhering to the philosophy of object-oriented design is the responsibility of the programmer, not the programming language. Still, we believe the careful use of this object methodology despite the lack of support in the language provides benefits in the implementation and later maintenance of the kernel. It also may make the later conversion of the kernel to some other object-oriented language, such as Aeolus, more convenient.

The other difference reflects our concern for the efficiency of kernel functions and the operation invocation mechanism for objects. Many of the functions of the kernel are time-critical, or because of their frequent use require very efficient implementations. The operation invocation mechanism has overhead that we suspect cannot be afforded in these situations. Therefore, operation calls on kernel objects are handled differently than operation calls on user-defined objects. Calls on kernel operations may be performed as ordinary procedure calls or even as simple transfers to blocks of code. However, the appearance outside the kernel and the overall philosophy is that of an object-oriented kernel.

Some kernel objects are not generally available outside the kernel. For example, this is the case with the objects comprising the storage manager. Operating system code may occasionally require direct access to secondary storage, but it is hoped that for the most part the abstractions provided by objects will suffice.

The storage manager is based on three sets of objects: device objects, partition objects, and segment objects. Each of these objects manages the same actual item (secondary storage), but provides different abstractions. The device objects provide conventional device-level access to secondary storage. Partition objects allow devices to be sectioned logically according to the intended use of the storage on a device. Segment objects are the secondary storage extensions of the segment type provided by the kernel. Recoverable permanent objects are implemented at the level of segment objects.

The remainder of this report outlines a design for a storage manager for the Clouds kernel. It covers the important aspects of the structure and function of the storage manager, and discusses how the storage manager is used by and uses other parts of the kernel. The next three sections deal with the design and implementation of the device object, the partition object and the segment object. Those sections specify the data structures required plus the interface to the

2. Also referred to as *client objects*.

objects. Section 7 then covers how these objects are used by the kernel. In that section we discuss some of the issues related to the reliability of the storage manager and its relationship to the rest of the kernel. Section 8 contains a summary of this report, and a few conclusions and reflections on the storage manager.

4. Device Objects

As with conventional systems, the storage manager for the Clouds kernel provides a device level interface to secondary storage. This level of interaction with secondary storage is almost exclusively the province of the Clouds kernel. In fact, even within the kernel, most accesses to secondary storage are performed at some other (higher) level; only the storage manager makes frequent use of device objects.

4.1 Device Media

The storage manager views devices as two parts: the device itself and the medium currently being used by the device. This viewpoint is moot for fixed media disks, but for other forms of secondary storage, such as tape and removable disk storage, it provides additional flexibility in the configuration of a system. This separation is complete; a sysname exists not only for each device in use on a system, but also for each medium. However, in many cases the distinction between accessing specific media and accessing devices is not important, so we wish to hide this separation. Therefore, the storage manager provides a mechanism for binding a medium to a device.

Bindings between media and devices are generally performed at the initialization of the system and involve the association of device and medium. Binding a medium to a device may also involve the formatting of the medium. In this latter case, a new sysname is generated for the medium. This formatting or initialization of a medium will destroy any previous information that existed on the medium. The old sysname will no longer give access to any medium. The formatting of a blank or obsolete medium includes initializing the tables and structures that the storage manager requires. These structures are discussed in section 2.1.

In other cases, an existing medium is bound to a device. An existing medium is one which has a sysname and is formatted. The binding will involve the reading of the sysname from the medium and comparing it with the sysname passed to the storage manager. The binding will take place only if a match occurs. We are not attempting to address security issues with this design. Our interest is to provide flexibility, while maintaining some control over what is accessible. The use of sysnames to access media provides this control.

Once a medium has been bound to a device, any reference to the device refers to the bound medium. The usual sort of device calls then need only refer to the device. This device-medium binding stays in effect until it is explicitly broken by the storage manager.

In addition to setting up this correspondence between device and medium, this binding also initializes an instance of a storage object in memory. In particular, critical tables and other structures required by the device are brought into system memory. We will now look at the data requirements of device objects.

4.2 Device Object Structures

The storage on a medium is presented as a sequence of 512-byte blocks that are addressed by offsets from some fixed block. The offset used to address a block is called a device block number (DBN). As we shall see in section 5, this storage can be subdivided into partitions, allowing the storage on a device to be apportioned for policy reasons. At the device level, though, the storage manager deals only with a contiguous string of blocks; partition boundaries are not visible.

The device object is responsible for the transfer of data between secondary storage and memory. The device requires two tables in order to function. The first such structure is the media header. This table contains basic information about the medium and the device using it. This information includes the medium and device sysnames, the amount of available storage on the medium, and specifications for the device to which the medium is bound.

The other major structure is the index table. The index table describes the partitions that exist on this device. This will include information such as the location, extent, and type. The partition table is 16 entries long. Partitions are discussed in section 5.

The medium header and index table must be resistant to failures — in particular, device failures such as head crashes. If the index table is destroyed by a head crash, for instance, we lose access to the partitions on that medium. We therefore provide copies of the tables, placing the copies on different cylinders in order to minimize the risk from multi-sector failures. The alternate copies will be located in known positions based on some computable function. We do not anticipate problems as far as maintaining the consistency of the slave and master versions of the table is concerned, since the tables are infrequently modified and any such modifications are generally done during the system initialization.

In addition, the device objects will maintain a structure in memory called a flush table. The use of this table is discussed in section 7, but briefly, the flush table allows a device to associate an action sysname with a set of requests. This supports the commit operation performed on recoverable objects. Some devices may require the device object to provide bad sector recovery. Objects written for these devices will have to maintain a bad sector table on disk.

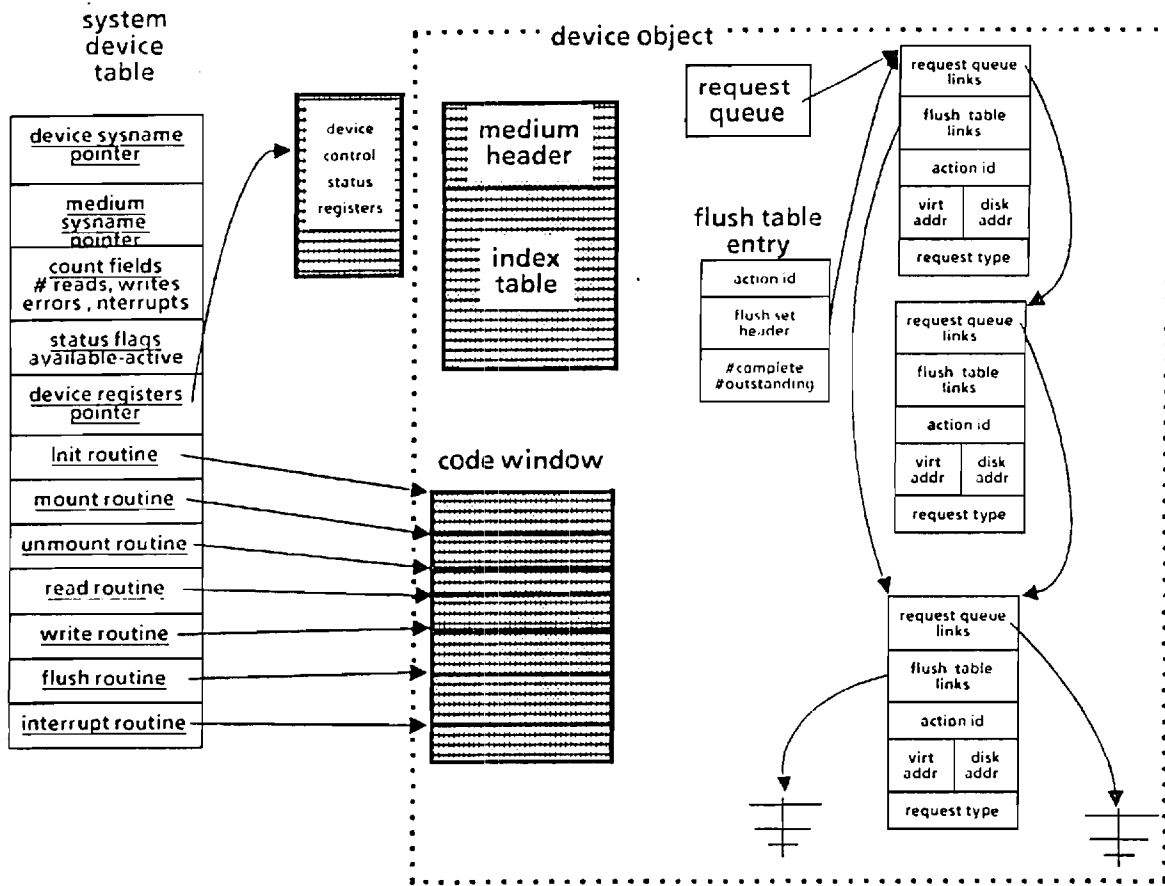


Figure 3. The system device table and other device object structures

The device object uses one other structure, the system device table. The system device table is not a part of the device object proper, but is actually the mechanism for managing the various instances of the device objects. This table lists all secondary storage devices that are active on the local machine. The device table entries contain pointers to device and medium sysnames, status variables for the device, device registers, and entry points into the operations for the

device object. Figure 3 shows a device object pointed to by one entry from the system device table.

4.3 Device Object Calls

The device object calls deal with the transfer of information to and from the device and with the relationship of the device to its medium. This allows for devices switching the physical medium used for storage in a uniform way. Device and media sysnames are generally needed by those calls setting up a binding between the medium and the device. Calls which perform i/o do not require a sysname. The proper device object calls are found through the system device tables.

4.3.1 *init(devname) return medname*

Init generates a sysname for the medium currently on the device specified by **devname**. This sysname is written in the medium header. Also written into the medium header is the device specific information that is required. An area for the medium index table is reserved. The return value is the medium. This is a format call; any existing structure on the medium is overwritten. No other formatting is done, however. Any partitions desired are created later by other calls. Redundant copies of the medium header and index table are created for reliability.

After the medium has been formatted, **init** mounts the device. See the description of **mount** for details.

4.3.2 *mount(devname, medname) returns integer*

This call binds the device called **devname** to the medium called **medname**. The sysname presented to the call is compared to that in the medium header. If the two match, the device and medium are bound. The medium index table and the medium map table are read from the disk. If the device requires it, a bad sector table is created from the device. The return value specifies the status of the call (success, failure).

4.3.3 *return_medium_cap(devname) returns medname*

This call returns the sysname of the medium that is bound to the device named **devname**. The return value is this sysname. If the device is unbound, a valid sysname might still be returned if a formatted medium is present in the device. In this case, the call can be seen as an operation to read a label.³ This allows us to use media for which all currently existing copies of the sysname are deleted or unavailable.

4.3.4 *unmount(devname) returns integer*

Unmount breaks a device/medium binding. All partitions on the medium are de-activated. The return value is the status of the call.

4.3.5 *read(lbn, address) returns integer*

This call transfers the contents of a record located at disk address **lbn** to the page in memory at **address**. **Read** blocks the calling process on a semaphore until the request is complete and returns an integer indicating success or failure of the request.

4.3.6 *write(id, lbn, address, flag) returns integer*

This call transfers the contents of a page in memory at location **address** to the record located at address **lbn** on the device in question. **id** is an identification used to associate this request with a set of requests being issued by an action. If **id** is an action sysname, then the device object looks the action **id** up in a flush table and if it is not there, creates an entry for the action and the request; if the action **id** is in the table, the request is added to that entry. If **id** is zero, then there is no action associated with this request. **Flag** is used to indicate whether this is a forced

3. This kind of operation might seem to present a security hole in the system, in that it allows the system to determine the name of an unknown medium and then mount it. However, note that this call can only be executed by kernel code or by a user call with special kernel capabilities, and these are assumed to be trustworthy.

write. If flag is non-zero, the device interrupts the normal scheduling of requests by placing this request at the head of the queue. The new request is performed immediately after the current request is completed. A forced write blocks the calling process on a semaphore until the request is complete. Non-forced writes do not normally block the caller.

4.3.7 flush(id) returns integer

Flush uses the flush table maintained by the device object to ensure that all write operations associated with the action identified by **id** are actually completed. The return value indicates the status of the call. A positive return value (the number of requests completed) indicates a successful flush. A zero or negative return value indicates that the action's sysname was not found in the flush table or that some error occurred while attempting to flush the specified requests.

4.3.8 enter(partname, size) returns lbn

Enter registers a partition on the device. This involves making an entry for the partition in the index table for the device, placing the partition sysname, **partname**, and the partition size, **size**, in the entry, and allocating storage on the medium for the partition. The starting logical block number for the partition is placed in the index table and is returned as the value of the call. A negative return value indicates that an exceptional condition occurred, such as not enough storage for the partition on the device. **Enter** is called as part of creation of a partition.

4.3.9 remove(partname) returns integer

This operation allows the caller to remove a partition from the device. **Partname** is the sysname for the partition. The entry for the partition is removed from the index table on the device and the storage for the partition is deallocated. The return value indicates success or an exceptional condition, such as a non-existent partition. **Remove** is called as part of the removal of a partition from the device.

4.3.10 partitions(partarray) returns integer

Partitions places the partition entries in the device's index table into the array parameter **partarray**. The major use of **partitions** is expected to be at system initialization, where it passes partition's sysnames to the boot code so that the partitions may be activated. The return value is either the number of partitions written into the **partarray** (a non-negative value) or a negative value indicating an exceptional condition, such as a bad index table.

5. The Partition Object

The partition object represents an intermediate level of abstraction of secondary storage. Partitions are consecutive blocks of secondary storage that reside completely on one device. Each partition is a logical object in that it manages the allocation of its own storage and maintains the structures used to do so. A Clouds partition does not enforce any logical organization of the data which resides on the partition, at least not in the sense of a Unix⁴ partition. A Unix partition represents a separate file system and all the files on the partition have a hierarchical relationship. The objects residing in a Clouds partition may bear no relationship to each other. It is simply an administrative organization imposed by the partition system indicating how storage in a particular partition is managed.

The two important types of partitions are recoverable and non-recoverable. When a partition is made non-recoverable, it is a declaration that no recovery will be provided for object data stored on that partition and that recoverable objects should not be placed in it. There is no similar restriction for recoverable partitions; such partitions may contain a mix of recoverable and non-recoverable objects. Other partition types include those used for paging surfaces and those reserved for temporary items.

Partitions manage storage as device independent blocks of storage and these are the smallest units of allocation that partitions support. The blocks are addressed by a partition block number (pbn) which is an offset from the beginning of the partition. All partitions are a multiple of this block size.⁵

The partition has as its initial block a header containing partition specifications. The header repeats most of the information found in the medium index table entry for this partition, plus information about the partition's state. This structure is generally accessed only when the partition is activated or some change is made to the partition; at other times the information is in memory and is referenced there.

Another structure used by the partition object is the system partition table. Like the system device table, the SPT is not part of any one partition object instance, but is part of the underlying mechanism. The table contains entries for all partitions which reside on the local machine. Each entry in the table associates a partition sysname with the data structures and information for that partition. These structures and information include the starting block number for the partition, pointers to in-memory structures and buffers used by the partition object, and a pointer to the device object on which the partition resides. This last pointer is actually a pointer into the system device table. Figure 4 shows the complete relationship amongst these structures.

Another function of the partition object is to maintain the location of segments and make available this information upon request. One of the features supported by the Clouds kernel is the location independence of objects (and thus segments). We mean by this that an object may reside on any partition on any node in the Clouds system and may be moved to any other partition on any other node. This implies that each access to an object requires that a (potentially) system-wide search be initiated. The sysnames given to objects give no (definite) information as to the location of the objects. As can be imagined, such searches can be time-consuming. In particular, searches on the partitions at a node might require one or more disk access each. We discuss one method of lessening the impact of these searches shortly.

4. Unix is a trademark of AT&T Bell Laboratories

5. The preliminary implementation will undoubtedly make this size equal to the size of a main memory page frame.

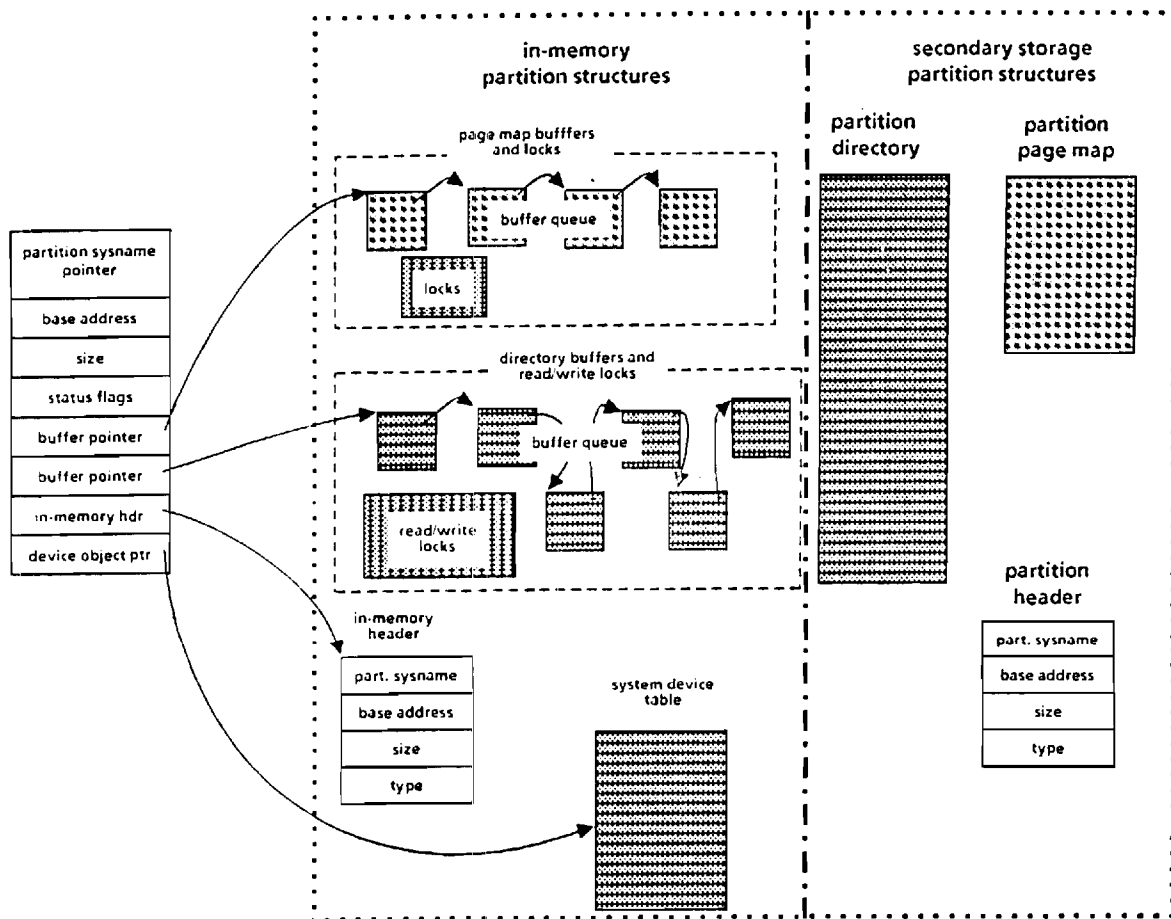


Figure 4. The system partition table and other partition object structures

5.1 Partition Data Structures

Two of the major functions provided by a partition object are the location of objects and the management of storage. To provide these functions the partition object maintains two structures: the partition directory and the partition page map. The partition directory is a large hash table which is composed of page-sized buckets. In our current implementation the bucket size is 512 bytes, allowing each bucket to hold sixteen entries, each entry consisting of sysname-pbn pairs. The sysname is the id for a segment and the pbn is the offset of the segment within the partition. The entries to the directory are hashed to the proper bucket on the sysname and then to the proper entry in the bucket by a secondary hash function, also on the sysname.

The page map is simply a bit map representing the storage allocation for the partition. This structure, along with the directory, contain most of the information that composes the partition state. As such, they are crucial to maintaining the reliability of the partition and the system as a whole, and some care must be taken in the modification and access of the partition directory and page map, as explained in section 7. Additionally, the storage manager must protect these structures from device failures. The basis for this protection is redundancy of the information. The partition directory and page map have duplicates at known locations in the partition. We are not overly concerned with the extra storage required; we calculate that even with duplicate structures we can keep the storage requirements for these two structures below one per cent of

the total storage. Combined with the protocols we follow for maintaining the reliability of segments and partitions, we should be able to minimize the access overhead caused by this redundancy.

The partition directory and page map may be too large to completely reside in memory and, in fact, we will not have them mapped entirely into virtual memory. Instead, we will maintain buffer areas for the two structures, bringing in new pages from secondary storage as needed, and using a least-recently-used discipline for replacement. We suspect that locality for the page map will be fairly good so that allocations of storage can be done from the memory buffers. However, we suspect that accesses to the partition directory will typically take one access to secondary storage. If our hashing functions are chosen properly we may be able to handle directory requests in (at most) one secondary storage access.

The partition object maintains another structure which it uses to avoid unnecessary secondary storage accesses altogether (or at least make such accesses rare). The structure in question is a Bloom filter ^[10] which we have called the *Maybe Table*. The Maybe Table is a probabilistic membership checker. It will indicate either that the object in question definitely does not reside on the partition being checked, or indicate that it possibly does. Thus, the Maybe Table gives a method of short-circuiting secondary storage accesses in cases where it gives a negative response. However, a positive response may still lead to unnecessary accesses to secondary storage. The key to success is to reduce the ratio of non-resident positive responses to all positive responses to as small a value as possible.⁶

As described in ^[10], a probabilistic membership checker is a hash table where collisions are allowed. There are two techniques described in that paper that present methods that could be used with Clouds object sysnames. In the first technique, the Maybe Table consists of a table of transformed entries. The transformation is a hashing function which takes a 48 bit sysname and produces a shorter Maybe Table entry. Several sysnames may hash to the same entry value. This entry value is then placed in the Maybe Table by the use of another hashing function; this time collisions are handled in a conventional manner. To query the Maybe Table, the sysname is once more transformed with the first hashing function, and the proper entry located using the second. If the retrieved entry matches the transformed sysname, a positive response is returned. Otherwise, the collision handling mechanism is invoked and another entry is tested. If a positive response has not been returned upon termination of this procedure, a negative response is returned.

A second scheme is to treat the Maybe Table as a bit-string and use t different hashing functions, each of which returns an index into the bit-string. Placing a new entry in the Maybe Table requires setting the bit whose index is returned by each hashing function. The test for membership requires that all bits whose indices are returned by the hashing functions be set; any clear bit causes the return of a negative response. Figure 5 illustrates the use of these two techniques. In the example, the Maybe Tables are 18 bits in length. In each case, sysnames are represented by three bits in the Maybe Tables. In the first case, sysnames are represented straight-forwardly by three bit entries; in the second case, three bits are set for every sysname belonging to the table.

The benefit drawn from the use of a Bloom filter such as the Maybe Table is that it is a more compact representation of the universe in which membership is being tested. In the case of the Clouds kernel, this is the sysname population of a partition. This allows more of the table to be kept in virtual memory (perhaps all of it), and so queries on the Maybe Table can generally be

6. This is an area that is open to further research. We believe that the goal is achievable by careful selection of the (possibly more than one) filters used, and their manner of implementation. We hope to do some measurements and research on this once the system is working.

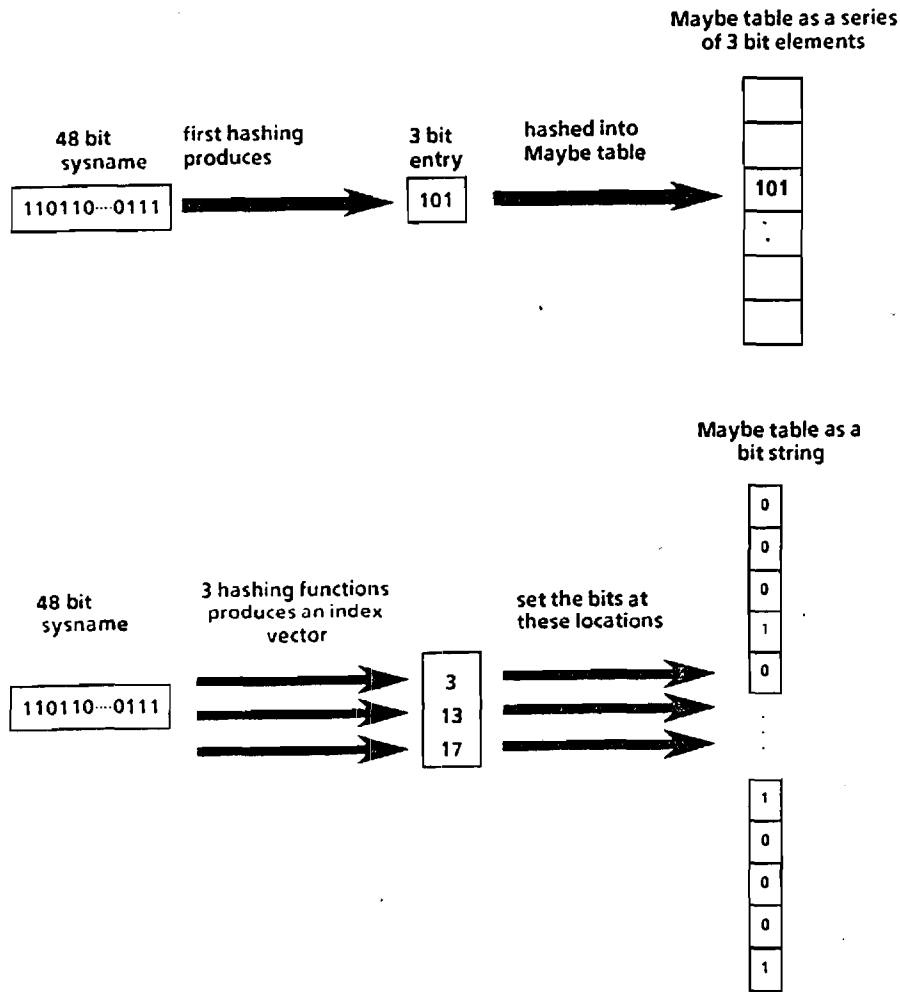


Figure 5. Two implementations of a Bloom filter

answered without going to secondary storage. If the response is negative, an unnecessary access to secondary storage is avoided, speeding the search for the proper segment. If the response from the query is positive, then an access to secondary storage is required, to either locate the segment or to ascertain that it is really not on this partition.

Maintaining the Maybe Table has several costs that must be considered. One, of course, is the initial creation cost. The storage manager will perform this initialization at system start-up for each partition and thus the time spent can be ignored. Another cost arises from the dynamic nature of the Clouds system. Objects are created on a partition, deleted from the partition, and moved to other partitions. Clearly, these changes must be reflected in the Maybe Table else the performance will be degraded. Creation of objects and the movement of objects onto a partition pose no problem: the sysname can simply be incorporated into the table via the methods described above. However, deletions of objects and movement of objects from a partition are more troublesome. An entry or set of bits in the Maybe Table cannot be cleared to remove a sysname's presence from the Maybe Table because several sysnames may be represented by the same entry or set of bits.

The simplest solution is to simply reconstruct the Maybe Table at intervals during the lifetime of the system. This reconstruction may be done asynchronously as a background task. The question of when the Maybe Table should be rebuilt is not yet answered. It would seem best to base the interval between reconstructions on activity of the partition, particularly the rate of deletions. This could be done indirectly by recording the performance of the Maybe Table and reconstructing the table when the performance falls below a given threshold. Or the monitoring could be more direct, measuring the number of deletions and movements of objects from the partition. Both of these methods have advantages and disadvantages. The indirect method for example, seems to be desirable since it measures the attribute that we want to optimize (avoiding disk accesses). However, a burst of queries for a sysname not resident on this partition but which happens to hash to the same entry or set of bits could cause a severe drop in performance even though the table as a whole is behaving reasonably well.

We are currently incorporating a Maybe Table into the partition object as described in [10]. We wish to get the maximum performance from the Maybe Table with the minimum impact on virtual memory. Therefore, we may consider other implementations for the Maybe Table, depending on the performance obtained. It may be, for example, that we are able to take advantage of the nature of the sysname population to improve the performance of the table.

5.2 Calls on the Partition Object

The storage management system uses the following calls to manipulate the partition data. Most of the calls require at least one sysname as an input parameter, usually a sysname for the partition (the exception being `create_partition`; see below). Occasionally, sysnames for segments and devices may also be required.

5.2.1 *P_create(devname, size, partatt) returns partname*

`P_create` reserves a sequence of records on a device to form the partition. **Size** is the size of the partition in bytes (this parameter is rounded by the call to the record size of the device) and **devname** is the sysname of the device on which the partition is to reside. A sysname for the new partition is generated and returned as the value of the call. The record location of the initial record of the new partition is stored, along with the size (in device records) and the partition sysname, in the media index table. The attributes of the partition, specified in the input parameter **partatt** are also stored in this new partition entry. `P_create` makes use of the enter call on the device object to perform its task. In particular, `P_create` must be able to request allocation of storage from the device.

5.2.2 *P_destroy(devname, partname) returns integer*

This call takes the two sysnames given as input parameters and frees the chunk of storage used by the named partition. **partname** specifies the particular partition to be destroyed and **devname** specifies the device on which it resides. The integer return value indicates the status of the partition after the call (destroyed or not found on this device). The call removes the partition's entry in the media index table and releases the storage used by the partition. The device manipulations are performed with the device object call `remove`. `P_destroy` also makes calls on the device object to perform its task.

5.2.3 *P_enter(partname, segname, pbn) returns integer*

`P_enter` places an entry in the partition directory for a segment. **Segname** and **partname** identify the segment and partition, respectively. The entry in the directory includes the segment sysname and the partition block number, **pbn**. The call also modifies the Maybe Table. The return value indicates success or an exceptional condition.

5.2.4 *P_remove(partname, segname) returns integer*

This call removes the entry for a segment from the partition directory. **Segname** and **partname** identify the segment and partition, respectively.

5.2.5 P_return(partname, segname, seginfo) returns integer

P_return returns the segment header indicated by **segname** which resides on the partition specified by the input parameter **partname**. The header includes the sysname for the segment, the size of the segment (in partition records), the record address of the segment header, and whether the segment is recoverable. The segment header is placed in the parameter **seginfo**, which is a pointer to a block of storage reserved for the information. If the segment is present, the return value of the call is positive; otherwise the return value is negative. The call finds the information by searching the partition sysname map and examining the segment header found. The Maybe Table is first queried in an attempt to avoid unnecessary secondary storage accesses.

5.2.6 P_get_{first,next}(partname, number, segarray) returns integer

These two calls are similar to **P_return**, in that they return the attributes of a segment found on the partition specified by the input parameter **partname**. The segment is unspecified, however. **P_get_first** places the first **number** of segment sysnames appearing in the partition directory in the parameter **segarray**. **P_get_next** can then be used to retrieve the attributes of the **number** subsequent segments. The two calls share a static variable which holds the index of the next segment about which information will be returned by **P_get_next**. The variable is reset to zero after the last entry in the partition directory is accessed and is initially set to zero, which is an array large enough to hold the requested number of sysnames. The return value is either zero, indicating no sysnames could be found, or the number of sysnames actually returned by the call.

5.2.7 P_available_space(partname) returns integer

This call simply returns the number of free records on the partition indicated by **partname**. A negative value may be returned in exception conditions. The call does a bit count on the volatile record map. Because the volatile free map contains allocations and deallocations for uncommitted actions and because no synchronization is done on the record map, the value returned should be considered only an approximation of the "true" number of free records.

5.2.8 P_{read,write}(partname, part_offset, address) returns integer

P_read causes the transfer of the contents of a partition record, **part_offset** from the partition specified by **partname** to the physical page in memory indicated by **address**. **P_write** reverses the procedure, transferring the contents from the physical memory page to the partition record. The calls use their return values to signal exceptional conditions. The virtual memory system uses this call to handle page faults.

5.2.9 P_getblk(partname) returns pbn

P_getblk simply returns the partition block number of a free page on the partition. The volatile page map is updated to reflect the allocation. A negative value is returned if there is no partition storage remaining.

5.2.10 P_returnblk(partname) returns integer

This call deallocates the page at the partition block number passed through **pbn**. The volatile page map is updated. A negative value indicates a bad partition block number.

5.2.11 P_restore(partname, pbn) returns integer

The **P_restore** operation is called on system startup to examine the partition. If necessary, the operation will perform any repairs to the partition structures required to bring it back into a consistent state. The call will also cleanup any unfinished action processing. This sort of repair is done on a partition-by-partition basis, since not all partitions have the same attributes and therefore will not require the same processing. In particular, cleanup of action processing is not necessary on partitions not supporting recovery and partitions being used as paging surfaces. **P_restore** must determine attributes of the partition by examining the partition header and then proceed accordingly. The details of **P_restore**'s operation are described in section 7, which is concerned with the reliability of the storage manager. **P_restore** also initializes structures used by the partition object, such as the Maybe Table.

6. The Segment Object

The segment object provides the final level of abstraction for secondary storage. With these objects, we are operating on blocks of storage allocated by the partitions. The abstraction provided by the segment object is that of a sequence of bytes (kernel segment type). The implementation is actually a tree of fixed length blocks of storage, as we shall see.

Segment objects provide a standard abstraction for the kernel to manipulate and process all Clouds objects. The object implementation provides mechanisms for mapping segment data in and out of virtual memory, creating and destroying segments, and modifying segments. The necessary algorithms for maintaining the reliability of the segment data exist at this level.

The segment object is unconcerned with the internal organization of the objects it is managing. The storage management system treats segments as uninterpreted bytes. Any interpretation is performed by other parts of the kernel, such as the object manager.

6.1 Segment Object Data Structures

Recall that a partition directory has a set of entries which contains the pbn for the segments residing on the partition. The partition block addressed by one of these entries contains a segment header that identifies the segment. The complete header is 512 bytes long and contains the segment (object) sysname, the object type sysname, a segment status field, a segment shadow pointer (the status field and pointer are used for recovery), and the size of the segment in bytes. The remainder of the header contains an array of pointers which lead to the segment data. These pointers address one of two sorts of blocks: index blocks, which are arrays of pointers to other blocks, and data blocks, which actually contain segment data. If, however, the storage required for segment data is less than that used for the array of pointers in the segment header, the segment data can be placed in the segment header itself. This would provide for the efficient processing of very small segments. Figure 6 shows the segment structure.

A segment is a tree whose depth depends on the amount of data in the segment. Hence, the smallest segment may have a depth of two (the header and the data blocks addressed by the header), but trees of arbitrary depth are supported. This also means that occasionally the segment will be restructured when its size is increased.

The interaction of the segment system and virtual memory is still being designed. It should be pointed out that much of the manipulations performed by the segment object will involve the segment's representation in virtual memory and the structures maintained by virtual memory itself. The segment system also makes some assumptions. One of these is that the location of the segment is known. That is, the action or process using the segment knows the partition on which the segment resides. Particularly, most segment calls do not require a partition sysname as a parameter.

6.2 Calls on the Segment Object

The following calls all require the sysname for the segment being manipulated. Any offsets are data record offsets, using the logical view of the segment.

6.2.1 *S_create(partname, segname, attr)* returns integer

S_create allocates storage for a segment and sets up the segment header and index records. The input parameters are the two sysnames for the partition and segment to be created,⁷ and a structure holding information about the segment (its size, object type, recoverability). The storage for the segment can be allocated and structured on the basis of the size field of *attr*.

7. Note that this call does not return a new sysname for the segment. If that were the case, it would not be possible to move existing segments into a partition and still reference them by their old names.

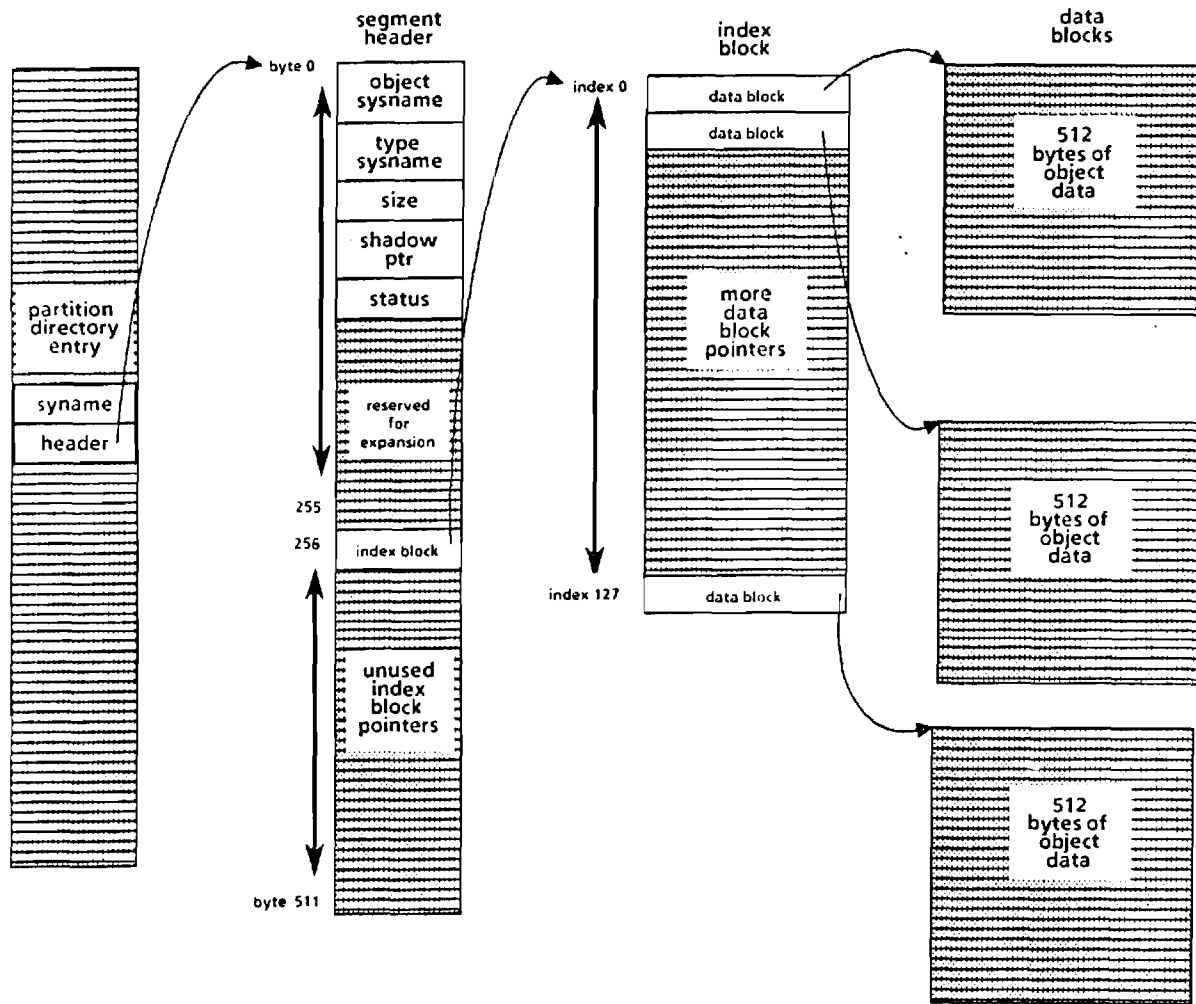


Figure 6. Clouds kernel segment structure

Data records are written in subsequent requests. The return value indicates the call status.

6.2.2 *S_destroy(partname, segname)* returns integer

This call deallocates storage for a segment. The sysname for the segment, **segname**, is removed from the partition directory.

6.2.3 *S_read(segname, offset, size, addr)* returns integer

The **S_read** call causes the transfer of **size** number of pages from storage to memory. **Segname** identifies both the memory and storage versions of the segment. The source of the pages is at location **offset** of the segment named by **segname**. **Addr** is the virtual memory address of the transfer destination. The return value indicates the status of the call.

6.2.4 *S_write(segname, offset, size, addr)* returns integer

S_write transfers data from memory to storage. **Addr** is the source of the transfer, in this case a virtual memory address. **Segname** is the sysname for the object (segment) whose data is to be transferred. Note that this identifies both the memory pages (source) and the secondary storage pages (destination) that must be transferred. **Size** number of pages, beginning at offset **offset** of the segment, are copied from virtual memory to the storage segment. The return value indicates the status of the call.

6.2.5 S_precommit(aid, touchlist) returns integer

S_precommit performs the segment level precommit protocol as described in section 5. **Touchlist** is a list of the objects which have been modified by the action. **Aid** is the sysname of the action making the precommit call. The call return value indicates the success or failure of the call.

6.2.6 S_eoa(segname, flag) returns integer

This operation performs the segment level commit or abort protocol as described in section 5, depending on the value of **flag**. The return value indicates the success or failure of the operation.

6.2.7 S_chgsize(segname, delta) returns integer

The call allocates or deallocates storage from the end of a segment. **Delta** is the number of records to allocate or deallocate (positive or negative value, respectively). The return value is the status of the call.

6.2.8 S_status(segname) returns integer

This call determines the state of a secondary storage segment by examining the status field of the segment header. The return value is this status (permanent, shadowed, precommitted).

7. Reliable Storage Management

In this section we look at the techniques used to ensure the reliability of the storage manager in the presence of machine failures and action aborts. All the techniques described below require the information and features provided by the use of atomic actions. This information includes the knowledge of when it is correct to make the effects of an operation permanent and what data has been modified. The storage manager provides a set of protocols that use this information to make the correct updates to secondary storage so as to leave the storage system in a consistent state. In order to understand these techniques and the motivation behind them, we need to understand how the Clouds kernel manages actions.

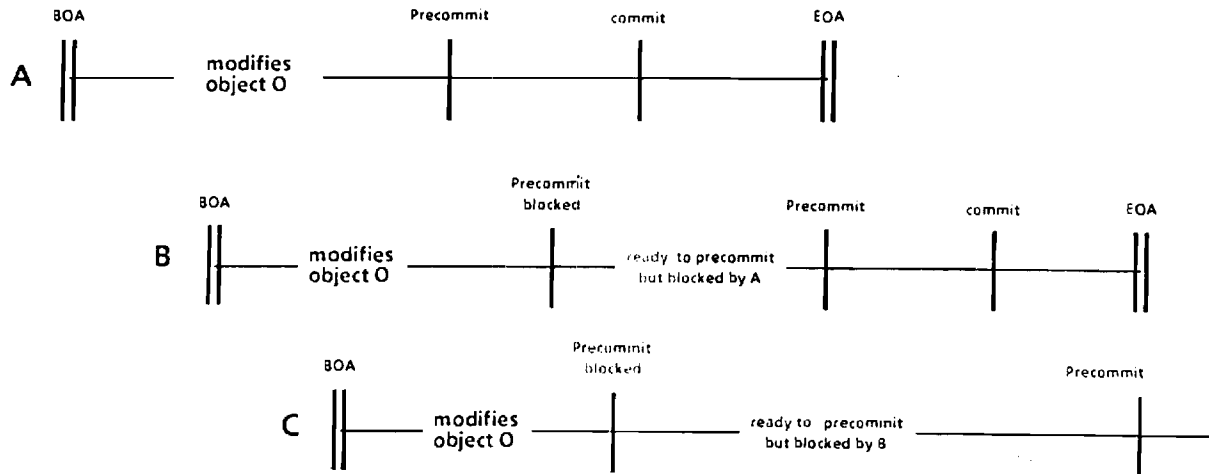


Figure 7. Actions block on competing commits

The Clouds system considers actions to be units of work. Many actions may be active in the same object, with each action updating object data. The only restriction enforced by the kernel⁸ on the synchronization of actions which are operating concurrently on a single object is at action precommit. An action that precommits in an object blocks all other actions from precommitting in that object until the precommitting action is committed. Other actions still update and process the object's data; the only restriction is on the precommit procedure. Although this restriction may seem to create potential bottlenecks, the simplifications it provides in the processing of commits will keep the blocking intervals short enough so as to cause no problems. In particular, this restriction means that the storage manager must provide reliable updates for only one action per object per time period.

There are two levels at which the storage manager must supply this sort of reliability: at the partition level, and at the segment level. The partition has critical data which must be updated correctly to allow the storage manager to function correctly. As stated previously, this data includes the partition directory and the partition page map. At the segment level the storage manager is responsible for the consistent update of object data and the underlying structures that represent this data. We use two rather distinct approaches to providing the recovery for these two levels. In both cases the techniques provide pessimistic recovery; no changes are actually made to the "live" data until the responsible action commits.

8. The programmer may define other forms of synchronization within the implementation of the object based upon semantic knowledge and other design factors. The kernel does not preclude such choices.

7.1 Segment level recovery

Segment recovery is accomplished via a shadowing scheme^[11]. That is, segments on which actions are operating will have shadow versions which the actions will actually see. We note that one of the goals of the recovery scheme is, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few storage accesses as possible. Shadowing, then, will be minimal, with only those parts of the segment actually modified being shadowed.

The shadowing scheme consists of a set of protocols that indicate what the storage manager must do for specified segment states and action events. We consider these states and events in the following paragraphs and develop the protocols that shadow segments. When an action is started, the storage manager is involved initially in the transfer of the data for the object being operated upon from storage to memory. Until precommit occurs, the only transfer of information is from device to system. All modifications to the action data are handled in memory by the action manager. On the action commit the storage manager starts transferring information back to storage. These transfers are the result of the action management system protocols for transferring action updates to the permanent state of the object.

7.1.1 The precommit protocol

The precommit protocol ensures that updated pages of object data that an action has modified are recorded on non-volatile storage to prepare for the final commit of the action. The storage manager performs the shadowing and data transfers as follows:

- P1 The storage manager determines how many pages are to be shadowed and allocates storage for shadow versions through calls to the virtual memory system and the partition object, respectively. The storage manager allocates shadow storage not only for modified data pages, but also for the segment header, plus any index pages that are required to reach a modified data page.
- P2 The storage manager shadows the segment. The segment header is copied to the shadow segment header. The modified data pages are copied from memory to the shadow data pages. Modified versions of index pages are copied to shadow index pages. Some index pages must be modified and shadowed so that the shadows point to the shadow versions of data pages. The storage manager places a modified version of the segment header into the shadow segment header. Modifications made to the segment header data could include a change in the size, and changes to the array of pointers (some of these pointers may point to shadow pages, as with the index pages).
- P3 The permanent segment header is modified so that the status flag indicates that the segment is being shadowed. A pointer is also set in the header which indicates the location of the shadow segment header.

One point to note about the above protocol is that there are a number of reads assumed to get the segment structure into memory. Also note that the number of pages that must be shadowed and the identification of which index pages must be shadowed can be determined by knowing the size of the segment and which data pages must be shadowed. The segment header is modified last to reduce the work necessary to restore the segment in the event the system crashes before the precommit is completed.⁹

Once the precommit completes, we are left with two versions of the segment. The two versions overlap in spots as illustrated in Figure 8, where blocks within the dashed box are part of the

9. A crash at any point before this final write will recover with the shadow pages still listed in the free space list and completely unreferenced, and thus they get scavenged automatically.

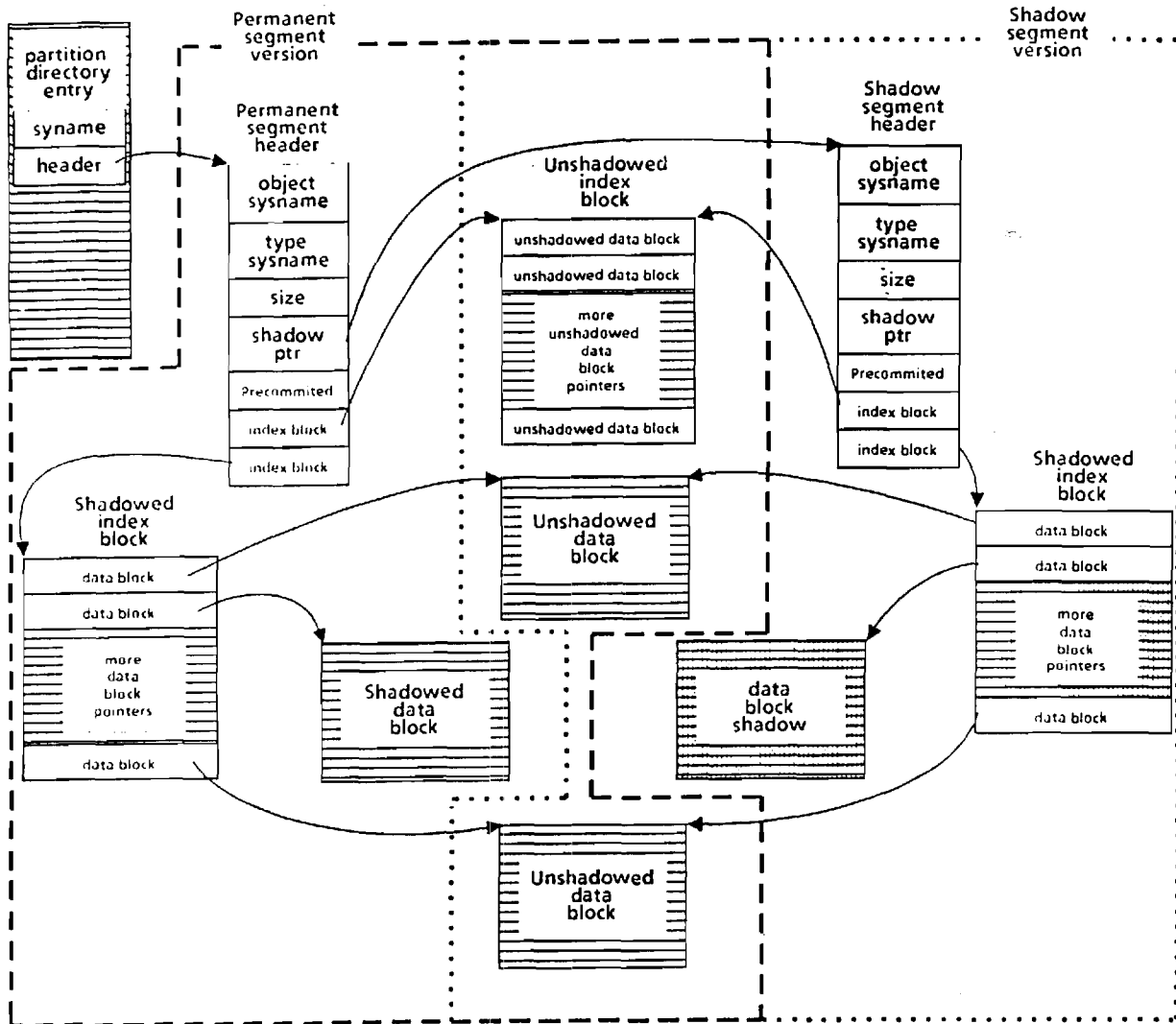


Figure 8. Precommitted segment

permanent version, while blocks inside the dotted box are part of the segment shadow. Read operations on unshadowed pages refer to permanent pages. The shadow version is visible only to the action which is performing the commit.

We must point out that the storage manager's precommit protocol is not the same as the action manager's precommit. After the storage manager has completed the shadowing, the action could still abort and the shadowed version would have to be removed. An example of such a situation is when the action spans several nodes and uses a two-phase commit protocol. Phase one is complete only when all nodes have completely shadowed any object data the action touched on their storage. If one node cannot do this, the action aborts.

7.1.2 The commit protocol

Once the segment is shadowed and the action decides that it can continue the commit, the storage manager performs its own commit protocol. The storage manager must switch the shadow version for the old permanent version of the segment. There is some bookkeeping for the partition as well. The protocol is as follows:

- C1 Update the permanent page map on storage. This requires that all addresses for shadow records be allocated in the page map and all modified records of the segment including the segment header be deallocated in the page map.
- C2 The partition directory is set so that it points to the new segment header for the segment.
- C3 The shadow segment header is set so that it is now the permanent segment header, that is, it is marked as "permanent."

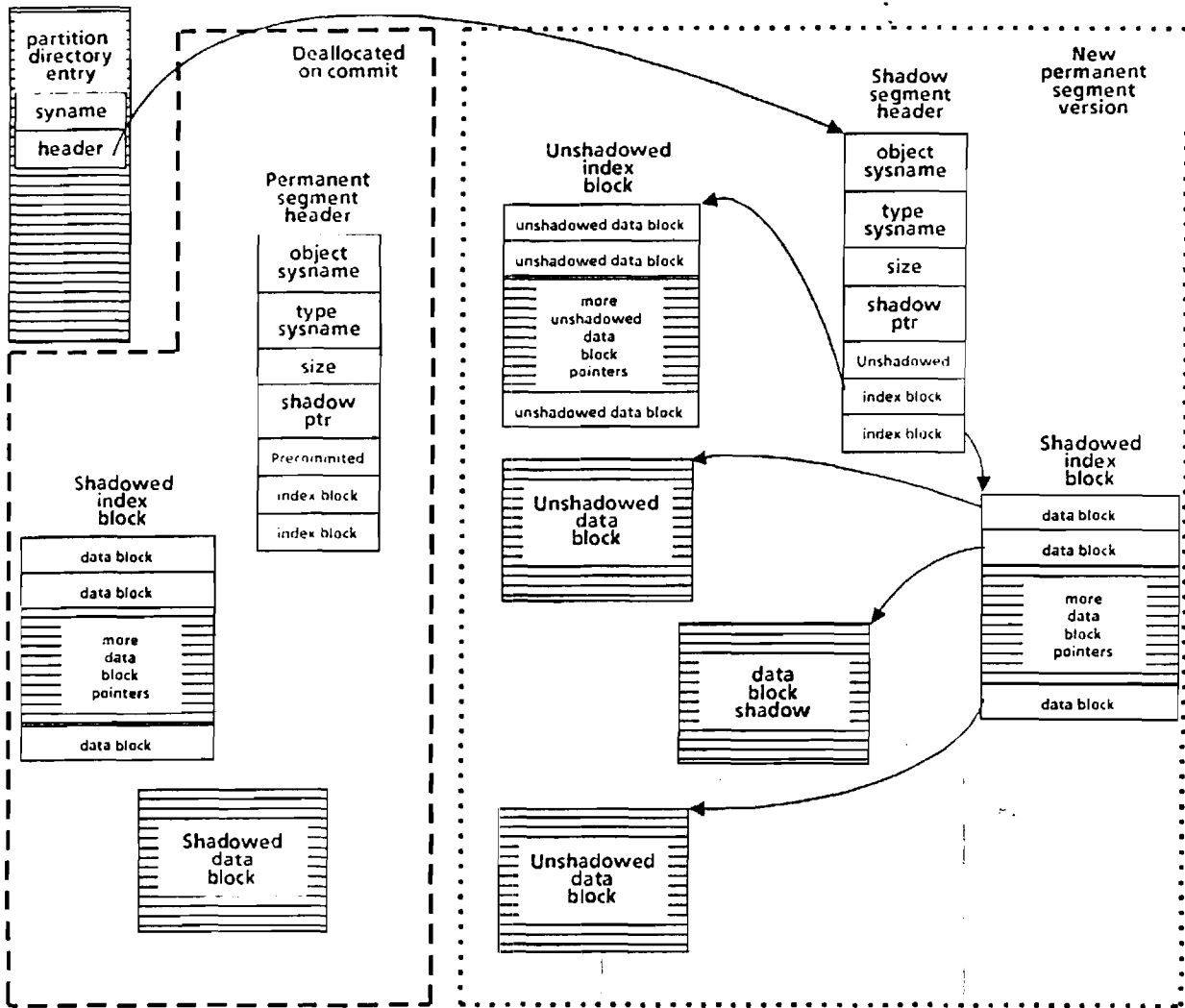


Figure 9. A committed segment

Once this protocol is complete, any references to the segment will refer to the new version of the segment. The new segment is a merging of old unmodified records and new records. Figure 9 shows a committed segment. The blocks in the dashed box were parts of the permanent segment being shadowed during precommit. These blocks are deallocated as part of the commit during step C1. During this phase of the protocol, the storage manager updates the permanent page map on secondary storage. Recall that Clouds uses pessimistic recovery and any effects of an action, including storage allocation to perform the commit, cannot become permanent until the action commits. Therefore, all allocations are performed on a volatile page map. We discuss this and other ideas in the section on partition level recovery.

7.1.3 The abort protocol

Actions can also abort for one reason or another and the storage manager requires a protocol for this event as well. The protocol simply rids the segment of any trace of the action's work as follows:

- A1 The volatile page map is updated to remove allocations that the action has made to shadow the modified pages of the segment.
- A2 The status flag of the permanent segment header is set to show that the segment is unshadowed and then the shadow pointer is set to null.

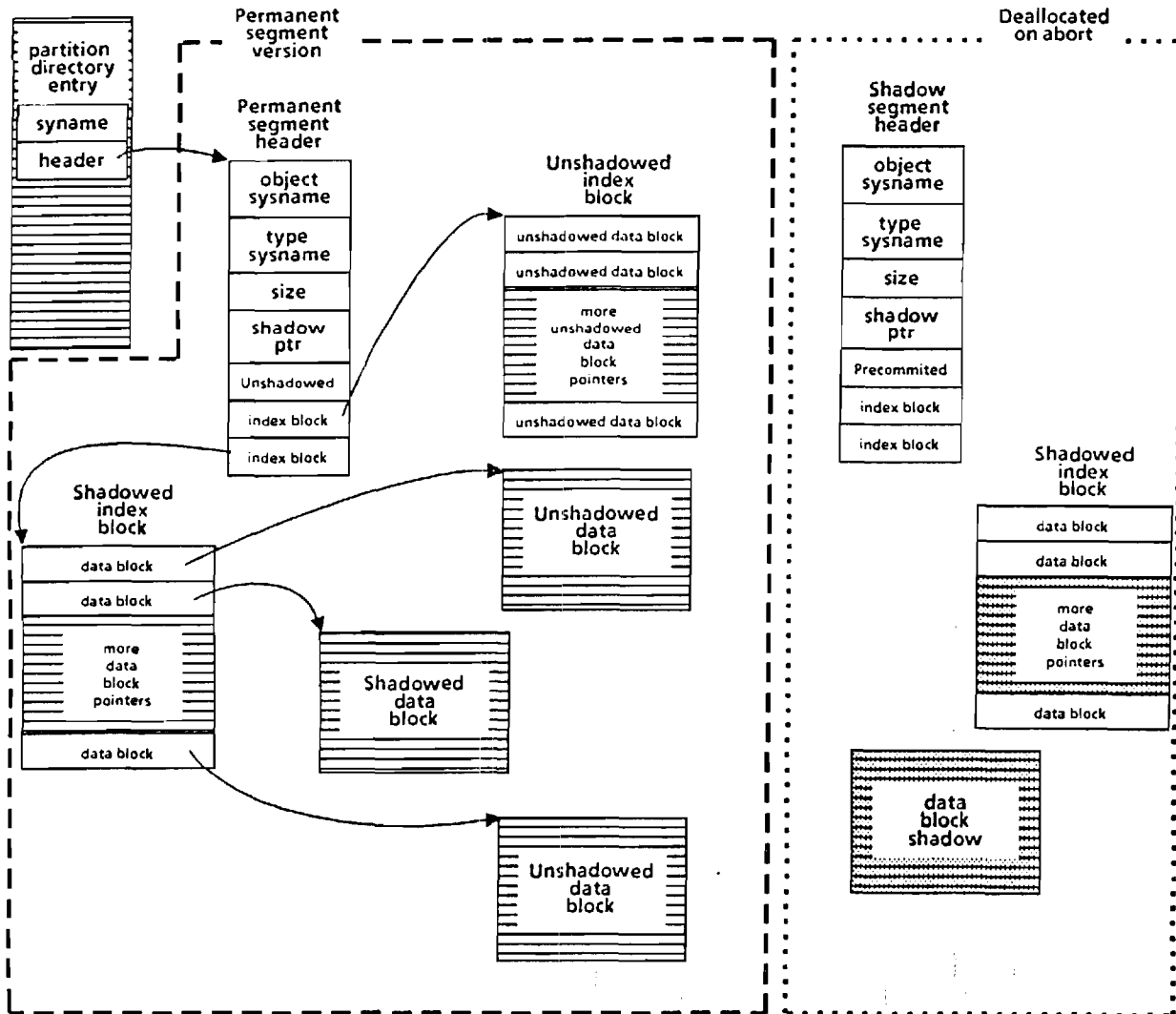


Figure 10. An aborted segment

The storage manager uses this protocol only when an action has started to commit and aborts in the middle. If the action aborts before attempting to commit, the storage manager is not involved at all. Figure 10 illustrates the results of the abort protocol. In this case, the blocks inside the dotted line are deallocated upon the abort, as these blocks are only shadows for the permanent segment.

7.1.4 System failures

One final event must be considered. That is how does the system recover from a machine crash? Specifically, we are concerned with restoring the segment and partition to a consistent state after the system is brought up again. The system may have had a number of actions in various states at the time of the crash and we want to insure the appearance of indivisibility of actions. Under the Clouds policy, any action that has not precommitted when a crash occurs is aborted when the system is restored. As we have already noted, actions which do not begin precommit before the system crashes do not concern the storage manager; these actions have no effect on system storage. For objects which completed precommit processing, we must determine whether their action's effects become permanent or are erased. This depends on the state of the action. The crash recovery protocol, then, is as follows:

CR1 A new volatile page map is created for the partition.

CR2 The storage manager determines which actions touched segments on this partition and determines the state of each such action. The storage manager polls a kernel database and examines the segments on its local storage to identify these segments.

CR3 If a segment was touched by an action that has completed phase one and should be committed, the storage manager performs the commit protocol on the segment, as above.

CR4 If the action which modified this segment was aborted by the action manager, the storage manager uses the abort protocol, as given above.

At the end of crash recovery, the partitions are in a consistent state; either the actions occurred or they did not. The database referred to in step CR2 is a kernel level database shared by the nodes in the system. The database exchanges information amongst the systems using a suite of algorithms developed in ^[1]. The information in the database represents an approximate state of the network. This database is copied from other nodes by the kernel when a node is added to the Clouds system. Among the information kept in the database is a list of actions, their status, and segments touched by the actions. Generally, the storage manager can find here the information needed for crash recovery. In some cases, though, a local action (one which does not leave the site on which it is born) may not appear in this list, even though its status at the crash time was complete and known. In cases such as these, the storage manager can find shadowed segments only by an exhaustive examination of the partitions.

Another issue is that of a system failure during an action write, so that only part of the write is actually completed. In the discussion thus far, we are assuming that we have atomic single record writes. The atomicity we are concerned with is failure atomicity, whereby the write either takes place or not. In practice, this means that we can detect an incomplete write (the system failed during a record write) and we are not overwriting the only copy of the data in question. If a device we are using does not support detection of incomplete writes, we can simulate the effect using the standard method of stable storage as described by Lampson and Sturgis in ^[12]. In ^[13] the question of when the atomic single record write assumption can be relaxed, if at all, and under what circumstances, is investigated.

7.2 Partition level recovery

In the last section we outlined the techniques used to provide reliability for the segments on storage. We now turn to the problem of maintaining the consistency of partition structures, particularly the page map and the segment directory. These structures were discussed to a small extent in the last section because they are involved in shadowing segments. We did not discuss how the structures themselves must be modified to maintain their consistency. Once again, let us consider the action environment provided by the kernel. Recall that a committing action blocks all other actions from committing in a segment it has modified. The partitions are objects, so that any action committing would block all other actions from committing in any object residing in that partition. For a one partition node, this permits only one action at a time to commit. We feel that this is too restrictive.

We allow any number of actions in a partition to commit simultaneously, excluding any segment conflicts. Given this, we do not feel that shadowing can be used to provide recoverability of the page map and directories. Maintaining the various shadow versions in itself would be complicated, but in addition we would need to propagate committed data to as yet uncommitted shadowed data. We therefore reject our segment level shadowing scheme as an approach for partition level recovery and we must develop another method for this task.

The partition directory does not have a volatile component. There are two copies of the directory residing on the partition (for the redundancy necessary to protect against media failures) and a committing action on a partition object must update both copies in a consistent manner to indicate that the new object version is to be used. Once again, we assume atomic single record writes, which will allow us to determine whether the copies are consistent, when the writes are performed in a determined order. An examination of both permanent copies and the header of the segment involved, if done in the proper order, will reveal any inconsistencies and the manner in which they should be resolved.

The partition page map has a volatile component which the storage manager uses to make non-committed storage allocations and which disappears after a system crash. Note that the volatile page map provides correct storage allocation information excluding system failures. Now recall that the commit protocol for storage management entails three steps, the second of which involves installing the action's storage allocations onto the permanent page map. We have two approaches we feel will provide consistent updating of the permanent page map. The first approach simply does away with the permanent page map of the partition, and maintains only the volatile version. As noted earlier, this provides correct storage allocation until a system failure occurs and the page map is lost. Clearly, we must be able to recover the page map after the system is restarted, and the obvious solution is an examination of the partition. Equally clearly, this will require quite extensive processing upon system startups.

The second approach to maintaining the partition page maps involves the use of intention lists and does require a permanent copy of the page map. With this approach, the storage manager during step one of the segment commit protocol does not write directly to the permanent page map, but instead writes an intention list of storage allocations (deallocations) to disk. Because the volatile page map reflects the correct storage allocation for a partition, the actual updating of the permanent page map from the intention list can be performed as background processing by the storage manager. If the system crashes before some updates are performed, they can always be done as part of the system startup processing. The steps required by this protocol are shown below:

1. The creation of the intention list begins at precommit. When the shadow is allocated, the storage manager places these pages on the allocation intention list. The pages to be replaced by the shadows are placed on a deallocation intention list.
2. When the signal is given to start the final commit, these lists are written to a list of pending allocations maintained by the partition.
3. At some later time, these lists are merged into the page map as part of normal partition bookkeeping.

The only restriction is that the updates from the intention list must be performed in the order in which the allocations and deallocations were committed.

Our initial implementation of the storage manager will use the first mechanism. We have two reasons for doing this. First, we are concerned more with the cost of commit processing than we are with system startup processing simply because we feel that system failures will be infrequent and because action processing is our model of computation. This approach both simplifies the implementation and makes the commit process more efficient, since no extra disk writes are required to update a permanent page map.

Secondly, an extensive examination generally will be made of the partitions at system startup to clean up any unfinished action commits or aborts. The reconstruction of the page map is partially subsumed in this processing.

7.3 Device support for recovery

The above protocols have several implicit assumptions on which they rely to operate correctly, two of which concern the device object. We have already mentioned the assumption that devices can perform atomic single record writes. The other assumption concerns the transfer of data from system to storage. The protocols assume that upon completion of a call to any of the "write" operations the data intended for transfer to storage has, in fact, been transferred. Under conventional systems, this is not necessarily the case, since requests for writes to storage may be buffered. Data may or may not actually be transferred before the system crashes. If the data were not actually transferred, there is no way to recover the segment or partition when the system is restarted.

At the device level, then, the storage manager requires some way in which to ensure the timely completion of data transfers. We wish to accomplish this without adversely affecting the other processing on the system. Also, the action causing the writes to storage must be informed of the completion of the writes in order to continue its commit processing.

There is a great deal of latitude with the timing of when the action writes are forced to the device. One discipline is to have a synchronous write operation that immediately forces the device to schedule requests issued by the operation. By this we mean that any requests currently being processed are completed and then normal scheduling is pre-empted. Synchronous write requests are then carried out in order of receipt. Thus, action writes are forced to the device early in the sequence of action commit processing. The drawback is that requests for synchronous writes appear in bursts at precommit and commit. Any scheduling that the device does for efficiency of the device's operation is disrupted.

Another approach is to allow the device to schedule the requests subject to its own constraints and simply inform the storage manager when the requests are completed. This allows the devices to schedule requests efficiently, but can delay action commit processing. However, the storage manager does know when the completion of the precommit and commit protocols can be safely signalled.

A compromise approach initially allows precommit and commit to be enqueued as usual and handled as normal requests. It is only when completion of the commit or precommit is imminent that the write must be forced to storage. To accomplish this, requests must be identifiable by the storage manager so that the manager can signal which requests must have priority. The manager can simply place the action id of the committing action in a field of the request when requesting a write to storage.

When the storage manager determines it is necessary, it can make a call on the device object to reorder its queue of requests, giving priority to this action's requests. This technique may prove useful if a significant amount of time can elapse before the storage manager must complete the precommit and commit procedures. In cases where the action has touched a number of objects on several systems this may indeed be the case. In such situations, the devices can operate efficiently (and possibly reduce the number of pending precommit and commit requests, reducing the disruption when it becomes necessary to force them to storage), and the action is not delayed, since it is not ready to complete its commit. To accomplish this as stated, the storage manager must be able to identify when requests must be forced to storage. This will be based on the results of any two phase commit that is performed and the storage manager will rely on the action management system to signal when final commit is to be performed.

Each device object maintains a flush table (as discussed in section 4) to control the forcing of action writes. When the list of requests for the action entry in the flush table is empty, the storage manager can inform the action that the commit processing can continue.

7.4 Summary

Support for reliability and recovery is integrated throughout the storage manager from the lowest level to the highest. The segment system, via the use of segment objects, provides for recovery of client object data recovery through the use of shadowing of modified data and the discipline of the shadowing provided by the protocols discussed above. The data that the storage manager uses to manage Clouds objects is made recoverable by the partition objects. At this level, our primary concern is how to maintain the data across system failures, and we present a few approaches for doing this. At the device level, support is provided to ensure that data is written when necessary, allowing action processing to be performed correctly at a higher level.

8. Conclusions

The motivation behind the Clouds project is the belief that systems in general and distributed systems in particular should provide reliable data management and reliable computation. This report documents part of our efforts towards that goal, namely the storage manager for the Clouds kernel. The Clouds storage manager, in addition to providing the traditional services of storage management, also provides support for the object-action methodology presented by the Clouds kernel.

We have presented an overview of the storage manager for the Clouds kernel. The storage manager is presented as a collection of objects, each of which provides an abstract view of the secondary storage. At the lowest level, secondary storage is viewed through the device object, and the physical storage medium is viewed as a sequence of pages (in the current implementation, a page is 512 bytes) with very little structure, other than the device header and index table. One step higher in our hierarchy is the partition object, which manages a portion of the raw storage provided by the device object. Once again storage is viewed as a sequence of pages, but that storage has a more defined structure. Each partition maintains a directory and a page map, so that each partition is responsible for managing its storage and for providing a location service for the next level of abstraction, the segment object. The segment object provides a view of storage that is a sequence of bytes and each segment object generally corresponds to some other kernel or user object. The storage manager views segments as a tree-like structure of pages.

We have described the data structures associated with each object and presented the operations with which the data structures can be manipulated. We have also tried to convey the relationships amongst the three objects and to show how they interact with each other and the rest of the kernel.

The research that we are conducting is primarily involved with how the storage manager provides the recoverability of the storage it manages and thus supports the reliability of the Clouds kernel. To that end the storage manager uses a set of protocols to ensure that object data is updated in a consistent manner and that even through system failures, enough information survives to maintain the consistency of the object. We show how these protocols are used to support the action/object programming paradigm of the Clouds system.

Each level of storage object discussed provides some support for recoverability. The device objects maintain flush tables which allow the storage manager to ensure that action writes are completed before a commit is finalized. The partition object maintains a consistent view of allocated storage and insures the correct updating of the partition directory. The segment object provides recovery of object data through the set of protocols described.

REFERENCES

1. Allchin, Jim, *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1983.
2. McKendry, Martin, *Ordering Actions for Visibility*, Technical Report GIT-ICS-84/05, Georgia Institute of Technology, Atlanta, Georgia, 1984.
3. Allchin, Jim and Martin McKendry, *Object-Based Synchronization and Recovery*, Technical Report GIT-ICS-82/15, Georgia Institute of Technology, Atlanta, Georgia, 1982.
4. Allchin, Jim and Martin McKendry, "Synchronization and Recovery of Actions," *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, 1983.
5. Spafford, Eugene and Martin McKendry, *Kernel Structures for Clouds*, Technical Report GIT-ICS-84/09, Georgia Institute of Technology, Atlanta, Georgia, 1984.
6. Spafford, Eugene, *Kernel Structures for a Distributed Operating System*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.
7. Jones, A. K., "The Object Model: A Conceptual Tool for Structuring Software," *Operating Systems: An Advanced Course*, Springer-Verlag, New York, pp. 7-16, 1979.
8. Wilkes, C. Thomas, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, Georgia Institute of Technology, Atlanta, Georgia, 1985.
9. LeBlanc, Richard J. and C. Thomas Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing*, Denver, 1984.
10. Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, No. 13, Vol. 7, pp. 422-426, July 1970.
11. Gray, J. N., "Notes on Data Base Operating Systems," *Operating Systems: An Advanced Course*, Ed. by R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, 393-481, 1979.
12. Lampson, B. W. and H. E. Sturgis, *Crash Recovery in a Distributed Storage System*, unpublished paper, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California, 1979.
13. Pitts, David V., *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.



GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE • ATLANTA, GEORGIA 30332 • (404) 894-3152

October 22, 1985

To whom it may concern:

The attached paper, Preliminary Aeolus Reference Manual, by C. Thomas Wilkes which was partially funded under the contract #MDA 904-84-C-6035 is being submitted for your information. This paper will be published as a Technical Report in the School of Information and Computer Science at Georgia Institute of Technology.

Sincerely,

Richard J. HeBlanc
Associate Professor

RJL/kkh

Preliminary Aeolus Reference Manual

Technical Report

GIT-ICS-85/07

July 1985

Revised 22 October 1985

C. Thomas Wilkes

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Table of Contents

1. Introduction	1
2. Explanation of Notation	2
3. Tokens	3
3.1. Identifiers	3
3.2. Numbers	3
3.2.1. Ints	3
3.2.2. Floats	3
3.3. Litchars and Litstrings	4
3.4. Comments and Compiler Options	4
3.5. Reserved Words	4
3.6. Operators and Delimiters	5
3.7. Other Characters	5
4. Declarations and Scopes	5
4.1. Compilation Units and Their Scopes	6
4.2. Qualified Identifiers	6
5. Constant Declarations	7
6. Type Declarations	8
6.1. Type Identifiers	9
6.2. Anonymous Types	10
6.2.1. Enumerations	10
6.2.2. Index and Pointer Types	10
6.2.3. Structured Types	11
6.2.3.1. Arrays	11
6.2.3.1.1. Bitstrings	12

6.2.3.1.2. Strings	12
6.2.3.2. Records	12
6.2.3.3. Structures	13
6.2.3.4. Sets	14
6.2.4. Locks	15
7. Constraint Declarations	15
8. Variable Declarations	16
9. Expressions	16
9.1. Operands	16
9.1.1. Variables	16
9.1.2. Constructors	18
9.1.3. Allocators	19
9.2. Operators	19
9.2.1. Arithmetic Operators	19
9.2.2. Bitwise Operators	20
9.2.3. Address Operators	20
9.2.4. Logical Operators	21
9.2.5. Set Operators	21
9.2.6. Relational Operators	21
9.3. Type Compatibility	22
10. Statements	24
10.1. Simple Statements	24
10.1.1. Assignment Statements	24
10.1.2. Procedure Calls	25
10.1.3. Object Operation Calls	26

10.1.4. EXIT Statements	26
10.1.5. RETURN Statements	26
10.1.6. NULL Statements	26
10.2. Compound Statements	27
10.2.1. IF Statements	27
10.2.2. CASE Statements	27
10.2.3. LOOP Statements	28
10.2.4. USING Statements	30
10.2.5. REGION Statements	30
11. Procedures	31
11.1. Procedure Declarations	31
11.2. Procedure Invocations	33
12. Objects	33
12.1. Object Definition Parts	34
12.2. Object Implementation Parts	37
12.3. Object Operation Invocations	38
13. Actions	39
13.1. Action Events	39
13.2. Recoverable Areas	40
13.3. Permanent and Per-Action Variables	40
13.4. Action Invocations	41
14. Processes	42
15. References	44
Appendix A: Systems Programming Example	A-1
Appendix B: Example of Action Programming	B-1

Appendix C: LALR(1) Grammar for Aeolus	C-1
Appendix D: Definition of the Object <i>standard</i>	D-1
Appendix E: Definition of the Clouds Action Manager Interface	E-1
Appendix F: Dynamic Arrays	F-1
Appendix G: Examples of Permanent and Per-Action Variable Use	G-1

1. Introduction

The goal of the *Clouds* project at Georgia Tech [Allc82, Allc83a, Allc83b] is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, which will provide an environment for the construction of reliable applications. The *Aeolus*¹ programming language developed from the need for an implementation language for those portions of the *Clouds* system above the kernel level. *Aeolus* has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the *Clouds* notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the *Clouds* system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as *Clouds* [LeBl85, Wilk86].

Thus, the main interest of *Aeolus* lies not in the language itself, but in what may be done with the language. We have avoided providing high-level features for programming actions with the intention of evolving designs for such features out of our experience with programming in *Aeolus*. These features will then be incorporated into an applications language for the *Clouds* system.

Aeolus has its roots in a long line of structured programming languages, including *Simula*, *Pascal*, *Modula-2*, and *Ada*.² Thus, many of its features should be easy to understand for those familiar with one of these languages; in particular, familiarity with *Pascal* or *Modula-2* is assumed throughout this report, and features will often be explained in terms of the corresponding features in those languages.

The main structuring features of *Aeolus* (as of the *Clouds* system) are objects, actions, and processes. *Clouds* supports the *object* concept as a convenient structuring principle for facilitating recovery and synchronization; *Aeolus* also allows the programmer to use the object features of the language for the specification of abstract data types, without necessarily invoking the object and action management features of the *Clouds* system. Thus, *Aeolus* objects provide a separate compilation facility as well as access to the object support of *Clouds*; the separation of object specifications into *definition* and *implementation* parts (much as are *modules* in *Modula-2* or *packages* in *Ada*) provides a safe interface to separately-compiled objects, as well as facilitating the design of large systems consisting of many objects (possibly implemented by several people) or the use of predefined objects. *Aeolus pseudo-objects* provide a means of isolating system dependencies—such as input/output or low-level machine architecture—into object-like modules which provide operations facilitating machine-level programming.

Support of the *Clouds* notion of *actions* in *Aeolus* is fairly low-level. Essentially, means are provided for specifying that an operation (procedure) of an object may be invoked as an action, or that an operation invocation is to be executed as a (oplevel or nested) action. Also, the status of action executions may be checked by means of calls to a *Clouds* action manager.

The *process* concept in *Aeolus* corresponds roughly to the *program* construct of *Pascal* or *Modula-2*. That is, a process ties together the constituent parts (objects) of a programmed system, and the invocation of a process provides activity in the *Clouds* system.

Except for the access *Aeolus* provides to the action management facilities of *Clouds* (which control recovery in the system), nothing in the language is explicitly dependent on the *Clouds* system for its implementation. In the *Clouds* implementation of *Aeolus*, the details of synchronization and recovery of objects are hidden by the interface to the *Clouds* object and

¹*Aeolus* was the king of the winds in Greek mythology.

²*Ada* is a registered trademark of the U.S. Government—*Ada* Joint Program Office

action managers; thus, for example, it is transparent to the programmer (and to the language runtime support) whether an operation invocation involves a local or remote object. Therefore, an implementation of Aeolus—without its features for recovery handling—should be possible under any operating system; only the object management need be subsumed by the language runtime support, which should be trivial for a non-distributed system.

This report is not intended to be a tutorial on the Aeolus language; rather, it strives to be a concise definition of the syntax and semantics of Aeolus, and thus should serve as a reference for programmers and implementors.

2. Explanation of Notation

The *syntax* (grammar) of a language consists of rules for arranging sequences of *terminal symbols* (also called tokens) in the vocabulary of the language (keywords, numbers, names (identifiers), and certain other characters used as punctuation to make the language more readable) into *sentences* (or sentential forms) which have meaning in the language. A syntax rule often specifies that a sequence of terminal symbols be grouped into a *nonterminal symbol*, an entity in the language which often has an intuitive meaning, such as an *expression* or a *statement*.

To describe the syntax of Aeolus in this manual, we will use a notation known as the extended Backus-Naur form (EBNF). (A complete grammar for Aeolus in LALR(1) form is presented in Appendix C.) In this notation, the so-called *metasymbols* [and] are used to enclose an Aeolus sentential form which is optional; the metasymbols { and } are used to enclose an Aeolus sentential form which may be repeated any number of times (possibly zero times). Tokens are enclosed in double quotes (""); nonterminal symbols are enclosed in angle brackets (<>). The left-hand side of a syntax rule specifies the nonterminal which is being defined, while the right-hand side of the rule gives the sequence of terminal and nonterminal symbols which are valid for the nonterminal being defined; the two sides of the rule are separated by the metasymbol (meaning "expands into").

Thus, for example, the syntax rule

$$\langle \text{identifier list} \rangle \rightarrow \langle \text{identifier} \rangle \{ \text{,} \} \langle \text{identifier} \rangle \}$$

specifies that the nonterminal *identifier list* consists of either a single *identifier* nonterminal, or a sequence of two or more identifiers separated by the comma token (","). The following are valid identifier lists:

```
foo
foo, bar
foo, bar, baz
```

Also, the rule

$$\langle \text{variable declaration} \rangle \rightarrow \langle \text{identifier list} \rangle \text{:} \langle \text{type} \rangle [\text{:} \text{=} \langle \text{expression} \rangle]$$

indicates that a *variable declaration* consists of an identifier list followed by the colon token (":"), a specification of the *type* of the variable(s), and an optional *initialization* of the variable(s) consisting of an assignment operator token (":=") followed by an expression. The following are valid variable declarations:

```
foo : real
foo, bar: integer := baz + 1
```

3. Tokens

The tokens, or terminal symbols, of the Aeolus language include *identifiers*, *int* and *float* numbers, *litstrings*, and *keywords* (or reserved words) and other *delimiters* (such as arithmetic operators and other types of special characters). In this section, we will discuss rules for the formation of these tokens.

The following general rules apply: the ASCII character set is assumed; blanks must not occur within tokens (except litstrings); line breaks may not occur within any token (thus a single token may not extend over several lines); and blanks as well as line breaks are ignored except where they serve to separate consecutive tokens. Arrangement of tokens on lines may be in free format; in particular, there may be multiple statements on a line. The case of letters is ignored in keywords and identifiers; however, the case of letters in litstrings is preserved.

3.1. Identifiers

An Aeolus identifier must begin with an upper or lower case letter, which may be followed by any number of letters or digits. Also, a *separator* (the underscore character “_”) may be placed between any two characters within an identifier to improve readability; however, a separator may not occur at the beginning or end of an identifier.

$$\langle \text{identifier} \rangle \rightarrow \langle \text{letter} \rangle \{ \{ \langle \text{separator} \rangle \} \langle \text{letter or digit} \rangle \}$$

Examples:

I am an_Aeolus_identifier As_am_I

3.2. Numbers

An Aeolus number is an “int” or “float” number, which may be specified in any base between 2 and 16 inclusive.

3.2.1. Ints

A decimal “int” starts with a digit (“0” through “9”), which may be followed by any number of digits, optionally separated by an underscore character (“_”) for readability. Ints in bases other than 10 may be specified by giving the base (a decimal number between 2 and 16 inclusive), followed by the character “#”, followed by the based number. A based number in a base greater than 10 may include the characters “A” through “F”, as appropriate to the base of the number. (Note that case is not significant for these characters.)

$$\begin{aligned} \langle \text{num} \rangle &\rightarrow \langle \text{digit} \rangle \{ \{ \langle \text{separator} \rangle \} \langle \text{digit} \rangle \} \\ \langle \text{basedit} \rangle &\rightarrow \langle \text{digit} \rangle, \text{“A” .. “F”} \\ \langle \text{basednum} \rangle &\rightarrow \langle \text{basedit} \rangle \{ \{ \langle \text{separator} \rangle \} \langle \text{basedit} \rangle \} \\ \langle \text{int} \rangle &\rightarrow \langle \text{num} \rangle \\ \langle \text{int} \rangle &\rightarrow \langle \text{num} \rangle \text{“\#”} \langle \text{basednum} \rangle \end{aligned}$$

Examples:

1 32767 32_767 2#101010

8#52 16#2A 16#ff 13#42

3.2.2. Floats

A “float” number consists of a *whole part* followed by either a *fractional part* or an *exponent* or both.³ The whole part is a (possibly based) number. The fractional part consists of a fractional point “.” followed by a number with the same base as the whole part. The exponent consists of the letter “E” or “e” followed by a (possibly signed) decimal number, indicating the power of the base by which the float number should be multiplied. The base of a

³Thus, a float number must always begin with a digit.

float number is given as for an int; however, if a float number is based and has an exponent, the character “#” must appear before the exponent. If no base is given, base 10 (decimal) is assumed.

```

<exponent>  →  “E” [ <sign> ] <num>
<sign>      →  “+”, “-”
<float>     →  <num> “.” <num>
<float>     →  <num> [ “.” <num> ] <exponent>
<float>     →  <num> “#” <basednum> “.” <basednum>
<float>     →  <num> “#” <basednum> [ “.” <basednum> ] “#” <exponent>
    
```

Examples:

```
3.14159 8#7.77 0.1e32 2#1011#E-27 16#7f.a2#e+5
```

3.3. Litchars and Litstrings

A *character* is any member of the ASCII character set, including both printable characters (alphanumeric and punctuation) and control characters. Also, some systems may define extensions to the ASCII character set (for instance, graphics characters) which may be considered character tokens on those systems. A *litstring* (literal string) token is a sequence of characters enclosed in single quotes (“’”). To include a single quote as a character in a litstring, the single quote must be doubled (“’’”). A special case of the litstring token is the *litchar* (literal character) token, which is a litstring token consisting of a single character.

```

<litstring> →  “’” { <character> } “’”
<litchar>  →  “’” <character> “’”
    
```

Examples of LITSTRINGS:

```
‘Hello, world’ ‘Don’t be sad’ ‘This is a “litstring”’
```

Examples of LITCHARS:

```
‘a’ ‘?’ ‘’’’ ‘’’’
```

3.4. Comments and Compiler Options

A *comment* is explanatory text inserted into code for the reader’s benefit; it is ignored by the compiler, and does not affect the meaning of the code. In Aeolus, a comment may be placed anywhere within a line where a blank may be placed. It begins with an exclamation point (“!”) and ends either at the next exclamation point or the end of the line on which the comment started, whichever comes first. Thus, comments do not extend over multiple lines.

Examples:

```
! This is an in-line comment. !      !As is this.!
```

```
! This comment goes to the end of this line.
```

A *compiler option* is used to communicate to the compiler the desired settings for various options which the compiler being used may implement, for example, whether range checks for valid variable values are to be generated. A compiler option begins with a dollar sign (“\$”) and ends either at the next dollar sign or at the end of the line on which the compiler option started, whichever comes first.

Examples:

```
$r+ $      $pagelength=84
```

3.5. Reserved Words

The following is a list of the reserved words (keywords) of Aeolus. These words *may not* be used as identifiers! Although the reserved words are shown here in upper case, upper and

lower case may be freely mixed in these words.

ACTION	FORWARD	OVERRIDES
ARRAY	IF	PER
AUTORECOVERABLE	IMPLEMENTATION	PERMANENT
AUTOSYNCH	IMPORT	PROCEDURE
BEGIN	IN	PROCESS
BY	INDEX	PSEUDO
CASE	INITHANDLER	PURE
CONST	INLINE	RECORD
CONSTRAINT	IS	RECOVERABLE
DEFINITION	LOCAL	REGION
DELETEHANDLER	LOCK	REINITHANDLER
DO	LOOP	RETURN
DOMAIN	MODIFIES	RETURNS
DOWNTO	NONRECOVERABLE	SHARED
ELSE	NOT	STRUCTURE
ELSIF	NULL	STRUCTURED
END	OBJECT	THEN
EVENTS	OF	TO
EXAMINES	OPERATIONS	TOPELVEL
EXIT	OTHERWISE	TYPE
FOR	OUT	USING

3.8. Operators and Delimiters

The following are characters or groups of characters used as operators or delimiters (punctuation) in Aeolus.

()	->
{	}	@
	~	"
:=	+=	-=
*=	/=	=
^=	<<=	>>=
&=	%=	=
<	>	<>
<=	>=	
^	<<	>>
&	*	/
+	-	%
,	:	..

3.7. Other Characters

As mentioned before, blanks (except in litstrings) are ignored wherever they are not required to separate other tokens; thus, blanks may be used freely to improve the readability of code. Semicolons (“;”) are ignored in the same way as blanks; thus, semicolons may be used to separate or terminate statements if so desired, but are not required. Non-printable (control) characters are also ignored.

4. Declarations and Scopes

All identifiers in Aeolus code must be introduced by a *declaration*. In this section, the rules for ordering and extent of declarations will be presented.

4.1. Compilation Units and Their Scopes

Those sentential forms described by the Aeolus grammar which may be compiled are called *compilation units*. Compilation units include *object definition parts*, *object implementation parts*, and *processes*. As will be clarified in section 12, an object definition part serves to declare those identifiers—constants, types, and operations—which the object makes available to other objects or processes, while the object implementation part actually provides the code for the object. Other objects or processes may *import* an object definition, and use the identifiers declared by it as if those identifiers had been declared locally.

Every compilation unit implicitly imports the *standard* object, which defines various useful identifiers. (These are listed in Appendix D.) Before any other declarations are given, the compilation unit may import other objects via an *import clause* (see section 12). Then, declarations of constants, types, variables (except in object definitions), and procedures (operations) may be given in any order, as long as the declaration of any identifier used in another declaration textually precedes this use. There are, however, two exceptions to this general rule.⁴ A procedure may be declared *forward*; that is, only its header is declared, while the declaration of its body is delayed until later (see section 11). Also, a type may be declared *forward*; pointer types may then be declared with the forward type as base type (section 6.2.2).

After an identifier has been declared, other declarations and statements may refer to it, as long as these references occur within the *scope* of the identifier. The scope of an identifier extends from the point of its declaration to the end of the *block* in which it was declared. That is, if the identifier was declared in the the declaration part of a compilation unit, its scope extends to the end of that compilation unit; if, however, the identifier was declared in the declaration part of a procedure, its scope extends to the end of the procedure. The scope of identifiers introduced in a USING statement (section 10.2.4) extends to the end of that statement.

The scope defined by a procedure is said to be *nested* within the scope defined by the surrounding compilation unit. As implied by the rules above, identifiers in a nested scope are not *visible* (available for reference) in the surrounding scope. An identifier in an nested scope may have the same name as an identifier in an enclosing scope; the identifier in the enclosing scope is then not visible in the nested scope. Within a scope, however, an identifier must be unique; that is, an identifier may not be declared with the same name as another identifier already declared in the same scope (see below). Procedure declarations may not be nested (within other procedure declarations); thus, the maximum *nesting level* in Aeolus is 2, where the level of a compilation unit is 1.

4.2. Qualified Identifiers

As was stated above, an identifier must be unique within the scope in which it is declared so that the entity which it represents may be correctly identified. However, it often occurs that different object definitions declare constant or type identifiers with the same name, or that different enumeration types have members with the same name,⁵ or that different objects have operations with the same name, or that different records have fields with the same name. Thus, it is sometimes necessary to *qualify* an identifier with the name of its defining type or record to ensure that it is unique.

If types or constants with the same name defined by more than one imported object type⁶ are visible in a scope, or if similarly-named members of different enumeration types are visible in a scope, these names must be qualified with the names of their defining types:

⁴These exceptions allow more general data structures and procedural definitions to be formulated, in particular recursive structures.

⁵This problem may also occur in Pascal, which does not provide for qualification of enumeration types; thus, so-called "holes" may be left in the types.

⁶As we shall see in section 12, the names of imported object definitions may be used as the names of types. Variables declared to be of an object type are said to be *object instances*.

Examples:

```

i : const integer := -10
j : const integer := i + abs (2*i)

```

6. Type Declarations

The declaration of a data type specifies the set of values which members of that type (i.e., variables, record fields, or procedure parameters declared to be of that type) may assume. In the case of structured types, the type declaration also gives a "blueprint" of the structure of members of that type.

The general syntax for declaration of new types is:

<type decl>	→	"type" <new type name> [<formal type param option>] "is" <new type indication>
<formal type param option>	→	"(" <formal type param decl> {", " <formal type param decl> } ")"
<formal type param decl>	→	<param id list> ":", <properly constrained type name> [":= " <scalar const>]
<properly constrained type name>	→	<type name> [<complete params or constraint>]
<complete params or constraint>	→	<actual type param spec>
<complete params or constraint>	→	<constraint spec>
<actual type param spec>	→	"(" <scalar const> {", " <scalar const> } ")"
<constraint spec>	→	"[" <subrange> "]"
<subrange>	→	<scalar const> ".." <scalar const>
<new type indication>	→	["shared"] ["permanent"] <new type spec>
<new type indication>	→	"forward"

As we shall see in the remainder of this section, types fall into two general classes: (possibly parameterized) type identifiers (the names of previously-declared types, including object types), and anonymous types⁸ (including enumerations, index and pointer types, structured types, and locks). The compatibilities of types are discussed in section 9.3.

Types may be *parameterized*, that is, some of the attributes of a type may depend on the values of *formal type parameters*. These parameters are declared in a *formal type parameter option*. (Object types may also be parameterized; see section 12.1.) A formal type parameter may be declared to be of a (possibly constrained) scalar type. The formal type parameters are associated with the values of actual parameters specified in the declaration of variables of that type (see section 8). The values of the type parameters of a member of a parameterized type may be accessed via field dereference operations on that member; for example, if type *t* were declared with type parameter *p*, and variable *v* were declared to be of type *t* with a value of *i* for parameter *p*, then the value of the expression *v.p* would be *i*. A default value for a type parameter may be specified in its declaration; the type is then said to be associated with a delayed constraint (see section 6.1). The value of a type parameter may be specified only in declarations of members of the type, in allocators for members of pointer types, or (if a default value for the type parameter has been given) in a constructor for a constant of the type (see section 9.1); the value of a type parameter may not be otherwise modified. Parameterized types may be nested within other parameterized types; the parameters of the nested types may depend only on the parameters of the enclosing types. Examples of parameterized types are given in section 6.2.3.

⁸The term *anonymous type* refers to the fact that such a type is not given a name by the programmer; however, the effect of an anonymous type is as if that type had been declared with a system-generated name, and that name used

Any type may have an optional indication that members of that type may be *shared*. This attribute is indicated by the use of the keyword *shared* before the type indication. The use of shared variables is explained in section 10.2.5. Similarly, any type declared in a *Clouds object* implementation part (see section 12) may have an optional indication that members of that type are to be allocated in the object's *permanent storage*; this attribute is indicated by use of the keyword *permanent*. The use of permanent variables is explained in section 13.3.

As mentioned in section 4.1, a type may be declared *forward*, that is, its specification may be left temporarily incomplete by use of the keyword *forward* in place of an actual type specification. Forward-declared types may be used only as the base types of pointer types. A complete specification for the forward-declared type must eventually be given within the same scope in which the forward declaration appeared.

6.1. Type Identifiers

The simplest sort of type specification is simply the name of a previously-declared type, optionally followed by a *actual type parameter specification* or by a *constraint specification*:

<new type spec>	→	"new" <constrained type name>
<constrained type name>	→	<type name> [<params or constraint>]
<params or constraint>	→	<complete params or constraint>
<params or constraint>	→	"(" ")"

If a type is declared with a formal type parameter option, a declaration of a member of that type must supply values for the type parameters in an *actual type parameter option*; the types and number of actual type parameters in the actual type parameter option must agree with those in the formal type parameter option. There are three cases, however, in which an empty actual type parameter option ("()") may be given in a declaration of a member of a parameterized type: if the parameterized type is a pointer type, to indicate that specification of the parameters is being delayed until the member of that type is allocated (the values of the type parameters must then be specified in the allocator); if the member of the parameterized type is being declared as a formal procedure parameter, to indicate that the type parameters of the formal procedure parameter will assume the values of those of the actual procedure parameter; or, if the parameterized type declaration included default values for the type parameters, to indicate that the values of the type parameters will be specified in a constructor for a constant value of that type (see section 9.1). Note that this does not preclude specification of type parameter values rather than an empty actual type parameter option in the above three cases.

If no formal type parameter option was declared for the previously-declared type, no actual type parameter option may be given; however, if the previously-declared type was a scalar type (excepting *real*), a *constraint specification* for the scalar type may be provided. The constraint specification indicates the range of values which may be assumed by members of that scalar type; the constraint is not considered to be a part of the type, but rather associated with the type as an attribute. Constraint specifications are further described in section 7.

Several useful predefined types are provided by the object *standard*, which is automatically imported by every compiland. The definition part of *standard* is shown in Appendix D. It defines the following basic scalar types:⁹

- type *integer*, whose variables assume values between MININT and MAXINT;
- type *longint*, whose variables assume values between MINLONGINT and MAXLONGINT;

in place of the anonymous type.

⁹As shown in Appendix D, the types *integer*, *longint*, *unsigned*, and *longuns* may be considered to be new types derived from constraints on an underlying *int* number "type" (which includes all numbers representable by an "int" token), while type *real* may be considered to be derived from a constraint on an underlying *float* number "type" (which includes all numbers representable by a "float" token). The types derived from "int" tokens are denoted collectively as the "int types" in this document.

- type *unsigned*, whose variables assume values between MINUNS and MAXUNS;
- type *longuns*, whose variables assume values between MINLONGUNS and MAXLONGUNS;
- type *boolean*, whose variables assume values FALSE or TRUE;
- type *bit*, whose variables assume values compatible with "int" numbers in the range 0..1;
- type *char*, whose variables assume values of the character set used by the computer on which the program is being used (that is, those values representable by *litchar* tokens); and
- type *real*, whose variables assume real numbers as values.

Scalar types provide the basis for the construction of structured types.

6.2. Anonymous Types

The anonymous types include enumeration types, index and pointer types, structured types, and locks.

<new type spec> → <anonymous type>

6.2.1. Enumerations

An *enumeration* (or *enumerated type*) consists of a list of identifiers which are used as constants in the program. Variables of that enumeration type may assume *only* those identifiers as values. The sequence of the identifiers in the declaration of the enumeration defines an ordering of those identifiers; the ordinal value of the first identifier is 0.

<anonymous type> → "(" <enumer id list> ")"
 <enumer id list> → <id decl> {"," <id decl> }

Example:

type days is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)

6.2.2. Index and Pointer Types

An *index* type is a scalar type, variables of which will be used as indices in FOR loops or as array indices.¹⁰ A variable of an *index* type must be declared locally to the scope within which it is used. Structures may not have components of an *index* type, nor may variables of an *index* type be passed as *out* or *in out* parameters to procedures or operations. The index variable of a FOR loop must be of an *index* type.

<anonymous type> → "index" <properly constrained type name>

Example:

type loopindex is index integer[1..10]

Members of a *pointer* type may assume as values pointers to variables of another type *t* specified in the declaration of the pointer type:

<anonymous type> → "->" <constrained type name>

A pointer value is generated via an *allocator* (see section 9.1). Also, a member of any pointer type may assume the value NIL, which means the variable is not pointing to anything.

¹⁰This declaration provides a hint to the compiler that a variable of this type would be a good candidate to be placed in a register.

Example:

```
type intptr is -> integer
```

As was mentioned in section 4.1 and above, the base type of a pointer may be a *forward-declared* type. This allows recursive data structures to be built with pointers.

6.2.3. Structured Types

Declarations of structured types provide blueprints for arranging groups of components of scalar types or of other structured types into a single structure. Structured types provide the programmer differing levels of abstraction with which to view data, from the most primitive view—sequences of bits—through high-level abstractions such as records.

<anonymous type> → <struct type>

The structured types include *arrays*, *records*, *structures*, and *sets*.

6.2.3.1. Arrays

An *array* is a sequence of a fixed number of components which are all of the same type. The individual components or *elements* of an array are specified by the element's *indices*, which are values belonging to the (possibly constrained) *index type* of the array.

<struct type>	→	"array" "[" [<array index type list>] "]" "of" <type>
<array index type list>	→	<properly constrained type name> { " , " <properly constrained type name> }
<type>	→	["shared"] ["permanent"] <type spec>
<type spec>	→	<constrained type name>
<type spec>	→	<anonymous type>

Example:

```
type smallarray is array [ integer[1..10] ] of integer
```

A declaration of the form

```
array [ T1, T2, ... , Tn ] of T0
```

with an *array index type list* of *n* array index types is considered shorthand for the declaration

```
array [ T1 ] of array [ T2 ] of ... of array [ Tn ] of T0
```

The declaration of an array type is associated with a constraint which gives an indication of the number of elements in a member of that type, so that the size of the member may be determined at the time of compilation. As explained in section 7, a constraint thus associated with a type declaration is inherited as an attribute by members of that type, although the constraint is not considered a part of the type itself. This constraint is derived from constraints on the index types of the array. It is sometimes useful, however, to delay the specification of the size constraint of an array type, especially in the case of a type which is to be used as the type of a formal procedure parameter. This is done by use of formal type parameters in the specification of the index constraints.

Example of a parameterized array type:

```
type anon(first, last : integer) is array [ integer[first..last] ] of integer
type smallarray is new anon(1, 10)
```

Although each member of an array type has a fixed number of elements, arrays with a flexible number of elements (so-called "dynamic" arrays) may be simulated by using pointers to parameterized arrays. Examples of a method for implementing dynamic arrays are presented in Appendix F.

Two important parameterized array types provided by object *standard* are described in the following subsections.

6.2.3.1.1. Bitstrings

A *bitstring* provides the most primitive structured abstraction of data, that of simply a sequence of *bits*:

```
type bitstring(length : unsigned) is array [ unsigned[1..length] ] of bit
```

The length constraint of the bitstring (in bits) is indicated by the value of the parameter "length."

Example:

```
type nybble is bitstring(4)
```

The "system" object, defined for each computer system on which Aeolus may be compiled,¹¹ provides declarations of several useful bitstring types. These types are referred to collectively as the *storage classes*, since they define the units of storage supported by the hardware of most computer systems: types *bit*, *byte*, *word*, *longword*, and *quadword*, with lengths BITSIZE, BYTESIZE, WORDSIZE, LONGWORDSIZE, and QUADWORDSIZE, respectively.

Another important bitstring type, *address*, is also defined by the "system" object. The *address* type is defined as *bitstring(ADDRESSIZE)*. The "system" object defines a constant of type *address* called NIL, which was mentioned in section 6.2.2. The relationship between *address* types and pointer types is discussed in section 9.3.

6.2.3.1.2. Strings

A *string* is a sequence of components of type *char* (that is, a sequence representable by a *litstring* token), terminated by a NUL character (ASCII 0).

```
type string(length : unsigned) is array [ unsigned[1..length] ] of char
```

The maximum length constraint of the string (including the NUL terminator) may be specified by the value of the parameter "length."

Example:

```
type tokenstring is string(128)
```

6.2.3.2. Records

A *record* is a sequence of a fixed number of components which are of possibly differing types. An individual component or *field* of a record is specified by its *field name*, qualified by the name of the record variable to which the field belongs.

A record type declaration specifies the names and types of each field in a variable of that record type. A *parameterized record type* may also have *variant fields*. A variant field consists of a *variant list*, each of whose *variants* is prefaced by a *variant label list*, a list of constants whose types match that of the *discriminant*. The discriminant of a variant field is one of the formal type parameters of the enclosing record type; the name of the discriminant is indicated following the keyword *case* in the variant field specification. The value of the discriminant selects the variant from the variant list one of whose variant labels matches that value.

The rules for modifying type parameters used as record discriminants are the same as for other type parameters, and are described above and in section 9.1.

¹¹At present, Aeolus is supported on the DEC VAX and IBM PC-XT and -AT families of computers; the system objects for these families are named *VAX_System* and *PC_System*, respectively.

<struct type>	→	“record” <field list> “end” “record”
<field list>	→	<field> {“,” <field>}
<field>	→	<field name list> “:” <type>
<field>	→	<variant field>
<field name list>	→	<id decl> {“,” <id decl>}
<variant field>	→	“case” <discriminant name> “of” <variant list> [<variant otherwise>] “end” “case”
<variant list>	→	<variant> {“ ” <variant>}
<variant>	→	<variant label list> “:” <field list>
<variant label list>	→	<variant label> {“,” <variant label>}
<variant label>	→	<scalar const>
<variant label>	→	<subrange>
<variant otherwise>	→	“otherwise” <field list>

Syntax of Record Type Declarations

```

type t (discr1, discr2 : days) is
  record
    case discr1 of
      Monday .. Friday :
        office_no,
        work_phone : integer
    ||Saturday, Sunday :
        home_phone : integer
    end case
    last_name : string (20)
    case discr2 of
      3, 5 .. 7 :
        weekly_rate : integer
    ||8 .. 10 :
        monthly_rate : integer
        benefits    : boolean
    otherwise
      hourly_rate : integer
      temporary   : boolean
    end case
  end record

```

Example of a Record Type Definition

6.2.3.3. Structures

A *structure* is a special case of a record somewhat similar to the *packed record* construct of

Pascal. The declaration of a structure type specifies the storage class which the structure will fit:

```
<struct type> → "structured" <storage class name>
                <field list>
                "end" "structure"
```

(The storage classes were discussed in section 6.2.3.1.1.) A field in a structure typically represents a bitstring or scalar; the fields are packed together consecutively within an object of the specified storage class (without implicit padding), with the first field specified starting at the most significant bit position in the storage class. The compiler checks that the fields declared for the structure together fit into the specified storage class. A structure may not have variant fields.

6.2.3.4. Sets

A *set* type defines a powerset of sets of values of the specified *base type*:

```
<struct type> → "set" "of" <constrained type id>
```

The base type of a set must be scalar. There is no restriction on the number of elements that the base type may have.

Example (see section 6.2.1):

```
type dayset is set of days
```

```
type VAX_processor_status is
  structured longword
    CM,                                     ! 31: Compatibility Mode
    TP                                     : boolean ! 30: Trace Pending
    MBZ1                                   : bitstring (2) ! 29-28: must be zero
    FPD,                                   ! 27: First Part Done
    IS                                     : boolean ! 26: Interrupt Stack
    current_mode                           : 0 .. 3 ! 25-24
    previous_mode                           : 0 .. 3 ! 23-22
    MBZ2                                   : boolean ! 21: must be zero
    IPL                                     : 0 .. 16#1f ! 20-16: Interrupt Priority Level
    MBZ3                                   : byte ! 15-8: reserved (must be zero)
    DV,                                     ! 7: Decimal oVerflow bit
    FU,                                     ! 6: Floating Underflow bit
    IV,                                     ! 5: Integer oVerflow bit
    T,                                     ! 4: Trace bit
    N,                                     ! 3: Negative condition code
    Z,                                     ! 2: Zero condition code
    V,                                     ! 1: oVerflow condition code
    C                                     : boolean ! 0: Carry condition code
  end structure
```

Example of a Structure Type Definition

6.2.4. Locks

A *lock* type may be used to declare variables which in turn may be used to implement locking protocols on particular *values* in some domain.¹²

```

<struct type>   →  "lock" "(" <compat list> ")" ["domain" "is" <type>]
<compat list>  →  <compat> {",", <compat>}
<compat>       →  <id use> ":", "[", <compat id list> "]"

```

A lock declaration includes the specification of a *compatibility list*, which defines, for a given *mode* of the lock, which other modes are compatible with that mode.¹³ The presence of an identifier in a compatibility list serves as a declaration of that identifier as a mode of the lock type; the modes of a lock type may together be thought of as an enumeration. An empty compatibility list indicates that the given mode is incompatible with all other modes.

The lock declaration may also specify the *domain* of values which may be locked. If the domain specification is omitted, a simple lock (i.e., one which does not lock over any particular domain) is assumed.

Examples:

```

type simple_lock is lock (   busy : []           )
                             type file_lock is lock (   read : [read]           ,
                                                         write : []           ) domain is string (20)

```

The declaration of "simple_lock" above defines a lock type with a single mode "busy" which is incompatible with itself; thus, only one client may set a lock variable of type "simple_lock" at any one time. The declaration of "file_lock," on the other hand, defines a lock type over the domain of strings of length 20. Clients may set a lock variable of type "file_lock" on a given string with modes "read" or "write." The "read" mode is specified as being compatible with other settings of "read" mode; the "write" mode is incompatible with itself and with "read" mode. Thus, a client may set the lock with "read" mode on a given string even if several other clients have outstanding settings of the lock with "read" mode on that string; however, a client wishing to set the lock with "write" mode on a given string must wait for all outstanding settings of "read" mode on that string to be released.

7. Constraint Declarations

A constraint, which indicates the minimum and maximum values of the range of values which a variable having that constraint may assume, may be specified for any scalar type except *real*. As was described in section 6, a constraint may be associated with a type declaration; although the constraint is not considered to be part of that type, members of that type inherit the constraint as an attribute. The type being constrained may have already had a constraint associated with it; the new constraint replaces any previous constraint. The effect (or lack thereof) of constraints on type compatibility is described in section 9.3.

A constraint may also be associated with a previously-defined named type, and this association may be given a name which may be used as if it were a type identifier. Such an association is called a *constraint declaration*.¹⁴

```

<constraint decl> → "constraint" <new constraint name> "is" <constrained type name>

```

¹²Note that a lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be obtained on a file name even if that file does not yet exist. The lock structure is directly supported by the Clouds architecture.

¹³A lock may be set with a specified mode only if other modes already set, if any, are compatible with that mode. Thus, a process adhering to a protocol using that lock may wish to block until the requested mode is available. Operations are provided by object *standard* for testing, setting, and releasing locks (see Appendix D).

¹⁴A constraint declaration is similar to a *subtype* declaration in Ada.

A constraint declaration does not create a new type, but rather acts as a *renaming* of the named type, with the new constraint replacing any constraint previously associated with the named type; the constraint name is thus a *synonym* for the named type. As explained in section 9.3, synonyms for a named type are considered to be equivalent to the named type.

Example (see section 6.2.1):

```
constraint weekdays is days[Monday .. Friday]
```

8. Variable Declarations

A *variable declaration* introduces a variable into a process or object implementation part; it associates the variable with a unique identifier and with a fixed type. All variables whose identifiers appear in the same declaration list have the same type. A variable declaration may have an optional initialization clause, which consists of a constant expression of the same type as the variable type. This expression is evaluated, and its value assigned to the variable, before the block is entered in which the variable is declared.¹⁵

```
<var decl> → <id decl list> ":" <type> [":=" <expr>]
```

A variable may also be declared to be located at a specified address:

```
<var address decl> → <id decl> "[" <address expr> "]" ":" <type> [":=" <expr>]
```

The address expression must be a constant expression of type *address*.

Examples:

```

i, j : integer [1 .. 10] := 0
a : array [ integer [1 .. 10] ] of
  record
    realpart, imaginarypart : real
  end record

string_array : array [ integer [1 .. 10], integer [100 .. 200] ]
  of -> string (80)

KB_flag [16#0017] : PC_keyboard_flag
```

9. Expressions

The use of *expressions* allows the programmer to obtain the values of variables and to generate new values by specifying computations to be performed. An expression is constructed from *operands* and *operators*.

9.1. Operands

An operand is either a literal constant (a number or constructor [see below]), an allocator, or a variable.

9.1.1. Variables

A variable may be designated either by a (possibly qualified) simple identifier, or, if the variable is of a structured type, by a *structured variable*, which consists of the variable name followed by *selectors*. Selectors serve to designate the desired component of a variable. A call to a value-returning object operation or procedure (function) may also be used anywhere a variable may be used; in particular, the value returned by such a call may be dereferenced with

¹⁵Variables declared global to a compiland are static, and may be initialized before execution (that is, at compilation or link time).

selectors, if this return value has the appropriate type.

<variable>	→	<id use>
<variable>	→	<rvalue proc call>
<variable>	→	<rvalue obj op call>
<variable>	→	<structured var>
<structured var>	→	<variable> "." <id use>
<structured var>	→	<variable> "->"
<structured var>	→	<variable> "[" <expr> {"," <expr>} "]"
<structured var>	→	<variable> "[" <subrange> "]"

If the variable is of a pointer type, the *pointer dereference* operator (">") may be used to obtain the item referenced by the pointer. If the variable is of a record type, an individual field of the record may be obtained by use of the *field dereference* operator ("."), followed by the name of the field. An individual element of a variable of an array type may be obtained through use of an *element selector* operator, which specifies the index of the array element desired. Thus, the structured variable *a*/*<expr>*/ designates that element of array *a* whose index is the value of the expression *<expr>*. The list of array index expressions in an array element selector, such as

```
a [ <expr 1> , <expr 2> , ... , <expr n> ]
```

is considered shorthand for the sequence of selectors

```
a [ <expr 1> ] [ <expr 2> ] ... [ <expr n> ]
```

for an array *a* declared with *n* dimensions. The type of each element selector expression must be compatible with the type of the corresponding index type of the array (see below).

Examples of variable designations (see section 8):

```
a[5].realpart
```

```
a [i] .imaginarypart
```

```
string_array [i, 110] ->
```

```
string_array [10, 150] -> [80]
```

As well as the ability to index single elements of an array, Aeolus provides the ability to specify a *slice* (or contiguous group of elements) of an array. A slice is denoted by a subrange in an array index expression.

Examples of slice designations:

```
type realarray(first, last) is array [ integer[first..last] ] of real
a : realarray(1, 10)
b : realarray(1, 5)
b := a[1..5]
a[6..10] := b
```

A slice may be applied only to a one-dimensional array.¹⁶

¹⁶Note, however, that any multidimensional array is equivalent to a one-dimensional array the element type of which is an array containing the other dimensions; thus, this restriction merely states that slices may only be applied to the first dimension of an array.

9.1.2. Constructors

As stated above, operands may be literal constants as well as variables. The specification of a literal constant of an integer or real number is simply a token of that type (see section 3). A constant of a structured type, however, must be built by specification of its elements in a *constructor*. Constructors for constants of structured types are built using the following syntax:

```

<constructor>  →  <type id> “ ” “[ <con elem> { “ , ” <con elem> } “[ ”
<con elem>    →  <expr> [ “ : ” <expr> ]
<con elem>    →  <subrange>

```

The constructor is prefaced by the name of the type to which the constant being constructed belongs. The value of each element of the constant is then specified (in the order in which the elements were declared in the relevant type declaration) by an expression which must have the same type as the corresponding element in the structured type. If a structure has several elements of the same type in sequence, the same value may be assigned to each element by specifying an optional *repetition factor* (a [positive] constant integer expression); thus, the constructor element 0:10 would specify that the value 0 be assigned to the next 10 elements in a structure.

The constructor for a constant of a set type merely lists those elements of the base type which are to be included in the set constant. An empty constructor (“[]”) for a constant of a set type implies the so-called *null set*, which is a set with no members.

Constants of bitstring and string types may also be expressed using more traditional styles of constructors for these types. The alternative constructor for a constant of a bitstring type is simply an unsigned binary number (or a number in another base with the equivalent bit pattern) with same number of bits in its representation as the length of the bitstring. We have already seen (in section 3) the alternative constructor for constants of a string type, that is, a string token with enclosing quotes. The string constructor may have no more characters than the maximum length of the string type. When the standard constructor syntax shown above is used for constants of bitstring or string type, each element need not be individually specified; rather, (bit)string constants of smaller (maximum) length may appear as constructor elements, as long as the total (maximum) length of all constructor elements matches the (maximum) length of the target (bit)string type. The individual (bit)string constants are concatenated into the resulting constant.

Constants of other array, record, or structure types may be built only by using the above constructor syntax. Constants of parameterized types may also be specified by means of constructors; the values of the type parameters are given as record field values at the outermost level of the constructor, while the value of the member of the parameterized type is given as a nested constructor (which must be consistent with the values of the type parameters). If the constant thus constructed is to be assigned to a member of the parameterized type, then if a value for a type parameter was given in the declaration of that member, the value for the type parameter may not differ from that given in the declaration of the member (but must still be specified in the constructor); if, however, the type parameter was given a default value (thus associating the parameterized type with a delayed constraint), the value given for it in the constructor may differ from the default value. Note, however, that even in this case the value of the type parameter may not be changed without specifying the complete value of the member at the same time.

Examples of constructors (see section 6 and below):

```

smallarray"[1, 2, 3:5, 4:2, 5]
word"[byte (2#1000), byte (2#0010)]
tokenstring"['Hello, world! ', 'Bye, now. ']
dayset"[Monday, Wednesday, Friday]
dayset"[]

```

9.1.3. Allocators

A value of a pointer type may be generated by an *allocator*. The allocator consists of the keyword *new* followed either by the (completely constrained) name of the type of object to be allocated (and which the pointer variable will reference), or by a constructor for the object to be allocated, including any necessary type parameters. (The rules for specification and modification of type parameter values were described in section 9.1 above.)

```

<allocator>    →  "new" <allocation options>
<allocation options> →  <properly constrained type name>
<allocation options> →  <constructor>

```

Instances of object types may also be generated using allocators of the first form (i.e., using the name of the object type with any necessary object actual parameters; see section 12).

9.2. Operators

The syntax of Aeolus expressions defines *precedence levels* of operators similar to those in Pascal or Modula-2. There are four levels of precedence: the logical NOT operator and the bit-wise complement ("~") operator have the highest precedence (level 1), followed by the *multiplicative operators* (level 2), then the *additive operators* (level 3), and finally the *relational operators* (level 4). When a sequence of operators has the same precedence, the sequence is executed from left to right in textual order. The order of evaluation in an expression may be changed by enclosing parts of the expression in parentheses.

The operators provided by the Aeolus language are listed below. Unless otherwise specified, these are binary operators. In certain cases, the same operator symbol has different meanings when applied to data objects of different types. The intended operation is then identified by the types of the operands.

9.2.1. Arithmetic Operators

These operators apply to compatible operands of type *integer*, *longint*, *unsigned*, *longuns*, and (except for the modulus operator) *real*:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
+	addition	3
-	subtraction	3
*	multiplication	2
/	division	2
%	modulus	2

The operators "+" and "-" may also be used as unary operators. They then denote the *sign* of a term; the "-" operator implies negation, while the "+" operator implies the identity operation. The "%" or modulus operator yields the remainder of an integer division of its (integer) operands:

$x \% y$ gives the remainder of x / y , for $y > 0$.

<factor>	→	"int"
<factor>	→	"float"
<factor>	→	"litchar"
<factor>	→	"litstring"
<factor>	→	<constructor>
<factor>	→	<allocator>
<factor>	→	<variable>
<factor>	→	"not" <factor>
<factor>	→	"~" <factor>
<factor>	→	"(" <expr> ")"
<term>	→	<factor> {"multop" <factor> }
<expr>	→	<simple expr>
<simple expr>	→	["sign"] <term> {"addop" <term> }
<expr>	→	<rel expr>
<rel expr>	→	<simple expr> "relop" <simple expr>

Syntax of Expressions

The division operator ("/"), when applied to integer operands, yields the truncated quotient of its operands.

9.2.2. Bitwise Operators

The following operators may be applied to compatible operands of a bitstring type, except that the right operand of the shift operators is an expression of type *integer*:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
	bitwise OR	3
^	bitwise XOR	3
<<	left shift	3
>>	right shift	3
&	bitwise AND	2
~	bitwise complement (unary)	1

The left and right shift operators yield the value of their first operand shifted left or right (respectively) by the number of positions given by the value of their second operand; the vacated bits are zero-filled. The results of these operators are undefined if the value of the right operand is greater than the length (in bits) of the left operand. The bitwise complement operator ("~") yields the one's complement of its operand.

9.2.3. Address Operators

Arithmetic on pointers is not allowed in Aeolus. However, the bitstring type *address* allows the programmer to perform address computations via explicit conversions from pointer types (see section 9.3). The "system" object for the computer for which a compiland is being compiled (such as *VAX_System* or *PC_System*) defines three named operations on data of type *address*:

addr(v) Returns a value of type *address* representing the storage address of variable *v*, which may be a static or dynamic data item.

`next(a, t [, <expr>])`

Increments the address-type variable *a* by an amount equal to the product of the value of *<expr>* and the size in address units (bytes or words, depending on the system object being used) of the type represented by type identifier *t*. The type of *<expr>* must be one of the "int types." If *<expr>* is omitted, the value 1 (one) is assumed for it.

`prev(a, t [, <expr>])`

The same as *next*, but the address-type variable *a* is decremented rather than incremented.

9.2.4. Logical Operators

The following operators apply to operands of type *boolean* and yield a *boolean* result:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
OR	logical conjunction	3
AND	logical disjunction	2
NOT	logical negation (unary)	1

9.2.5. Set Operators

The following operators apply to compatible operands of a set type and yield a value of the same type:

<i>symbol</i>	<i>operation</i>	<i>precedence</i>
+	set union	3
-	set difference	3
*	set intersection	2
/	symmetric set difference	2

The following named operations are also provided for sets by object *standard*:

`inset(elem, s)` Returns TRUE if the scalar *elem* is currently a member of set *s*, FALSE otherwise. The type of *elem* must be the same as the base type of *s*.

`outset(elem, s)` Returns the value of *not inset(elem, s)*.

`incl(s, elem)` The scalar *elem* is included in (becomes a member of) the set *s*. The type of *elem* must be the same as the base type of *s*.

`excl(s, elem)` The scalar *elem* is excluded from (is no longer a member of) the set *s*. The type of *elem* must be the same as the base type of *s*.

The following statements define the (binary) set operations:

```

inset(x, s1 + s2)  iff  inset(x, s1) or inset(x, s2)
inset(x, s1 - s2)  iff  inset(x, s1) and outset(x, s2)
inset(x, s1 * s2)  iff  inset(x, s1) and inset(x, s2)
inset(x, s1 / s2)  iff  inset(x, s1) <> inset(x, s2)

```

9.2.6. Relational Operators

The relational operators apply to compatible operands of scalar, set, and bitstring types,

and yield results of type *boolean*:

<i>symbol</i>	<i>relation</i>
=	equality
<>	inequality
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The relations “<=” and “>=” denote improper inclusion when applied to sets, while the relations “<” and “>” denote proper inclusion. The relations “=” and “<>” may also be applied to operands of a pointer type. Operands of a bitstring type are considered to be unsigned values of the equivalent length for purposes of comparison. The relations “=” and “<>” are also defined for compatible operands of a pointer, string, record, or array type. All relational operators have the lowest precedence (level 4).

9.3. Type Compatibility

The operands of a binary operation are said to be *compatible* if they are of the same type, that is, if the types of the operand are *equivalent*. The types of operands are equivalent if the operands were declared with the same named type or with the same anonymous type. (This is known as *name equivalence* of types.) Thus, for example, given the declarations

```

type t is array [ integer [ 1 .. 10 ] ] of integer
a : t
b : t
c, d : array [ integer [ 1 .. 10 ] ] of integer
    
```

the variables *a* and *b* have equivalent types (and are hence compatible) since their types both derive from the named type *t*. Also, variables *c* and *d* are compatible, since they share the same anonymous type. However, variable *a* is not compatible with variable *c* since their types, although structurally equivalent, are not name equivalent, since the anonymous type of *c* is not name equivalent to the named type *t*.

The Aeolus language does not allow incompatible operands to appear in an expression; that is, there are no implicit type conversions (coercions). However, it is sometimes desirable to perform operations on operands of differing types. Thus, Aeolus provides the programmer with powerful means of explicit type conversion. An expression takes on the type of its left-most operand unless modified by explicit use of a conversion function.

Every named scalar type definition also implicitly defines a conversion function of the same name as the scalar type. This function may accept as a parameter an operand of any other scalar type, or of a bitstring type of the same storage class. The result of the function invocation is considered to be a scalar of the named type. Thus, if we have

```

i : integer := 0
type fruits is (apples, oranges, lemons)
    
```

then the result of the expression *fruits(i)* is of type *fruits* and has value “apples;” also, the result of the expression *integer(oranges)* is of type *integer* and has value 1.

As mentioned in section 7, a constraint which is associated with a scalar type (by way of a constraint specification in the type’s declaration, or via a constraint declaration) is not considered part of that type, but rather is an attribute which is given to an entity (variable, parameter, or record field) of that type.¹⁷ Thus, a constrained entity is compatible with an entity which has the same type but a different (or no) constraint. For example, considering the declaration

¹⁷Constraints are used for range checking (if enabled) and for determining the sizes of structures, not for type checking.

of type "days" in section 6.2.1 and the declaration of constraint "weekdays" in section 7, if we have the following variable declarations:

d1 : days d2 : weekdays

then d1 and d2 are compatible. However, every type declaration creates a new type; thus, if we had the declarations

type i1 is new integer type i2 is new integer

then entities of type "i1" are incompatible with entities of type "i2," as well as with entities of type integer. Also, had "weekdays" been declared as a type rather than as a constraint, e.g.,

type weekdays is new days [Monday .. Friday]

then variables d1 and d2 would not be compatible.

The type of a so-called *int token* (see section 3.2.1) is determined by the size of the number it represents. Such a number may be assigned to any of the "int types" *integer*, *longint*, *unsigned*, or *longuns* (see section 6.2). Since these types are constrained, however, when range checking is enabled, the number may not be greater than the maximum (or less than the minimum) value representable in the target type.¹⁸ Since the "int types" are declared as types rather than as constraints, an expression of one of these types is not compatible with an expression of some other "int type" without explicit use of the appropriate conversion function.

The conversion functions *integer* and *longint* may also be applied to real expressions; if the resulting value is not too large for the given type, the result is an integer (or longint) value which represents the integral part of the real number (obtained by truncation). The real representation of an integer expression may be obtained by using the conversion function *real*.

An array slice with *n* elements is compatible with any array with *n* elements and a compatible element type. Also, a slice of one element is compatible with any variable of a type compatible with the element type of the slice. Note that this implies the following correspondences:

bit	<=>	bitstringslice[1..1]	<=>	bitstring(1)
char	<=>	stringslice[1..1]	<=>	string(1)

Thus, bit is compatible with array [integer [1..1]] of bit; char is compatible with array [integer [1..1]] of char; and, in general, type *t* is compatible with an array of one element of element type *t*.

The declaration of a named bitstring type implicitly defines a conversion function to that bitstring type from any scalar type representable in that length bitstring. Thus, access may be obtained to the bit representation of data in an explicit manner. This implicit conversion function will also accept a data item of another bitstring type as parameter, as long as the parameter's length (in bits) is no greater than that of the target type. If the length of the source type is less than that of the target type, the resulting value is padded on the right with zeroes to the length of the target type. Also, two named operations are defined by object *standard* for selecting parts of the *word* bitstring type. The *highbyte* and *lowbyte* functions return (as values of type *byte*) the high-order and low-order (respectively) bytes of their word parameter.

The definition of a named pointer type provides a conversion function of the same name from a variable of type *address* to that pointer type. However, the result of such a conversion may be used only for dereferences; it may not be assigned to a pointer variable. Values are assigned to address variables via the *addr* operation discussed above; a value may be assigned to a pointer variable only by use of an allocator (see section 9.1.3), or via assignment from another variable of the same pointer type. The exception to the above rules is a special constant of type *address*, called NIL, defined by each "system" object. The constant NIL, which denotes a null pointer or address reference, may be assigned to a variable of any pointer or address type.

¹⁸This implies that negative numbers may not be assigned to *unsigned* or *longuns* variables, since the minimum value representable in those types is 0.

A definition of a named string type provides a conversion function of the same name to that type from other string types with maximum lengths no greater than that of the target type. The resulting value is a null-terminated string with the same value as the source value, but with the same maximum length as the target type. A number of named operations are provided in object *standard* for other string manipulations and conversions, such as substring extraction and conversions between strings and numbers; these are listed in Appendix D.

The conversion functions discussed above are for the most part somewhat restrictive in the types of the arguments which they will accept. Also, if the range checking option is enabled during compilation, most of these functions will generate runtime range checks of their parameters. However, Aeolus provides a less restrictive (and less safe) means of type conversion. The *retype* function accepts as parameters a value of any type and a type identifier; the result is a value of the type specified by the type identifier, left-justified bitwise. For example:

```
longword_value := retype (integer_value, longword)
```

No type checking is performed; the only restriction is that the target type representation be no smaller (in bitlength) than the type of the source value. Any range checking or filling of unused space (when the target type is larger than the source type) is the responsibility of the programmer.

10. Statements

A *statement* allows the programmer to specify activities such as assignment of a value to a variable, decision branching, or repetitive execution of groups of statements. The so-called *simple statements* do not contain other statements, while the *compound statements* may contain other statements as parts. One or more statements may be grouped into a *statement list*:

```
<stmt list> → <stmt> {<stmt>}
```

for use as a part of a compound statement.

10.1. Simple Statements

The simple statements include the assignment statement, procedure call, object operation call, EXIT statement, RETURN statement, and NULL statement.

```
<stmt> → <simple stmt>
```

10.1.1. Assignment Statements

An assignment statement denotes the replacement of the value of the variable designated by the left-hand side with some function of the value of the expression on the right-hand side. The *assignment operator* used in an assignment statement describes what function of the value of the right-hand side is to be used. The simplest assignment operator is “:=” (pronounced “gets”), which denotes simple replacement. Other assignment operators apply some binary operator to the value of the variable designated by the left-hand side and the value of the expression on the right-hand side; the resulting value replaces the value of the designated variable. An example of such an assignment operator is “+ =” (pronounced “plus-gets”); the assignment statement “i + = 1” is equivalent to “i := i + 1”. The other binary operators

(listed throughout section 9) also have corresponding assignment operators.

<simple stmt>	→	<assign stmt>
<assign stmt>	→	<lhs> "assignop" <rhs>
<lhs>	→	<lhs elem> {",", <lhs elem>}
<lhs elem>	→	<id use>
<lhs elem>	→	<structured var>
<rhs>	→	<expr>
<rhs>	→	["toplevel"] "action" "(" <action invocation> [<timeout option>] ")"
<action invocation>	→	<proc call>
<action invocation>	→	<obj op call>
<action invocation>	→	<lhs> <assign op> <rvalue invocation>
<rvalue invocation>	→	<rvalue proc call>
<rvalue invocation>	→	<rvalue obj op call>
<timeout option>	→	"for" <expr>

Besides a single variable designation, an assignment statement may also take a list of variable designations as its left-hand side; this is called a *multiple assignment*. Here, the value of the expression on the right-hand side is assigned to each of the variables designated on the left-hand side, from the right of the list to the left. For example:

i, j, k := m + 1

is equivalent to the series of assignment statements:

k := m + 1 j := m + 1 i := m + 1

Assignment statements with other assignment operators may also be multiple assignments. The variable designation rightmost in the variable list is used as the left operand for the binary operator. Thus:

i, j, k += 1

is equivalent to the series of assignment statements:¹⁰

k := k + 1 j := k + 1 i := k + 1

An assignment statement may also take the form of an *action invocation*. Action invocations are described in section 13.

10.1.2. Procedure Calls

A procedure call statement activates a named procedure. The procedure call may have a list of actual parameters, which are substituted for the corresponding formal parameters defined by the procedure declaration:

<simple stmt>	→	<proc call>
<proc call>	→	<proc id> "(" <param list> ")"

Procedure calls are more fully described in section 11.2.

¹⁰This may be compared to the equivalent C statement:

i = j = k + = 1;

10.1.3. Object Operation Calls

Object operation calls are similar to procedure calls. However, an object operation must in general be invoked on that instance of the object type given by the *object ID* specified in the operation call:

```

<simple stmt>   →   <obj op call>
<obj op call>  →   <obj id> "@" <obj op id> "(" <param list> ")"
<obj id>       →   <id use>
<obj id>       →   <structured var>

```

Operation calls are more fully described in section 12.3.

10.1.4. EXIT Statements

An EXIT statement specifies the termination of one or more enclosing loops (see section 10.2.4). The keyword *exit* may be followed by either a period ("."), which specifies the termination of the immediately enclosing loop, or by an identifier, which specifies the termination of the enclosing loop with the same name:

```

<simple stmt>   →   <exit stmt>
<exit stmt>    →   "exit" <name option>
<name option>  →   "."
<name option>  →   <id use>

```

An EXIT statement may not appear outside a loop; however, a loop may contain several exit statements.

Examples:

```

        exit .
        exit outer_loop

```

10.1.5. RETURN Statements

A RETURN statement specifies the termination (and return from) the enclosing procedure. The keyword *return* may be followed either by a period (".") if the enclosing procedure does not return a value, or by an expression of the same type as the declared return type if the procedure is value-returning:

```

<simple stmt>   →   <return stmt>
<return stmt>  →   "return" <value option>
<value option> →   "."
<value option> →   <expr>

```

A RETURN statement may not appear outside a procedure body; however, a procedure body may contain several RETURN statements.

Examples:

```

        return .
        return 1.0 - cos (2.0*PI)

```

10.1.6. NULL Statements

A NULL statement indicates that no action is to be taken:

```

<simple stmt>   →   <null stmt>
<null stmt>    →   "null"

```

The NULL statement is useful in constructs in which a statement or statement list would ordinarily appear, but where no action is desired, for instance, in certain cases in a CASE statement or as the body of a procedure stub which is to be filled in later.

10.2. Compound Statements

The compound statements include the IF statement, CASE statement, LOOP statement, USING statement, and REGION statement.

<stmt> → <compound stmt>

10.2.1. IF Statements

The IF statement allows the programmer to construct decision control structures:

```

<compound stmt>  →  <if stmt>
<if stmt>        →  "if" <expr> "then"
                   <stmt list> {<elsif option>} [<else option>]
                   "end" "if"
<elsif option>   →  "elsif" <expr> "then" <stmt list>
<else option>    →  "else" <stmt list>

```

The expressions following the keywords *if* and *elsif* must be of type boolean. These expressions are evaluated in order, and the corresponding statement lists skipped, until one of the boolean expressions yield the value TRUE; the statement list following the keyword *then* after this expression is then executed, and control is then transferred to the statement following the keywords *end if*. If the optional ELSE clause is present, the statement list following the keyword *else* is executed if none of the boolean expressions evaluate to TRUE.

10.2.2. CASE Statements

The CASE statement allows the programmer to specify a multiple-branch decision structure based on the value of a single expression:

```

<compound stmt>  →  <case stmt>
<case stmt>      →  "case" <expr> "of"
                   <case list> [<otherwise part>]
                   "end" "case"
<case list>      →  <case elem> {"|" <case elem>}
<case elem>      →  <case stmt label list> ":" <stmt list>
<case stmt label list> → <case stmt label> {"," <case stmt label>}
<case stmt label> → <scalar const>
<case stmt label> → <subrange>
<otherwise part> → "otherwise" <stmt list>

```

```

if inset (ch, charset['a' .. 'z']) then
  process_alpha (ch)
elsif inset (ch, charset['0' .. '9']) then
  process_digit (ch)
elsif inset (ch, charset['!', '!', ';']) then
  process_punctuation (ch)
else
  error_message ('Not a valid character')
  ch := ' '
end if

```

Example of an IF Statement

First, the expression following the keyword *case* is evaluated. This expression must be of a scalar type (excluding type *real*); also, each of the *case statement labels* must be a scalar constant or a scalar constant subrange the type of which is compatible with that of the expression (no value may occur or be contained in more than one label). Second, if one of the *case statement label lists* contains a label which matches or contains the value yielded by the expression, the statement list following that label list is executed, and control is transferred to the statement following the keywords *end case*. If none of the labels matches the value of the expression, and the optional *OTHERWISE* clause is present, the statement list following the keyword *otherwise* is executed; if no *OTHERWISE* clause is present, control is transferred to the end of the *CASE* statement.

10.2.3. LOOP Statements

The *LOOP* statement allows the programmer to specify that a statement list be executed repeatedly, either for a specified number of iterations, or while some condition is true, or until the loop is explicitly exited. The basic form of the *LOOP* statement, without the optional *iteration clause*, is essentially an infinite loop: the enclosed statement list is executed until the loop is explicitly exited by means of an *EXIT* statement (see section 10.1.4).

```

case ch of
  'a' .. 'z' :
    process_alpha (ch)
  || '0' .. '9' :
    process_digit (ch)
  || ', ' .. ' ; ' :
    process_punctuation (ch)
otherwise
  error_message ('Not a valid character')
  ch := ' '
end case

```

Example of a CASE Statement

<compound stmt>	→	<loop stmt>
<loop stmt>	→	[<iteration clause option>] <basic loop>
<loop stmt>	→	<loop id dec> “:” [<iteration clause option>] <basic loop> <loop id use>
<basic loop>	→	“loop” <stmt list> “end” “loop”
<iteration clause option>	→	“while” <expr>
<iteration clause option>	→	“for” <index id> “:=” <expr> <direction> <expr> [<by clause>]
<direction>	→	“to”
<direction>	→	“downto”
<by clause>	→	“by” <expr>

Syntax of LOOP Statements

Two iteration clause options are available for control of the repetitive execution of the LOOP construct. The simplest of these two options is the WHILE clause, which specifies that the loop is to be continued as long as some condition is fulfilled. The expression following the keyword *while* must be of type boolean. This boolean expression is evaluated before each execution of the statement list enclosed by the LOOP construct; this repetition continues as long as the expression yields the value TRUE.

The second iteration clause option is the FOR clause, which specifies that a progression of values is to be assigned to a variable during the repetitive execution of the loop. The identifier following the keyword *for* is called the *loop index variable*; this identifier must have been declared as a variable of an index type (see section 6.2.2). The loop index variable may not be the target of an assignment statement within the statement list enclosed by the LOOP construct. The direction of the progression of values is specified by the use of one of the <direction> keywords *to* or *downto*; the former specifies an increasing progression (that is, the loop index is incremented on each iteration), while the latter specifies a decreasing progression (the loop index is decremented). The ordinal amount by which the loop index is incremented or decremented on each iteration is specified by the value of the expression following the keyword *by* in the optional BY clause; this expression must yield a positive value. If no BY clause is given, the value 1 is assumed for the increment or decrement. The starting value of the progression is given by the value of the expression following the token “:=”, while the ending value of the progression is given by the value of the expression following the <direction> keyword; the types of these two expressions must be compatible with the base type of the loop index. All three expressions (starting value, ending value, and increment) are evaluated before the loop is entered. Execution of the statement list enclosed by the LOOP construct continues until the value of the loop index variable exceeds the ending value, in the sense of the direction of the progression.

A LOOP statement may optionally be qualified by a *loop identifier*. The appearance of this identifier at the start of the construct is considered to be the declaration of the loop identifier; if a loop identifier is specified, the same identifier must appear after the *end loop* keywords. The scope of the loop identifier is the extent of the LOOP statement which declared it. A loop identifier may be used in the <name option> clause of an EXIT statement (see section 10.1.4) to specify the termination of an enclosing loop with that name.²⁰

Examples:

```

InOut @ ReadChar (ch)
while ch <> ' ' loop
    process_char (ch)
    InOut @ ReadChar (ch)
end loop

for ch := 'z' downto 'a' by 2 loop
    process_char (ch)
    for i := integer (ch) to 10 * integer (ch) loop
        InOut @ WriteChar (ch)
    end loop
end loop

```

²⁰This is especially useful when the named loop is not the loop immediately enclosing the EXIT statement.

```

outer :
  loop
    loop
      InOut @ ReadChar (ch)
      if ch = '.' then
        number_of_sentences += 1
        exit outer
      elsif ch = ' ' then
        exit .
      end if
      process_char (ch)
    end loop
    number_of_words += 1
    skip_spaces ()
  end loop outer

```

10.2.4. USING Statements

The USING statement allows the programmer to “alias” parts of complicated variable designators. These “aliases” may then be used in place of those parts of the designators within the statement list enclosed by the USING construct:

```

<compound stmt>  →  <using stmt>
<using stmt>    →  “using” <use spec list> “do”
                  <stmt list>
                  “end” “using”
<use spec list> →  <use spec> {“,” <use spec>}
<use spec>      →  <id decl> “for” <variable>

```

The effect of a USING statement is the creation of a nested scope for the extent of that USING statement; the identifiers on the left-hand sides of each <use spec> in the <use spec list> are considered to be declared within this scope. The effective address value yielded by the variable designation on the right-hand side of a <use spec> is assigned to the identifier on the left-hand side of that <use spec>²¹ (that is, the identifier denotes the so-called *lvalue* of the variable designation). That identifier may then be used as shorthand for the variable designation within the statement list enclosed by the USING statement. An identifier declared in a <use spec> may also be used in the variable designation of any <use spec> following it textually.

Example (see section 9.1):

```

using s1 for string_array [10], s2 for s1 [150] ->, aj for a [j] do
  InOut @ WriteString (s1 [149] ->)
  InOut @ WriteString (s2)
  InOut @ WriteChar (s2 [80])
  s2 := string80["thanks for all the fiche.", ' ':55]
  s2 [1] := 'T'
  aj.imaginarypart := 0.0
end using

```

10.2.5. REGION Statements

The REGION statement implements a *critical region* protocol for mutual exclusion on execution of a region (list of statements). In the header of the REGION statement, the programmer specifies a variable designator on which the statements enclosed by the REGION statement

²¹This value is also considered to be a good candidate to be placed in a register.

will operate:

```

<compound stmt>  →  <region stmt>
<region stmt>   →  "region" <variable> "do"
                  <stmt list>
                  "end" "region"

```

The type of the entity designated by <variable> must have the attribute *shared* (see section 6), which indicates that access to the entity may be safely shared among concurrent processes. To ensure safe access, a shared entity may appear as the target of an assignment only within a REGION statement designating that entity. Note that this entity may be an element of a structured type such as an array or record.

The effect of the attribute *shared* is to associate the shared entity with a semaphore. (Note that if an array is composed of shared elements, this implies that the array is associated with an array of semaphores.) The semaphore associated with a shared entity is used to control access to all regions designating that entity by the variable designator <variable>. The first process to enter such a region when the semaphore is free will then gain exclusive access to that region; other processes attempting to enter that region, or other regions designating the same entity, will be forced to wait in a queue on the semaphore. When a process leaves the region, it signals the semaphore so that the next process in the queue gains access (in a first in—first out manner).

Example (see section 9.1):

```

region a [j] do
  a [j].realpart := 10.5
  a [j].imaginarypart := 0.2
end region

```

11. Procedures

The procedure construct provides a type of control abstraction known as *procedural abstraction*. A statement list may be associated with an identifier by means of a *procedure declaration*; then, the use of that identifier in a *procedure call* statement implies the activation of that statement list, with the possible substitution of *actual procedure parameters* for *formal procedure parameters*. Also, a procedure may be declared as *value-returning*, in which case the procedure may be activated within an expression; the *return value* of the procedure call may then be used by the expression for further computation.

11.1. Procedure Declarations

A procedure declaration consists of a *procedure header* and a list of statements enclosed by a *procedure block*. The header contains declarations of the procedure's name and (optionally) its formal parameters, return type, and procedure attributes; the block may contain, besides the statement list, any *local declarations* of constants or variables. The procedure block may be replaced in the declaration by the keyword *forward*, which indicates that the procedure block will appear in a second declaration of the procedure which must appear later within the same compilant; the specification of parameters, return type, and attributes must appear in the procedure header in the so-called *forward declaration*, and may not be repeated in the procedure header of the second declaration.

The visibility of constants and variables declared locally to a procedure, as well as the visibility within a procedure of items declared in a procedure's environment was discussed in section 4.1. The values of locally-declared variables are undefined upon entry to the procedure unless these variables have an associated initialization clause. Note that a procedure may not be declared within the declaration of another procedure; that is, a procedure declaration may not be nested. The use of a procedure's identifier in a procedure call within its procedure block declaration denotes the recursive activation of the procedure.

<procedure decl>	→	<proc hdr> "forward"
<procedure decl>	→	<proc hdr> <proc block> "procedure"
<proc hdr>	→	"procedure" <proc name decl> "(" [<formal params>] ")"
		["returns" <properly constrained type name>]
		"is" [<proc attr>]
<formal params>	→	<formal param section>
		{ " , " <formal param section> }
<formal param section>	→	<id decl list> " : "
		[<param mode>] <constrained type name>
<param mode>	→	"in"
<param mode>	→	"out"
<param mode>	→	"in" "out"
<proc attr>	→	"inline"
<proc attr>	→	"pure"
<proc block>	→	[<proc block decl pt>] <stmt pt>
<proc block decl pt>	→	<proc declaration> { <proc declaration> }
<proc declaration>	→	<const or var decl>
<proc declaration>	→	<type decl>
<proc declaration>	→	<constraint decl>
<proc declaration>	→	<var address decl>
<stmt pt>	→	"begin" <stmt list> "end"

Syntax of Procedure Declarations

The formal parameters declared in a procedure header act as "placeholders" in the procedure block for the actual parameters to be passed in a procedure call. At the time of a procedure call, the formal parameters are replaced by the corresponding actual parameters. The type of a formal parameter may be any (possibly constrained) named type. The scope of a formal parameter is the same as that of the local variables of the procedure, that is, its scope is the extent of the procedure. There are three possible *modes* of a procedure parameter:

- in** An *in* procedure parameter acts as a local constant to the procedure whose value is provided by the corresponding actual procedure parameter. The actual parameter may be any expression of a type compatible with the formal parameter. As in the case of constants, an *in* parameter may not appear as the target of an assignment statement nor as an actual parameter corresponding to a formal parameter of mode *out* or *in out*.
- out** An *out* procedure parameter acts as a local variable to the procedure; the value of the parameter is assigned to the corresponding actual procedure parameter no later than upon return from the procedure. The actual parameter must be a variable of a type compatible with the formal parameter; the identity of this variable is determined when the procedure is invoked, and may not change during the invocation. The value of an *out* procedure parameter is undefined upon procedure entry.
- in out** The same as *out*, except that the initial value of the procedure parameter is provided by the corresponding actual parameter.

If no mode is specified for an formal procedure parameter, the mode is assumed to be *in*.

A procedure may be specified to have a return type, in which case it is called a value-returning procedure or *function*. The type of the return value may be any (possibly constrained) named type. The value to be returned must be specified by an expression in a RETURN statement (see section 10.1.5); the type of this expression must be compatible with the return type.

A procedure declaration may also specify certain *attributes* for the procedure. These include *inline*, which specifies that the compiler should insert the procedure code "inline" at the point of the call to the procedure, rather than to compile an actual call to the procedure; and *pure*, which indicates to the compiler that the procedure does not modify any non-local variables or make any calls to non-pure procedures.²²

Example (see Appendices A and B for more examples):

```

procedure factorial ( i : in integer ) returns integer is pure
begin
  if i <= 1 then
    return 1
  else
    return i * factorial(i-1)
  end if
end procedure ! factorial !

```

11.2. Procedure Invocations

The invocation of a procedure may take place either as a *procedure call* statement (see section 10.1.2), or (if the procedure has been declared as value-returning) within an expression:

```

<proc call>   →   <proc id> "(" [ <param list> ] ")"
<param list> →   <expr> { ",", <expr> }

```

The values of the actual procedure parameters specified in a procedure call are evaluated before the call, and these values are substituted for the formal parameters within the called procedure. For *in* formal parameters, the actual parameter may be an expression. An actual procedure parameter which is substituted for a formal parameter of mode *out* or *in out* must be a variable designator; the selectors for components of structured variables are evaluated before parameter substitution takes place (that is, before the procedure call). The type of each actual parameter must be compatible with that of the corresponding formal parameter, and the number of actual parameters must match the number of formal parameters for that procedure, unless a parameter has been specified as *arbitrary* (more to come on this).

Example:

```
factorial(2*j)
```

12. Objects

The object construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction.

An Aeolus object may also have parameters indicating, for instance, sizes or element types of the abstraction implemented by the object; thus, an object implementing a bounded stack abstraction may be parameterized by the element type and maximum number of elements of the stack. Then, various *instances* of the bounded stack object may be created (instantiated) with differing element types and sizes; the implementation of the object need not be concerned with details such as the element representation, and the programmer does not need to create new object types for each combination of element type and stack size. Support for such *generic objects* increases the level of abstraction available to the programmer, and makes possible the creation of libraries of reusable object types.

²²This attribute gives the compiler a hint that certain optimizations may be possible in this procedure. This attribute is used at the programmer's risk; that is, the compiler does not attempt to verify that the procedure is actually pure.

The object construct also provides a safe separate compilation mechanism. The separation of an object specification into a *definition part* and an *implementation part* allows checking across the interface to an object, as well as allowing the use of an object definition before the corresponding implementation part is finished (thus facilitating top-down design).

12.1. Object Definition Parts

The definition part of an object defines the interface of the object with other compilands. It specifies the attributes of the object itself as well as the constants, types, and operations which the object provides to other objects and to processes.

Specification of the *autosynch* keyword in an object definition header causes code to be generated for automatic synchronization of object operation invocations based on programmer-supplied indications of operation effects (see below). This mechanism provides a simple read/write locking protocol; it may be used with any object class (see below).²³

The *object class* is also specified in the object definition header. The object classes fall into two groups: the non-Clouds object classes (*pseudo* and *local*) do not use any of the Clouds

<comp unit>	→	<obj def>
<obj def>	→	<obj def hdr> <obj visible decls> "end" "definition" "."
<obj def hdr>	→	"definition" "of" <obj class> ["autosynch"] "object" <obj name decl> [<generic option>] "is"
<obj class>	→	"pseudo"
<obj class>	→	"local"
<obj class>	→	"nonrecoverable"
<obj class>	→	"recoverable"
<obj class>	→	"autorecoverable"
<generic option>	→	"(" <obj formal param list> ")"
<obj formal param list>	→	<obj formal param> {"", " <obj formal param> }
<obj formal param>	→	<id decl> ":" <generic type>
<generic type>	→	<constrained type id>
<generic type>	→	"type"
<obj visible decls>	→	[<imports>] <decls&specs>
<imports>	→	"import" <import name> {"", " <import name> }
<decls&specs>	→	[<visible decl part>] [<op spec part>]
<visible decl part>	→	<visible decl> {<visible decl> }
<visible decl>	→	<const decl>
<visible decl>	→	<type decl>
<visible decl>	→	<constraint decl>
<op spec part>	→	"operations" <op spec list>
<op spec list>	→	<op spec> {<op spec> }
<op spec>	→	"procedure" <proc name decl> "(" <formal params> ")" ["returns" <properly constrained type name>] [<op effect>]
<op effect>	→	"examines"
<op effect>	→	"modifies"

Syntax of Object Definition Parts

²³For more information on the mechanisms supplied by the Clouds system to support synchronization and recovery, see [Allc83b].

facilities for action or object management, and are thus similar to *modules* in Modula-2 (for pseudo-objects) or to *generic packages* in Ada (for local objects), while the so-called Clouds object classes (*nonrecoverable*, *recoverable*, and *autorecoverable*) may make use of the object management facilities and (for recoverable and autorecoverable types) the action management facilities. The definitions of the object classes are as follows:

non-Clouds object classes:

- pseudo** (or pseudo-local) A class of local (non-Clouds) object of which there is only one instance. This object class is used mainly for definition of system libraries, for interfacing with (separately-compiled) collections of procedures written in another programming language, for abstraction of machine and system dependencies, and as a basic separate-compilation mechanism.
- local** The standard class of non-Clouds object, which may have multiple instances. Object management is provided by the Aeolus runtime system. Unlike Clouds objects, a local object may have no existence independent of the process or object which created it. Local objects simulate Clouds objects without incurring the expense of the use of the action and object management facilities.

Clouds object classes:

- nonrecoverable** The basic class of Clouds object. Objects of class *nonrecoverable* make use of the object management facilities, but may not contain recoverable areas or action event handlers.
- recoverable** The "roll-your-own recovery" type of Clouds object, as opposed to the *autorecoverable* class of objects (described below), which provides completely automatic recovery. In some cases, the programmer may be able to use knowledge of the semantics of the object and its operations to program synchronization and recovery mechanisms more efficient than the automatic mechanisms supplied by the *autorecoverable* class of objects. Automatic recovery involves checkpointing of the entire object state; automatic synchronization is based on a simple read-write model of operation interactions on entire operations. As will be discussed in section 13, Aeolus provides facilities that allow the programmer to specify which parts of the object state are to be checkpointed (recoverable areas), to access information about the states of actions and to change these states (via operations on the action manager), and to control the recovery process by specification of what is to be done during action events (action event handlers); also, the programmer may specify finer-grained locking mechanisms for greater control of synchronization (via the *lock* type; see section 6.2.3.8). Only *recoverable* objects may contain recoverable area specifications and action event handler specifications.
- autorecoverable** As mentioned above, *autorecoverable* objects provide completely automatic recovery. The entire object state (the global variables of the object) is recoverable, and the default event handlers are used.

An instantiation of an object (other than of class *pseudo*) is created by use of an allocator (see section 9.1); the allocator yields a capability to the newly-created object instance, which may be assigned to a variable of that object type. The variable may thereafter be used to qualify operation invocations on that object instance. The *init* object event handler (see section 12.2 below) for the object, if specified, as well as any variable initializations required by the object, is executed during the instantiation process.

If an object is to be generic, the programmer must specify the formal object parameters in the object definition header. Such a parameter may be of any (possibly constrained) named type, or it may be an identifier which is to be used within the object implementation as a type identifier (specified by the keyword *type* in place of a type name in the formal parameter specification). As stated above, these parameters may be replaced by actual parameters (in the

```
definition of local object bounded_stack ( size : unsigned, elem_type : type ) is
```

```
! Definition of a generic bounded stack object with size SIZE
! and elements of type ELEM_TYPE.
```

```
operations
```

```
procedure push ( elem : elem_type ) modifies
! Places ELEM on the top of the stack, if the stack is not full.
```

```
procedure pop ( ) returns elem_type modifies
! Removes the top element of the stack and returns it.
! The return value is undefined if the stack is empty.
```

```
procedure top_elem ( ) returns elem_type examines
! Returns the top element of the stack without removing it.
! The return value is undefined if the stack is empty.
```

```
procedure empty ( ) returns boolean examines
! Returns TRUE if the stack has no elements, FALSE otherwise.
```

```
procedure full ( ) returns boolean examines
! Returns TRUE if the stack has SIZE elements, FALSE otherwise.
```

```
end definition. ! bounded_stack !
```

Example of an Object Definition

form of expressions or type names) when a variable of that object type is declared; the values of the actual parameters then determine the sizes, element types, etc. of that instance of the generic object (see section 6.3).

Following the object definition header, the programmer may specify the names of other object definitions which contain constant or type specifications to be used in this object definition. The names of these objects are specified in an *import* clause. Definitions imported in an object's definition part are also available in that object's implementation part.

After any necessary imports are specified, the declarations of the object definition are given. These are called its *visible declarations* since the declarations are available publically to any object which imports the object definition. The visible declarations of an object may include specifications of constants, types, or operations, but not of variables. The specifications of the object's operations are listed following the keyword *operations*. Each specification consists of the procedure's header (see section 11.1), optionally followed by one of the operation effect keywords *examines* or *modifies*, which indicate that the operation reads from or writes to the object's state, respectively. This information is used by the compiler to generate automatic read or write locking for each operation if the *autosynch* attribute is specified for the object. If no operation effect is specified, the compiler assumes that the operation neither reads nor modifies the object state, and thus no automatic locking is done for that operation.

12.2. Object Implementation Parts

The implementation part of an object provides the actual code for the operations of the object, as well as the definitions of any private constants, types, variables, or procedures needed by the object. The definition part of the object being implemented is implicitly imported by the implementation part; thus, the attributes, formal object parameters, and public constant, type, and operation specifications provided by the definition part may not be repeated in the implementation part. Also, as mentioned in the previous section, any objects imported by the definition part are also available in the implementation part. The implementation part may import other objects as well via its own *import* clauses. All constants, type definitions, and operations declared in the objects made available by any of these methods are visible in the implementation part; also, the names of these imported object types may be used as the types of variables declared in the implementation part. Such variables must be initialized by use of an allocator (see section 9.1).

If the *recoverable* class is specified in the definition header of the object being implemented, the programmer may give an *action events part* and/or a *per-action part* in the object's implementation part. Action events part and per-action part specifications are described in sections 13.2 and 13.3, respectively.

An object implementation part must include full declarations of all operations specified in the object's definition part. As with the full (second) declaration of a forward-declared procedure, the parameter list of an operation is not given in its full declaration. Constants, types, or procedures declared in the <obj imp block> but not specified in the object's definition part are not visible to other compilands importing the object. Variables declared in the outer level

<comp unit>	→	<obj imp head> <obj imp block> <obj imp tail>
<obj imp head>	→	<obj imp hdr> <imports> <event part>
<obj imp hdr>	→	"implementation" "of" "object" <obj name> "is"
<event part>	→	"action" "events" <override list>
<override list>	→	<override> {"," <override> }
<override>	→	<id decl> "overrides" <id use>
<obj imp block>	→	[<obj block decls pt>] <obj events pt>
<obj block decls pt>	→	<obj block decls> {<obj block decls> }
<obj block decls>	→	<const or var decl>
<obj block decls>	→	<type decl>
<obj block decls>	→	<procedure decl>
<obj block decls>	→	<recoverable area spec>
<obj block decls>	→	<per-action spec>
<obj events pt>	→	<inithandler spec> <reinithandler spec> <delethandler spec>
<inithandler spec>	→	"inithandler" "is" <proc block> "end" "inithandler"
<reinithandler spec>	→	"reinithandler" "is" <proc block> "end" "reinithandler"
<delethandler spec>	→	"delethandler" "is" <proc block> "end" "delethandler"
<obj imp tail>	→	"implementation" "."

Syntax of Object Implementation Parts

of the `<obj imp block>` are global to the object, and are static (“own”) variables; that is, the values of such variables survive between calls to the object’s operations. The global variables of an object are called collectively the object’s *state*. In an object of class *recoverable*, part of the object state may be specified to be in a *recoverable area*. Recoverable areas are described in section 13.2.

An object implementation part contains specifications of handlers for the so-called *object events*. The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator (see section 12.1); the *reinit* or object reinitialization event, the handler for which is executed if the object has registered its desire for reinitialization with the action manager when the system is reinitialized after a crash or partition (see Appendix E); and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed. No default handlers for the object events are supplied; if no action is desired for an event, the programmer must supply a NULL statement as the handler body.

12.3. Object Operation Invocations

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself):

```

<obj op call>  →  <obj spec> “@” <obj op id> “(” <param list> “)”
  <obj spec>   →  <id use>
  <obj spec>   →  <structured var>
    
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially like those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities (i.e., the so-called “Clouds objects”).²⁴ Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level. (More to

```

implementation of object bounded_stack
  ! ( size : unsigned, elem_type : type ) ! is
    
```

! More to come.

```

end implementation. ! bounded_stack !
    
```

Example of an Object Implementation

²⁴This is because the code for pseudo-objects and for local objects is actually linked into the code of the compiland using these objects, whereas the code for Clouds objects is physically separate from the code of the invoking compiland. This code is paged in on demand by the object manager (see [All83b]).

come on operation invocation semantics.)

Examples (see previous two sections):

```
s1 : bounded_stack ( integer, 10 )
s2 : bounded_stack ( real, 5 )

if not s2 @ full ( ) then
  s1 @ push ( 42 )
elsif not s2 @ empty ( ) then
  r := s2 @ pop ( ) + 3.14159
end if
```

13. Actions

The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display “all-or-nothing” behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed. The rationale for the action concept and the mechanisms supporting it in the Clouds system are described in [Allc83b].

Support for actions in the Aeolus language is relatively low-level. The methodology of programming with actions is not at present well-understood compared with experience in programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable areas*, *permanent* and *per-action variables*, and *action invocations*.

13.1. Action Events

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action management system. The *action events* include:

<i>event name</i>	<i>purpose</i>
BOA	beginning of action
toplevel_precommit	prepare for commit for a toplevel action
nested_precommit	prepare for commit for a nested action
commit	normal end of action (EOA)
abort	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described in section 12.1, by specification of the keyword *autorecoverable* in the header of an object definition the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires checkpoints of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword *recoverable* is specified in the definition header of the

object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords *action events*, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override:

```

    <event part>   →  "action" "events" <override list>
    <override list> →  <override> {"", <override>}
    <override>    →  <id decl> "overrides" <id use>

```

Thus, for example, the specification (say, in an object called "stack"):

```

    action events
    stack_BOA overrides BOA, stack_precommit overrides precommit

```

indicates that the default handlers for the *BOA* and *precommit* action events are to be replaced by the procedures named "stack_BOA" and "stack_precommit," respectively, for the "stack" object only.

13.2. Recoverable Areas

As mentioned in section 8, if an object being implemented is specified to be *recoverable*, then some of its variables may be declared in a *recoverable area*:

```

    <recoverable area spec> →  "recoverable"
                               <var decl> {<var decl>}
                               "end" "recoverable"

```

The state of a recoverable area which has been touched by an action is maintained on a *version stack* by a Clouds action manager, and is saved to permanent storage upon commit of the action which touched it. If an action which touched a recoverable area is aborted, the version of that area which existed before the action touched it is restored.²⁶ Thus, the use of recoverable areas allows the programmer to provide finer granularity in the specification of that part of the object state which must be checkpointed, since the use of automatic recovery on object (the *autorecoverable* object class) performs checkpoints on the entire state of the object.

The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

Example:

```

    recoverable
    i, j : integer
    a : realarray(1,10)
    end recoverable

```

13.3. Permanent and Per-Action Variables

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer make take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables*.

As mentioned in section 6, any type may be given the attribute *permanent*. This attribute indicates that members of that type are to be allocated on the *permanent heap*, a dynamic

²⁶For more information on the semantics of recoverable areas and the mechanisms to support them, see [Allc83b].

storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow paged* during the *oplevel precommit* action event.²⁶ Any type which has as its base or element type a type with the attribute *permanent* inherits that attribute. Although other permanent types—such as permanent array types—may be declared, the only permanent types which may be used as the types of variables are permanent pointer types. In view of the support provided by the Clouds system, it is strongly recommended that the following discipline be observed in the use of permanent variables: those variables generated within an action by use of an allocator may be freely assigned values within that action; however, pre-allocated permanent variables—that is, those allocated outside the current action (by some other action)—should be assigned values only within a *oplevel precommit* event handler. However, this discipline is not enforced by the compiler.

Aeolus also provides the per-action variable construct. An object implementation part of class *recoverable* may declare a single per-action variable section:

```
<per-action spec>  →  "per" "action"
                    <var decl> {<var decl>}
                    "end" "per" "action"
```

A per-action specification resembles a recoverable area specification, and the semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: *Self* and *Parent*, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the *oplevel precommit* action event to the actual data structure in permanent storage. The use of permanent and per-action variables is described more fully in Appendix G.

13.4. Action Invocations

As mentioned in section 10.1.1, the right-hand side of an assignment statement may also take the form of an *action invocation*:

```
<rhs>  →  ["oplevel"] "action"
          "(" <action invocation> [<timeout option>] ")"
<action invocation>  →  <proc call>
<action invocation>  →  <obj op call>
<action invocation>  →  <lhs> <assign op> <rvalue invocation>
<rvalue invocation>  →  <rvalue proc call>
<rvalue invocation>  →  <rvalue obj op call>
<timeout option>  →  "for" <expr>
```

Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. The action ID may then be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort. (The interface to the Clouds action manager is described in Appendix E.) If the keyword *oplevel* is specified, the action is created as a "top-level" action; that is, as an action with no ancestors.²⁷ Otherwise, the

²⁶For more information on the management of permanent heap storage, see [Pitt84] and [Wilk86].

²⁷Thus, as we shall see, a top-level action cannot be affected by an abort of any ancestor of the action which created it.

action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* (in milliseconds) may be specified; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used.²⁸ Only an operation or internal procedure of a *recoverable* or *autorecoverable* object may be invoked as an action.

The semantics of an action invocation is as follows: the action manager operation *CreateAction* is invoked with the name of the operation to be performed as well as the list of arguments to be passed to that operation.²⁹ The action manager then invokes the BOA event handler on the object to which the operation belongs. Next, the action manager creates and dispatches a process in which the operation code runs. An attempt by the operation to return to its caller is considered an implicit attempt to commit the action, and will cause control to transfer to the *Commit* operation of the action manager, which terminates the process and invokes the precommit event handler of each object touched by the action. (An explicit invocation of the *Commit* operation has the same effect.) If precommit of the object is successful, the action manager then invokes the commit event handler of each touched object. If the action (or one of its ancestors) invokes the *Abort* operation of the action manager, the action manager terminates the process corresponding to the action and invokes the abort event handler of each object touched by that action.

It may sometimes occur that an object operation may be called either as an action invocation or as an ordinary object operation invocation. In the case that an operation is invoked normally (that is, not as the target of an action invocation), an invocation of the action manager operation *Commit* by the operation will cause the action manager to merely return control to the point of invocation of the original operation; thus, in this case the *Commit* call is effectively a normal procedure return. On the other hand, an invocation of the *Abort* operation by an operation invoked normally will cause the parent action of the invoker of the original operation to abort.³⁰ Thus, in the case of normally-invoked operations, a call to the *Abort* action manager operation provides a mechanism similar to an exception-handling mechanism with a single exceptional condition ("error").

14. Processes

The final structuring feature of the Aeolus language provides an abstraction of the *process* concept of the Clouds system. (The process is analogous to the *program* construct of Pascal or Modula-2.) The invocation of a process provides activity in the Clouds system; processes may be considered the "glue" which binds object operations, and possibly actions, to do useful work.

A process is introduced by a header which gives the name of the process, as well as

²⁸The default timeout value, as well as action manager operations to alter the timeout value after an action is invoked, are described in Appendix E.

²⁹The exact details of the manner in which this information is provided depends on whether the operation is a local procedure or a publicly-visible operation of the object to which it belongs.

³⁰Note that all processes in the Clouds system are descendants of the top-level "universal action," which cannot be aborted.

clauses detailing any imports of object definitions necessary (see section 12.1):

<comp unit>	→	<process head> <process block> <process tail>
<process head>	→	<process hdr> <imports>
<process hdr>	→	“process” <process name> “is”
<process block>	→	[<process block decls pt>] <stmt pt>
<process block decls pt>	→	<process block decls> {<process block decls>}
<process block decls>	→	<const or var decl>
<process block decls>	→	<type decl>
<process block decls>	→	<constraint decl>
<process block decls>	→	<var address decl>
<process block decls>	→	<procedure decl>
<process tail>	→	“process” “.”

Following any import clauses, the body (<process block>) of the process is specified; the <stmt pt> of this block is the entry point when the process is activated, and execution begins there after any necessary variable initializations of the <process block> have been performed.

```

process test_bounded_stack is

import bounded_stack

bs1, bs2 : bounded_stack ()    ! Delayed object parameter constraint

i : integer := 0

begin
  bs1 := new bounded_stack ( 10, integer )
  bs2 := new bounded_stack ( 20, integer )
  loop
    if bs1 @ full () then
      exit .
    end if
    bs1 @ push (i)
    if (i % 3 = 0) and not (bs2 @ full () or bs1 @ empty ()) then
      bs2 @ push (bs1 @ pop ())
    end if
    i += 1
  end loop
end process. ! test_bounded_stack !

```

Example of a Process (see section 12.1)

15. References

- [Allc82] Allchin, J. E., and M. S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982
- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also available as technical report GIT-ICS-83/23)
- [LeBl85] LeBlanc, R. J., and C. T. Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)
- [McKe84a] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," School of Information and Computer Science, Georgia Institute of Technology, May 1984
- [McKe84b] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)
- [Pitt84] Pitts, D., "Storage Management for an Action-Based Operating System," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, November 1984 (also available as Technical Report GIT-ICS-85/02)
- [Wilk86] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, in progress

FINAL REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT #MDA 904-84-C-6035
REPORTING PERIOD: 11 Sept 84 - 30 Sept 85

TABLE OF CONTENTS

	Page
1. Summary of Work Done	1
2. Distributed File Systems	2
2.1. Storage Management Design for an Action-based Operating System	2
2.2. Storage Management	5
2.3. Recovery Management and Virtual Memory	5
2.4. Storage Management Implementation	7
2.4.1. References	10
3. Language Support for Robust Distributed Programs	11
3.1. The Design of Aeolus	11
3.1.1. Features for Systems Programming	11
3.1.2. Features for Object and Action Programming	13
3.2. Programming Methodology	15
3.3. Implementation	16
3.4. Design Refinements	18
3.5. References	18
4. Conclusions	19
Funds Expenditure Graph	21
Funds Expenditure Report	22

1. Summary of Work Done

During the course of this project, a variety of work has done on the two tasks called for by the statement of work. These efforts are closely related to other work in progress within the Clouds Project, a major research effort in the School of Information and Computer Science of Georgia Tech in the area of reliable distributed computing. We are now very close to integrating the results of a number of individual tasks supported under this and other projects to produce a first prototype of the Clouds distributed computing system.

Under the Distributed File Systems task, work has concentrated on the development of the storage management system of the Clouds kernel. Since the fundamental concept of the Clouds system is to support transparent, reliable access to arbitrary objects located anywhere with a network multicomputer, distributed file system issues must be addresses at a much lower level than is traditionally the case. The major achievements of this task are as follows:

- (1) initial design of the storage management system (see quarterly progress report 1 and Appendix A to that report)
- (2) refinements to that design for improved efficiency (see quarterly progress report 2)
- (3) design and implementation of low-level device drivers to support Clouds reliability mechanisms (see quarterly progress reports 3 and 4)
- (4) integration of storage with kernel virtual memory management (see quarterly progress report 4 and Appendix A to that report).

The technical report "Notes on a Storage Manager for the Clouds Kernel" (appendix A to quarterly progress report 4), thoroughly documents the results our work on the Clouds Storage Manager.

Under the Language Support for Robust Distributed Programs task, the focus of our work has been our language, Aeolus, and its intended uses. The most important aspect of Aeolus is that it provides a high-level language interface to the action and object management features of the Clouds Kernel. Thus it is intended to be used as a systems programming language for the implementation of the layers of the Clouds

system above the kernel. In support of this implementation work, we have been studying programming methodology issues involving the use of these unique capabilities provided by the kernel. The major achievements of our language work have been:

- (1) completion of the initial design of Aeolus (see quarterly progress report 1 and Appendix A to quarterly progress report 3)
- (2) substantial progress on implementation of an Aeolus compiler (see quarterly progress reports 2 and 3)
- (3) initial programming methodology studies (see quarterly progress report 1 and 2 and Appendices B and A to those reports, respectively)
- (4) definition of the interface to kernel action and object managers (see quarterly progress report 4)
- (5) refinements of the Aeolus design (see quarterly progress report 4 and Appendix B to that report).

2. Distributed File Systems

2.1. Storage Management Design for an Action-based Operating System

The Clouds Project is an effort to provide support for a distributed computing system which achieves performance improvements (over conventional computing systems) through the parallelism possible in a multi-computer environment and reliability improvements through the redundancy available in processing resources and data storage. In order to achieve such improvements, the system must ensure the proper coordination of processes on various machines in the system and synchronize the use of shared data. The system as a whole must be able to deal with failures of one of its component machines, determining those processes on the failed machine which are necessary for the continuation of some larger task. A reliable distributed system must be able to ensure the consistency of data in the presence of machine failures, taking into account that data may be replicated.

The initial goal of the Clouds project is to produce an operating system kernel that provides the mechanisms needed by a reliable distributed computing system. In providing these mechanisms, the Clouds kernel must support other conventional

mechanisms such as virtual memory, process control and secondary storage management. The action and object support must be integrated with the conventional kernel functions so that support for a reliable distributed computing system is available through a well-defined kernel interface, and the implementation of the kernel is efficient and compact.

One subtask currently in progress is the design and investigation of a portion of the Clouds kernel: the *storage management system*. A Ph.D. research proposal by Pitts (attached as Appendix A to quarterly Progress Report 1) described the initial plans for work on this subtask. In addition to describing how such a system can be built, it also defined the interaction of the storage management system with other parts of the kernel, particularly its interaction with the virtual memory system.

Thus the purpose of this research task has been to design a kernel-level storage management (file) system (*storage manager*) that supports a reliable distributed computing system. The storage manager is responsible for the secondary storage available on the system. Specifically, the design presents the structures and mechanisms necessary to support the storage manager. The design includes support for both recoverable and non-recoverable objects. Mechanisms to create, delete, write and read objects on disks are defined. For recoverable objects the additional protocols and structures ensure recoverability of objects in the presence of machine failures. The design also discusses the interaction of the storage manager and the virtual memory system. This portion of the design specifies the structures and mechanisms required for virtual memory. The design also defines the support required for action management and object recovery. Finally a facility for the location of segments on secondary storage must be provided.

The design of the storage manager is being done two phases. Phase one has been a design of essential features for the system. The end-result will be an implementation for the Clouds kernel that will serve as a test-bed for further research. An analysis of the design and implementation will be done to determine the correctness and effectiveness of the design. The results of the analysis may have an effect on phase two. This phase of the design will include modifications and refinements to the original design. In general, phase two will include features not absolutely necessary for the storage manager, but which may be desirable later as the system is put to use as a research device. Feedback from the analysis of the original design may suggest

some of the changes found in the second phase. Phase two is not intended for immediate implementation.

The Clouds kernel will provide support for three basic mechanisms which will be important to later discussion: *processes*, *objects* and *actions*. Processes are the active agents of the system; to initiate and perform any work requires a process. The kernel has a process manager which handles all bookkeeping associated with creating, dispatching, and destroying processes.

Objects, on the other hand, are passive entities. Objects are typed collections of data. The type of an object determines what operations may be performed on the data, as well as how the data is organized. Object data can only be manipulated through these operations, and then only by a process which has a proper *capability* for the object. A capability is a unique name for an object along with a list of operations which are permissible for use by the possessor. The object manager handles the overhead of verifying capabilities and performing operation calls.

Objects are the organizational units of the system. By using objects, a programmer has a means for abstraction and isolation of data. The kernel also provides a mechanism for organizing sets of operations into a unit. This mechanism is the action. Actions are *atomic*. The set of operations comprising an action appears to execute completely (by *committing* its results) or not at all (by *aborting*). Also, the atomicity of actions prevents the execution of one action interfering with the execution of another. Actions provide a mechanism for making the effects of a set of operations consistent and recoverable.

Actions are managed by the object manager. Actions themselves are simply organizational units of work and require processes in order to perform any task. An action may have several processes or one process executing on its behalf.

The kernel provides processes, objects, and actions as efficiently as possible. Particularly, because objects have different types and different possible operations, the kernel needs access to objects in a manner which is consistent and convenient. For this reason, all objects have a secondary type, called the *segment* type. The segment type is a sequence of bytes with primitive operations such as read a page, write a page, and delete or add a page. The segment is accessible only by the kernel.

2.2. Storage Management

The Clouds secondary storage is managed as a set of *partitions*. Each partition is an autonomous logical device with its own set of interface routines for the transfer of information and the allocation of the secondary storage managed by the partition. A partition resides completely on one physical device and consists of a contiguous set of records on disk. The partition requires three structures to manage partition storage. First is a partition header, which holds information concerning the partition such as its size, whether it provides support for recoverability, a list of bad disk records for the partition, and other such information. The header should be duplicated to reduce the risk of its destruction by a media failure or other such disastrous error. The header is placed at a known location in the partition.

Each partition also maintains a directory, contains a mapping of sysnames (for objects) onto partition record addresses. Note that a partition directory contains mappings only for objects residing on that partition. Redundancy should also be insured for this structure. The partition directory is at a well-known location.

The third partition structure is a record map, which is a bit-map showing allocation of records for the partition. The driver uses the record map to determine which records are in use by segments and which can be allocated. Once again, the record map is an important structure which should be duplicated to prevent its loss after a media crash.

The remainder of the partition is available for the storage of object data, or as the storage manager treats objects, segment data.

Before a partition can be accessed by the kernel, it must be mounted on the system. This involves doing a consistency check on the partition storage, examining the directory and record map, and cleaning up any loose ends as far as recovery management is concerned. Of course, the physical device on which the partition resides must be active prior to this processing.

2.3. Recovery Management and Virtual Memory

Segment recovery is accomplished via a shadowing scheme. That is, segments on which actions are operating have shadow versions that the actions actually see. The

scheme is pessimistic, so that no modifications are made to a permanent version until the action making the modifications commits. The goals of the recovery scheme are, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few disks accesses as possible. Shadowing, then, will be minimal. That is, only those parts of the segment actually modified are shadowed.

The storage manager becomes involved in recovery only when a top-level action precommits and the shadow version of the segment on which the action is operating is created. Prior to precommit, all write operations are done in memory. An active segment is mapped into memory by the virtual memory system. An object's address space contains a block of permanent data and a block of volatile data. The permanent data block contains data which will survive a crash. This is basically the permanent object state. The volatile data block's contents will not survive a machine crash and generally consists of such structures as locks and semaphores for the object. Also contained in the volatile data block is much of the information maintained by the action management system.

When an action operates on a segment, the action management system maintains in the volatile data block versions of any modified recoverable parts of the segment. There may be any number of versions due to the nesting of actions and actions sharing the segment. When a top-level action precommits, data must be moved from the volatile data block to the permanent data block prior to shadowing the segment on secondary storage. To simplify the precommit procedure, we allow only one action per segment to pass the precommit point. For example, if actions A and B are both operating on object O and A precommits, B is prevented from precommitting. If B attempts to precommit, the action management system blocks the action. B still may access the object.

During the time precommit and commit are taking place, the virtual memory system must insure that modified pages of the permanent data block remain in memory and undisturbed. The virtual memory system can do this by physically locking the pages in memory, making them read-only. Then the pages can be flushed to disk to build the shadow version of the permanent segment.

The mapping of virtual memory to secondary storage is another of the storage manager's responsibilities. On page faults, the virtual memory system makes use of storage manager calls to locate the backing storage for faulted pages and also to allocate or locate backing storage for virtual pages being paged out.

2.4. Storage Management Implementation

The development of the Clouds storage manager involves the implementation of three components. These are the device object, the partition object, and the segment object. These objects are abstractions of the disk storage available on a Clouds machine. The device object manipulates device storage as a collection of uninterpreted blocks of data, which it transfers in and out of virtual memory. The partition object provides a mechanism for division of device storage for administrative purposes and also is involved in the location of data and the allocation of device storage. The segment object treats device storage as a collection of bytes. In fact, the segment object is just an alternate view of any Clouds object. We consider the device object the lowest level of abstraction and the segment object represents the highest level. In the paragraphs that follow, we describe the current state of the storage manager.

At the device object level, we are developing two disk objects. Clouds disk objects include not only the conventional device driver functions, but also provide necessary support for the recovery mechanisms of the storage manager. Specifically, the Clouds disk objects provide a mechanism, the *flush routine*, which insures that requests scheduled by an action are actually completed before the action commits. This mechanism differs from conventional disk management schemes, where a request may remain enqueued after the process that issued it terminates. The flush routine relies on the *flush table*, a *per device* structure. The table contains an entry for each action; the entry contains a list of requests made by the action and a record of the number of requests pending and completed.

The development of a RL02 disk object has been straightforward and we now have a working version integrated with the Clouds kernel. Minor changes in the way the object formats the medium are anticipated. Additionally, the object must be modified to lock physical pages for I/O transfers, because of the Clouds kernel's use of the virtual memory system as the basic I/O mechanism. The RL02 will allow us to go ahead with the development of kernel and particularly with the testing of the

storage manager. The RL02 will not be the primary disk for the Clouds system, as it holds only 10 Mb on each cartridge. The primary disk for the initial Clouds implementation will be the RA81, a disk object for which is under development in parallel with the development of the RL02 object. Because the RA81 is a "smart" device, progress has been slower and the integration of the facilities required by the Clouds storage manager is more complex. Testing is currently under way on this device. We have kept the device object interface for the two devices uniform and also have attempted to reduce any side-effects so that upon completion of the RA81 object, this object can be use in the place of or along with the RL02.

The next level of abstraction for the storage manager is the partition level, which we have already discussed in some detail. Implementation at this level is just being completed. A partition provides all the structures required to support the creation and management of Clouds objects. Specifically, the partitions provide support for the location of objects and the allocation of disk storage for objects. We have presented the partition structures necessary for these functions, namely the partition directory and record map. There is another structure which, while not necessary for the functionality of the partition object, we believe will considerably improve the efficiency of the partition object's performance.

To understand the significance of this structure note that the Clouds kernel provides for the location-independent invocation of an object operation, which requires the kernel to search for the object at each operation invocation. Object searches are network-wide and several techniques are being developed to short-circuit these searches. One concern is the necessity of going to disk in order to determine if the object is on a partition. Each partition maintains a structure called a *maybe table* which is intended to reduce the number of unnecessary disk accesses during object search (an unnecessary access is one which shows the object is not on the partition in question). The maybe table is a small in-memory representation of the partition membership. A maybe table is an example of a Bloom filter [Bloom70]. The table is a compressed hash table, in which several segment names may hash to the same entry. In trade-off for the reduced size of the table, only a negative response to a query is guaranteed to be correct. Responses indicating the object is on the partition may in fact be wrong and may require the partition object to access the directory on disk. We are really trading accuracy of the responses for speed since in most cases the query can

be answered without an unnecessary access to disk. Because searches are frequent events, we feel that the Maybe table will have a large effect on system performance.

The segment system forms the highest abstraction provided by the storage manager. Disk storage at this level is managed as a collection of arbitrarily sized segments, which generally represent some Clouds object. Segments provide a uniform interface through which the Clouds kernel can manipulate the object. Although segments conceptually provide a simple view of storage as a sequence of bytes, the actual implementation on disk is quite different. A disk segment is a tree structure, which has as its root a *segment header*. This header contains all information pertinent to the segment and is the entry into the segment (an entry in the partition directory points to the segment header). The leaves of the segment tree are the data records on the segment. The internal nodes of the tree consist of link records, providing connectivity between the segment header and the data records.

In addition, the segment system provides the recovery protocols discussed earlier. Implementation of the segment system is currently in progress.

As discussed above, the segment system, the action management system and object management system are involved in the management of virtual memory with respect to the mapping of objects. We are finalizing the extent of each system's responsibilities and influence in the virtual memory and the cooperation needed between the systems. For example, the storage manager shares with the object manager the responsibility for mapping the on-disk version of the segment to the virtual memory version. Each segment has one or more windows which represent chunks of the segments which are actually mapped into virtual memory. This allows the mapping of portions of large objects into virtual memory, avoiding the cost of mapping the entire object. The mapping of each object in memory will consist of several standard windows: a code window for the executable portion of the object; a permanent object data window; and a volatile data window. If the object is recoverable the permanent data window actually consists of several optional windows. There may be a static non-recoverable data window, containing object data not considered necessary as part of the recoverable object state. There may also be a static recoverable data window, which contains (part of) the recoverable object state. Finally, there may be a permanent heap window, which is used in objects which provide customized recovery [Wilkes85a, Wilkes85b]. Each of these windows may be mapped to different

partitions. For instance, the code window and static non-recoverable windows are mapped onto the disk segment image itself. The volatile heap window and recoverable windows are mapped to a paging partition, a special partition reserved simply for providing backing store. In the latter case, the storage manager is really providing two sets of mappings from disk to virtual memory: one for handling page faults and the other for handling the recovery aspects of object management.

The storage manager also makes use of the virtual memory system to assist the action management system in the committing of actions. The segment system uses virtual memory structures to determine which segment pages have been modified and then, based on its own information as to the structure of the segment, decides which segment pages must be shadowed to provide the necessary recovery. Because the storage manager is aware of the segment's virtual memory mapping, and the special attributes of the standard windows, it knows which modified pages actually need to be shadowed and which can be simply written to disk.

A technical report [Pitts85] which summarizes all of the work which has been done on the storage manager was attached to Quarterly Progress Report 4 as Appendix A.

2.4.1. References

[Bloom70] Bloom, B.H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, No. 13, Vol. 7 (July 1970), pp.422-426.

[Pitts85] Pitts, D.V. and E.H. Spafford, "Notes on a Storage Manager for the Clouds Kernel" Technical Report GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, October 1985.

[Wilkes85a]

Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985.

[Wilkes85b]

Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report

3. Language Support for Robust Distributed Programs

3.1. The Design of Aeolus

The major design goal of our language Aeolus is to make possible access to the features of the Clouds system from a powerful systems programming language which supplies those features—such as strong typing—which aid in the quick development of error-free programs, yet allows those features to be explicitly circumvented when necessary.

The major structuring features in Aeolus are processes and objects. Objects have two purposes in Aeolus: to provide support for data abstraction, and to reflect the recoverability and synchronization capabilities provided by the Clouds kernel. It has been argued elsewhere [Allc82] that the object construct provides a powerful tool for the organization of programs for recovery, both from the standpoint of the programmer and of the system. Objects may rely on the automatic operating system / run-time system support for synchronization and recovery (*autorecoverable* and *autosynch* objects). Alternatively, using powerful features provided by the language and the Clouds system, the programmer may take advantage of semantic knowledge about the application to explicitly code more appropriate recoverability and synchronization. However, Aeolus objects also provide abstraction features even when synchronization and recovery are not required. These *non-Clouds* objects provide a logical framework for the organization of modules for separate compilation.

3.1.1. Features for Systems Programming

In keeping with its purpose as a systems programming language, Aeolus incorporates several features which give the programmer access to the hardware and the lower levels of the systems software, as well as “convenience” features which allow more efficient coding, including:

- a full range of assignment and bit-manipulation operators similar to those in the C language;

- features for register optimization, such as a special *index* type for loop counters

- and array references;
- the option of specifying *inline* expansion of a procedure;
- a facility for specifying *arbitrary* procedure argument lists of unspecified length and (predefined) types (similar to the *nospread* arglists of Interlisp);
- and the ability to specify storage addresses for variables, as well as some capabilities for setting and doing arithmetic on pointers.

However, most of the power of Aeolus as a systems programming language, aside from the access it provides to the features of the Clouds system, lies in the ability it gives the programmer to specify low-level data structures as abstract data types, and in the treatment of the underlying hardware as an object with operations on its state available from the language.

In addition to the usual structured types (records and arrays), Aeolus provides a *structure* type, which allows the programmer to specify abstract types for the manipulation of bitfields. The *structure* is similar to the *packed record* construct of Pascal, except that the programmer indicates that its fields should fit one of the addressable entities defined by the target computer (byte, word, doubleword, quadword, etc.), and this correspondence is checked by the compiler. This provides a secure mechanism allowing bit fields within a low-level data structure to be referenced by name. Aeolus also provides the *byte* and *word* types as predefined objects. These objects have operations permitting manipulations similar to those of the *bitset* type of Modula-2. The programmer may define similar objects for bit strings of other lengths.

The ability to inspect and change the state of the hardware is also important in systems programming. Access to the underlying hardware is provided by the operations of special Aeolus objects. We call such an object a *pseudo-object* since only one instance of it may exist, whereas there may be an arbitrary number of instances of a normal object. An example of a pseudo-object is `PC_System`. This pseudo-object gives access to the registers and ports of a PC's microprocessor, and through the ports to the other system components, such as the interrupt controller, device controllers, and modem registers. For example, the `IN_BYTE` and `OUT_BYTE` operations of `PC_System` allow values to be input and output from the byte ports of a PC; other `PC_System` operations provide such capabilities as access to the register set, flags, and interrupt mechanism. These operations typically compile inline to a single machine instruction. For considerations of efficiency, some operations in hardware pseudo-

objects may give access to special instructions of the target machine, such as the string instructions of the PC or the polynomial instructions of the VAX.

3.1.2. Features for Object and Action Programming

The design of Aeolus is intended to support the recovery and synchronization capabilities of the Clouds system in a high-level systems programming language. Objects in Aeolus, besides providing an organizational tool for secure separate compilation, give access to the recovery properties of Clouds objects. Thus, if an Aeolus object is designated as *recoverable* or *autorecoverable*, the Clouds kernel mechanisms are used for invocations of its operations, allowing the system to control the recoverability properties of the object's state. In the remainder of this section, the features provided by Aeolus for accessing these features of Clouds are examined.

The code for an Aeolus object has two parts. The *definition* part is seen both by the object itself when it is being compiled, and by all other objects or programs which use that object. Compilation of a definition part produces a symbol table file which is used for type checking among these separate compilations. It can contain specifications of public types and constants defined by the object, and the interface definitions of the object's operations. Definition parts may not contain variable declarations. The *implementation* part contains the actual code of the operations, along with any needed local (private) type, constant, or procedure definitions. Local variables of an object share the lifetime of the object instance to which they belong, and thus act as "own" variables. This separation of definition and implementation provides a safe separate compilation mechanism similar to *packages* in Ada (TM) or *modules* in Modula-2.

In the header of an object definition, the programmer may specify the object *class* as being *pseudo*, *local*, *nonrecoverable*, *recoverable*, or *autorecoverable*. The classes *pseudo* and *local* are called the *non-Clouds* object classes; the classes *nonrecoverable*, *recoverable*, and *autorecoverable* are called the *Clouds* classes. If the object class is specified as being *pseudo*, the object is treated as being simply a separate compilation module; pseudo-objects are used as a simple separate compilation mechanism, as interfacing to the runtime system and kernel services, and for integrating objects written in other languages into Aeolus systems. *Local* objects provide some of the functionality of Clouds objects (such as access to multiple object instances) without the expense of

the Clouds object management facilities; however, local objects have no existence independent of a process, as Clouds objects may have. The simplest Clouds object class, *nonrecoverable*, makes use of the Clouds object management facilities but does not use the action management. Objects of the *autorecoverable* class, however, provide fully automatic access to the Clouds action management facilities. The entire object state is made recoverable, and default handlers for the *action events* (such as ABORT and COMMIT) are provided by the compiler/runtime system. Thus, the programmer may gain access to the action mechanisms of the Clouds system with a single keyword. However, the full power of the automatic Clouds action mechanisms may be unnecessary and inefficient in some cases. For those cases, the Aeolus/Clouds system provides mechanisms which allow the user to explicitly program recovery strategies tailored to the individual requirements of the problem at hand. If the object is specified as being *recoverable*, the programmer may specify part of the object state as being recoverable and may provide alternate handlers for action events.

The Aeolus language also provides access to the synchronization mechanisms of the Clouds system. When the *autosynch* object attribute is specified in an object definition header, it indicates that the default system synchronization procedures are to be used on the object's operations to provide concurrency atomicity. If the *autosynch* attribute is not specified, synchronization may be explicitly programmed using operations on the *lock* type provided by the language. A Clouds lock [Allc83b] is not associated with a physical object, but rather with values in the domain of the object. Thus—for example—a file name may be locked, even if a physical file with that name does not yet exist.

Object operations are programmed like procedures. An operation invocation looks like a procedure invocation with a prefix indicating the object instance upon which to operate:

<object instance id> @ <operation id> (<actual param list>)

An object instance may be created by declaring a variable of that object type, and then allocating the instance's data storage on the heap using an extended version of the allocation function, or by associating the variable with a "permanent" object, much as a file variable can be associated with a physical file in Pascal.

An operation or local procedure of a recoverable or autorecoverable Aeolus object may be invoked as an action. The invocation of an action is similar to a procedure or operation invocation; however, a unique *action-id* is created by a Clouds action manager for the invocation, which may be assigned to a variable of the invoking procedure, for example:

```
actionID := action ( object1 @ op1 ( param1, param2 ) )
```

This *action-id* variable may be used to retrieve information from the system about the status of the action, or to abort the action, using calls to a Clouds action manager. This mechanism allows general control structures to be formulated, e.g., for the concurrent invocation of actions.

3.2. Programming Methodology

The features of Aeolus described above (and in [Wilk85b]) provide easy access to the synchronization and recovery features of Clouds, and thus they provide a framework within which to study programming methodologies suitable for action-object systems such as Clouds. This study should lead to the design of high-level language features to support that methodology. Thus, our interest in Aeolus lies not so much in the language itself as in studying the sort of programming which may be done with it.

We have found Aeolus to be effective as a systems programming language during our studies of programming systems objects such as communications handlers for the Clouds workstations. In particular, the clarity of interface definitions made possible by use of pseudo-objects is extremely valuable for encapsulation of hardware details in such hardware-dependent programming. Through our experience with developing systems objects, we have come to understand techniques for using subactions as "firewalls" to limit the effect of failures. We have found that Allchin's generalized lock mechanism makes it relatively easy to specify special-purpose synchronization rules dependent on object semantics.

We intend to use Aeolus as a framework within which to study programming methodologies for action-object systems. Among the hardest questions which need more study is how replication can most effectively be used to provide availability.

Actions and resilient objects ensure that failures are not catastrophic, but they are concerned with data integrity, not with how a program reacts to failures. The availability question involves use of multiple objects on different nodes to represent a single resource, thus providing continued access to the resource in the presence of individual node failures. Algorithms for read and write access to such resources must be developed and evaluated. The recent paper by Daniels and Spector [Dani83] is one example of such an algorithm.

We must also consider possible representations of work so that it may be restarted; this is an area that has been until recently unexplored [McKe84]. Most of the work on actions and objects has been oriented toward protection of data from failures. The fact that processes are considered to be an important, independent component supported by the Clouds architecture gives us a point of departure for this study. McKendry's work on Petri nets [McKe84] lays the groundwork for an attack on this problem within the framework of Clouds. If we view a program as a collection of processes interacting through shared objects, some features akin to the process interconnection specifications of Pronet [Macc82] may prove to be useful.

Our initial studies in programming methodologies for resilience and availability are described in [Wilk85a]; there, a plan is presented for determining such methodologies appropriate to the design of objects needed in the Clouds system. Examples of a replicated object exhibiting the properties of resilience and availability are given there, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the Aeolus/Clouds system to provide these properties. The issues with which we are concerned include the use of semantic knowledge of objects in the programming of replication; trade-offs between consistency and availability; the appropriateness of current programming models for replicated data; and the support needed from the operating system and language runtime system to ensure availability and forward progress of processes. As we progress with these studies, we will take advantage of our experience in the implementation of the Aeolus runtime system and its interaction with the action and object managers of the Clouds system.

3.3. Implementation

A compiler for Aeolus is currently under development on one of the DEC VAX 11/750 computers of the Clouds project under Berkeley Unix (TM) Version 4.2. We

are using the *Amsterdam Compiler Kit* (ACK) [Tane83] to generate code generators for Aeolus for both the Clouds VAXes and the individual work stations which the Clouds system will use to interface to the VAXes. Work on the semantic routines for Aeolus is proceeding in parallel with the development of routines to generate intermediate code for ACK. This work is being done in *Pastel*, an extended Pascal dialect developed at the Lawrence Livermore National Laboratory.

The code-generation work is progressing quite well; we have been able to generate and execute code for object invocations which do not involve the facilities of the Clouds kernel or object managers (that is, code for what we call "non-Clouds objects").

Work is also progressing on the implementation of facilities for generating actual "Clouds objects." This entails the definition of the interface to the Clouds object and action managers, which will serve as an intermediary between user programs and the kernel facilities. Thus, the members of the compiler group are working with members of the kernel group on the definition and implementation of the action and object managers. These interfaces are now well-defined, and we expect the Aeolus compiler to be capable of interfacing with the action and object managers of the Clouds system, and thus to be capable of invocations on actual Clouds objects, by the end of 1985. Actual testing of object and action management calls awaits the implementation of the requisite Clouds system services, which is expected in the first quarter of 1986.

As mentioned above, the design of the interfaces of the runtime system with the Clouds action and object managers is essentially complete. Members of the Aeolus group have been working with members of the kernel group to design these interfaces and strategies for efficient action management. Of particular importance are our designs for support of recoverable areas in Clouds objects; these constructs enable the Aeolus language (in conjunction with the action management system) to provide *view atomicity* in addition to the *failure atomicity* provided by the kernel. Each action which touches an object which has a recoverable area gets its own copy (or *version*) of that recoverable area on which it may make its changes; when a nested action commits, it propagates its version of the recoverable area to its parent. View atomicity ensures that each action in the action tree which accesses an object sees the correct version of the data in that recoverable area. We have developed a technique for implementing recoverable areas using partial replacement of the object page table entries which

provides view atomicity without causing a time penalty for access to the data in the recoverable area. Rather, a small penalty is paid upon process exchange if a process is associated with an action.

3.4. Design Refinements

The work of the Aeolus group during recent months has been concentrated on the refinement and rationalization of the language design. The design of the language has undergone a significant reworking, especially those parts of the design concerned with specification of types. In view of one of the Aeolus design goals of providing the power necessary for systems programming without sacrificing the advantages of strong type checking, we wished to provide dynamic (flexible) data types; however, we felt that our previous design for this violated the goals of simplicity and readability. Our reworked design integrates flexible types into the language in a much cleaner manner, within the framework of general parameterized types. The changes to the design have been incorporated into the language description [Wilk85b], along with the interfaces to the Clouds system object and action managers. The revised language definition was attached to Quarterly Progress Report 4 as Appendix B.

These design changes have now been incorporated into the symbol table of the compiler and new semantic routines necessitated by the changes are being implemented. We have also taken advantage of the redesign to streamline parts of the semantic routine structure, taking into account our previous implementation experience. Work on the implementation is accelerating now that these changes are understood.

3.5. References

- [Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983.
- [Macc82] Maccabe, A. B., and R. J. LeBlanc, "The Design of a Programming Language Based On Connectivity Networks," *Proceedings of the Third International Conference on Distributed Computing Systems*, Miami / Fort Lauderdale, October 1982.

- [McKe84] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05).
- [Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM* 26, 9, September 1983.
- [Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985.
- [Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, July 1985 (revised October 1985).

4. Conclusions

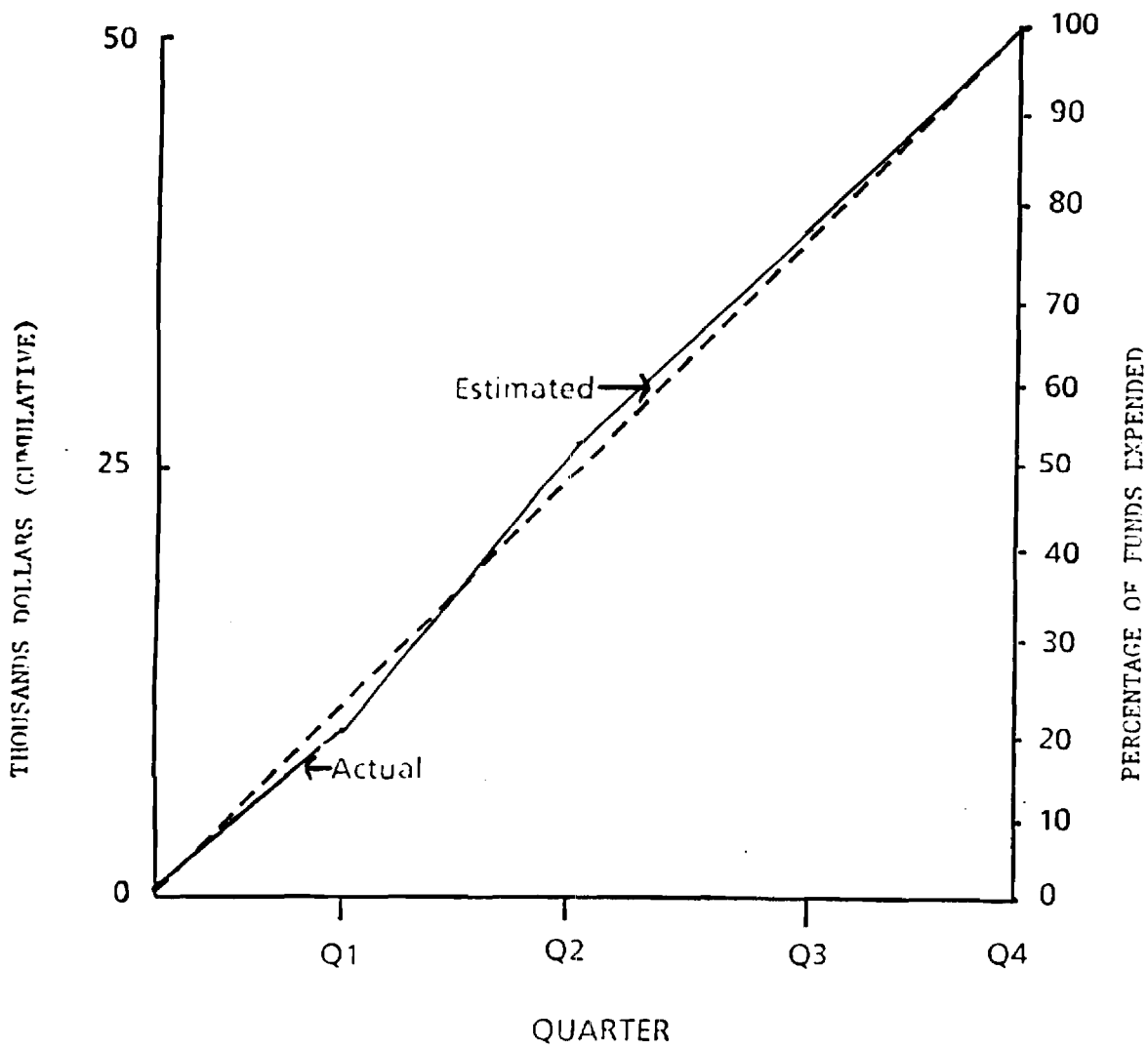
Work on both of the two tasks of the project has been very productive over the last year. In conjunction with related work on the Clouds kernel and the action management system, we anticipate having a working Clouds prototype by early in 1986. The storage manager designed and currently being implemented under the Distributed File Systems task obviously is strongly related to the kernel and thus must be combined with the kernel for testing. That integration should take place in the immediate future. The next phase of this work, which includes performance measurement and analysis followed by design refinement, will then be possible.

Under the Language Support for Robust Distributed Computing task, we have produced a language (Aeolus) which includes features that match the capabilities for action and object management provided by the Clouds kernel. This language has enabled us to begin our studies of programming methodologies for action/object programming. Further study in this area will be a major ongoing research effort. We have also made substantial progress on a compiler for Aeolus. It will be ready for use by the time the kernel and storage manager are integrated and available for managing Aeolus objects and processes. The compiler interface to the action management system has been defined and the compiler should be capable of generating code to utilize

this interface even prior to the implementation of the action manager.

In summary, our efforts under this project have been instrumental in the development of a prototype of the Clouds reliable distributed computing system concept. Work to be done in the next year will provide significant evidence of the viability of the Clouds concept.

FUNDS EXPENDITURE GRAPH



FUNDS EXPENDITURE REPORT

Column A ----- ORIGINAL PROPOSAL	Column B ----- Latest Accepted Revised Proposal	Column C ----- Reporting Quarter Expenditures	Column D ----- Cumulative Expenditures to Date Total Man Hours Dollar Value Pct. Dollar Value	Column E ----- Cost to Complete Estimate	Column F ----- Latest Cost Estimate
Direct Labor					
Type	Number of Hours	Hourly Rate	Dollar Total	Number of Hours	Dollar Total
-----	-----	-----	-----	-----	-----
DI	350	23.77	\$8320.00	430	\$10220.96
BRA	1300	11.41	\$14833.00	1648	\$18802.35
Clerical	175	6.74	\$1180.00	0	\$0.00
			-----	-----	-----
Total Direct Labor			\$24333.00		\$29023.31
Burden @ 24.6%			\$2337.00		\$2387.93
(21.0% starting 7/1/85)					
(Excluding BRA Labor)					
			-----	-----	-----
Total Direct Labor and Burden			\$26670.00		\$31411.24
TRAVEL EXPENSE			\$2500.00		\$1193.74
GENERAL & ADMINISTRATIVE EXPENSE			\$1500.00		\$413.46
COMPUTING CHARGES			\$1500.00		\$1565.18
			-----	-----	-----
TOTAL DIRECT COSTS			\$32170.00		\$34583.62
DIRECT COSTS @ 55.3%			\$17790.00		\$19959.38
(63.5% starting 7/1/85)					
			-----	-----	-----
CONTRACT PRICE			\$49960.00		\$54543.00
COMMITMENTS AND EXPENDITURES				\$17144.20	\$54543.00

Programming Methodologies for Resilience and Availability

C. Thomas Wilkes

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

The goal of the *Clouds* project at Georgia Tech is the implementation of a fault-tolerant distributed operating system based on the notions of *objects* and *actions*, which will provide an environment for the construction of reliable applications. As part of the *Clouds* project, we are designing and implementing a high-level language in which those levels of the *Clouds* system above the kernel level will be implemented. The *Aeolus* language provides access to the synchronization and recovery features of *Clouds*. It also provides a framework with which to study programming methodologies suitable for action-object systems such as *Clouds*.

This proposal describes a plan for determining programming methodologies appropriate to the design of objects needed in the *Clouds* system. Among the properties needed by these objects are *resilience* and *availability*. Examples of a replicated object exhibiting these properties are given, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the *Aeolus/Clouds* system to provide these properties.

1. INTRODUCTION

Among the benefits promised by advocates of distributed computing are improvements in system fault tolerance and reliability, increased availability of data and services, and faster response through use of distributed programs. Interest in reliability has grown as distributed systems have come to be used in an ever widening set of applications, including critical control systems. In the past, fault tolerance has principally been the concern of hardware designers, who mainly used redundancy as a solution. More recently, it has been realized that maintaining the integrity of distributed data is a crucial concern in providing the benefits listed above. Accordingly, there has been a growth in research interest in techniques for providing the required data integrity in the presence of hardware failures and concurrently executing processes.

The *Clouds* project at Georgia Tech [Allc82, Allc83a, Allc83b] is one of a number of recent proposals in which reliability in a distributed system is based on the use of *atomic actions*, a generalization of the transaction concept of distributed databases. As part of the *Clouds* project, we are designing and implementing a high-level language which will provide access to the synchronization and recovery features of the *Clouds* system; this language will be used to implement those levels of the *Clouds* system above the kernel level. It will also provide a framework within which to study programming methodologies suitable for systems based on the action concept, such as *Clouds*. Among the properties needed by systems data structures, the design of which must be addressed by such methodologies, are *resilience* -- survivability and consistency of the data despite crashes and other faults -- and *availability* -- increased possibility of access to replicated data despite network partitions or failures of some sites in a multicomputer system.

This proposal describes a plan for studying such methodologies, in particular those needed in the design of the type of data structures necessary for the implementation of the *Clouds* system. Section 2 of this proposal presents the problem explored by this work and describes the environment in which it will be examined (the *Clouds* system). Section 3 describes the plan of the research to be performed, and presents examples of a replicated object exhibiting the properties of *resilience* and *availability*, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the *Aeolus/Clouds* system to provide these properties. An outline of the proposed dissertation is presented in Section 4.

2. PROBLEM STATEMENT AND BACKGROUND

The purpose of the proposed research is to explore programming methodologies for action-based systems appropriate to the design of data structures exhibiting the properties of resilience and availability. In this section, the environment in which this study will be carried out is described, and a rationale for the need of such programming methodologies is presented.

2.1. The Clouds System

The goal of the Clouds project at Georgia Tech is to allow the construction of reliable application systems on unreliable hardware. We use the notion of an *object* to represent system components, such as directories or queues. An object consists of data and a set of operations on that data which maintain a set of associated invariants. A set of changes to objects is grouped into an action, a unit of work which appears to be *atomic* to its environment (in particular, to other actions). Objects are passive in the Clouds architecture; thus, *processes*, which may represent a single top-level or nested action, are used to provide activity in the system.

The actions in Clouds go beyond the related notion of transactions in a database system. Rather than modelling all access to objects as simple reads or writes, the Clouds approach supports arbitrary operations on objects and allows a programmer to take advantage of operation semantics to increase concurrency, and thereby performance. Through appropriate use of encapsulation, concurrent actions can be allowed to change objects without breaching serializability. Even breaches to serializability can be allowed, when it is semantically appropriate and it is necessary to improve performance.

Thus, objects, actions and processes are fundamental concepts supported by the architecture. To support these concepts, recovery and consistency are incorporated into the basic virtual memory mechanism. Synchronization mechanisms to control the interactions of actions are also provided. It is with these capabilities that Clouds is meant to support the data integrity required for the implementation of reliable, distributed application programs.

The mechanisms developed for the support of transactions in database systems, as well as the traditional operating system synchronization mechanisms, have been found to be insufficient for the support of the action-object approach in operating systems. In particular, the problems of ordering and atomicity for nested actions, and several simplifications which apply to many operating systems problems, are discussed in [McKe84b]. In particular, it is shown that through the use of *per-action variables*, it is not necessary to maintain complete versions of recoverable data for many types of systems data structures, giving substantial gains in space and time performance. The expediciencies made possible by these simplifications make the use of the action-object approach in the Clouds system viable.

These mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects (*servers*) which in combination provide a reliable environment for applications.

2.2. The Need for an Action-Based Programming Methodology

Actions are the key feature for guaranteeing data consistency. The "all-or-nothing" nature of actions really solves two problems. When an action fails, its effects are automatically undone; so, actions which fail due to machine failures cannot leave objects in an inconsistent state. Additionally, the required serializability of actions provides a coarse-grained synchronization among them. (Other features may be used to provide more concurrency by supporting synchronization at a lower level.) Actions which are aborted for logical reasons (e.g., deadlock) again can have no visible effects on the state of any object. Thus the action concept successfully broadens the recovery viewpoint provided by checkpoints, since it encompasses all the changes to any number of objects made by an arbitrarily complex action.

Actions alone do not provide all of the generally desired capabilities, since they do not address the question of the resiliency of individual objects. That is, they do not contribute toward the recovery of objects

located on machines that fail. Rather, they guarantee the integrity of surviving objects. Both Argus and Clouds support resilience through use of *stable storage*. (Stable storage has the property that information entrusted to it is extremely unlikely to be lost.) Various features are provided which cause the object support system to record sufficient information on stable storage so that the state of an object (*guardian* in Argus) may be recovered after a hardware failure. Note that for a combination of consistency and resilience, the state of an object must be written to stable storage whenever an action which modified the object commits (presuming that pessimistic recovery is being used).

Writing the state on an object to stable storage is, of course, just checkpointing. It is the coupling with the action mechanism which makes checkpointing of objects effective. That is, part of the implementation of a commit is a checkpoint of all affected objects. Thus checkpointing is made an effective means for providing consistent, resilient objects.

The mechanism for specifying just what must be written is one way in which Argus, Clouds and other proposals differ. In Argus, all *mutex* objects within a guardian are written. As suggested by the name, *mutex* objects also have certain synchronization properties, relating to their accessibility to concurrently executing actions. Clouds, on the other hand, allows an entire object or any data object within it to be specified as *recoverable*. As would be expected, if the entire object is recoverable, then all of its contained data objects are written to stable storage when a relevant commit occurs. Both of these approaches exemplify implicit specifications of what must be saved for recovery purposes. Yet another approach would be to require the programmer who defines an object to provide an explicit *write-to-stable-storage* operation to be invoked by the object management system at appropriate times. This variety of proposals reflects the need for study of a programming methodology based on use of objects and actions, so that we can determine just what kinds of features are most effective.

The Clouds architecture goes beyond others in that it can support actions that involve objects on more than one machine. In other words, a remote procedure call can be done without creation of a nested action. Allchin's work [Allc83b] provides a definition of the basic capabilities supported by the Clouds architecture and a design for their implementation. Now that that implementation is in progress, we are studying how these capabilities may be applied. In particular, we wish to study a programming methodology for systems like Clouds. The semantic knowledge about objects afforded by the encapsulation in object-oriented systems like Clouds should provide opportunities for optimizations in the treatment of the synchronization and recovery of replicated data unavailable in the simple read-write database model.

As part of the Clouds project, we are designing and implementing high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology). An overview of the *Aeolus* language is presented in [LeB185] (most of the materials in this and the preceding sections are also derived from this paper). *Aeolus* gives the programmer access to the features of the Clouds system discussed above. However, we also intend to use *Aeolus* as a framework for studying the sort of programming methodology appropriate to Clouds. This study should lead to the design of high-level language features to support that methodology. Additionally, it should suggest what capabilities are desirable both in the *Aeolus* run-time system and in the underlying action and object management support of the Clouds system.

3. THE PROPOSED RESEARCH

In this section, a plan for the proposed research is presented, as well as some preliminary examples of the sort of studies which will be made.

3.1. Plan of Research

The investigation of programming methodologies appropriate to systems objects having the properties of resilience and availability necessitates the programming of such objects. In the course of our studies, we propose to design simple versions of fault-tolerant servers which will be useful in the Clouds system, such as name servers, directory managers, system queues, etc. These designs may not be complete, since the aim is to study the issues of availability and resilience in terms of these data structures, rather than such issues as

naming in distributed systems. Complete designs of these objects will be left for those researchers studying the relevant issues within the environment of the Clouds system.

Among the issues which will be treated during this investigation are:

Use of semantic knowledge of objects in programming of replication

Can significant savings be achieved in programming the synchronization and recovery of replicated data by taking advantage of semantic knowledge of complex objects rather than using a simple read-write model?

Trade-offs between consistency and availability

Is it necessary, in programming replicated system services, to maintain strict consistency (serializability) among replicates in all cases, or are there cases in which global consistency may be weakened in favor of availability of services?

Blocking (pessimistic) vs. non-blocking (optimistic) replication methods

Again, there are trade-offs here between consistency and availability; for the systems objects of interest here, is the increased availability during partitions or site failures afforded by optimistic methods worth the possible cost incurred by the necessity to resolve inconsistencies introduced among the replicates during the partition?

Appropriateness of current programming models for replicated data

Are programming models useful in programming non-replicated objects also useful when replication is to be taken into account? That is, do the present models suffice, or must new models be developed for efficient programming of replicated objects?

Support needed for replication

What support is needed from the run-time system and from the operating system to support the programming of replicated objects?

Support needed to ensure forward progress

What support is needed from the job scheduler [McKe84a] to continue execution of operations on replicated objects despite failures?

Based on the knowledge gleaned about programming for resilience and availability during these studies, we propose to design language features to aid in programming fault-tolerant, replicated objects. Besides this upwards migration of knowledge, a downwards migration -- capabilities required of the language run-time support, as well as the action and object managers -- should manifest itself.

3.2. Preliminary Studies

As examples of the types of explorations which will be made in the course of this research, some preliminary studies are presented here. These examples include investigation of the programming of replicated objects using *recoverable variables* and using *per-action* and *permanent variables*. Also, a tentative design for a permanent heap to provide part of the run-time support for these objects is presented.

3.2.1. Replication Using Recoverable Variables

An example of an Aeolus object programmed to provide availability via replication is shown in Example 1. This object, which implements a simple symbol table, is derived from a single-copy (non-replicated) object (discussed in [LeB185]) which uses actions to provide recovery "firewalls" around its critical operations and the Clouds lock mechanism to specify synchronization rules which allow a high degree of concurrency the use of its operations.

The replicated SYMTAB object shown here uses *recoverable variables* to provide resilience of data in the face of failures. Recoverable variables, discussed in [Allc83b], are similar to *versions* of distributed database work, and require the creation of a new version (copy) of a recoverable variable for each action which modifies the variable. These versions are maintained on a *version stack* by the action managers of the Clouds system, which control visibility of the versions to nodes of the action tree, as well as writing versions to stable storage upon action commit.

Availability of the SYMTAB object is achieved by replication. Two instances of this object (*partners*) are created by the parent process, one of which is arbitrarily made the *master* instance, while the other is made the *slave*. Operations may be invoked on either of these instances; however, the slave instance merely relays the operation requests to the master. The master instance will then carry out the operations both locally and at the slave instance (by means of special operations) as if they had been originally invoked at the master. Were the instances equal partners (that is, if operations could be carried out at either instance), it would be necessary for each instance to obtain appropriate locks both locally and at the partner in order to maintain consistency, which could lead to deadlock unless some sort of global locking mechanism is available. In our implementation, locks need be obtained only at the master; thus, deadlock problems are avoided without the need for global locks. Should the slave instance become unavailable to the master (because of network partition or failure of the slave's site), the master will no longer attempt to replicate operations at the slave until a reinitialization operation is invoked by the Clouds action/object management support. Should the master instance become unavailable to the slave, the slave makes itself the master and carries out operations locally until reinitialization is invoked. When reinitialization occurs, the original master instance is again made master, and the states of the two instances are merged. The merge process is aided by maintenance of a *version vector*, which contains a sort of timestamp of both the latest version of the data maintained locally and the last version which was known to be consistent with the partner instance.

The implementation of the replicated SYMTAB object stresses availability at the expense of strict consistency among the replicated instances across partitions by using an optimistic (non-blocking) recovery method (see the section on "Related Work"). Rather, it seeks to maintain a "reasonable" view of the data at each instance, and resolves inconsistencies during the state-merge process using backout or compensatory operations. The programming of the merge process takes advantage of semantic knowledge of the object. Thus, the replicated SYMTAB object may be used -- despite site failures or partitions -- at any node at which an instance is available, without running the risk of an operation being blocked in order to maintain global consistency.

Example 1. Replicated SYMTAB object using recoverable variables

```
implementation of object symtab ( replicate_number : integer ) is

    ! Two-copy master/slave symbol table object using the action management
    ! facilities of Aeolus/Clouds for recovery firewalls and the lock
    ! mechanisms for synchronization, and demonstrating optimistic
    ! (non-blocking) site crash and partition recovery methods.

    ! The definitions of MAXREPLICATE, REPLICATE_RANGE, and VERSION_VECTOR
    ! actually appear in the definition part of SYMTAB, but are shown here for
    ! convenience.

    MAXREPLICATE : const integer := 2

    type replicate_range is 1 .. MAXREPLICATE

    type version_vector is array [replicate_range] of integer

    !
    ! The actual declarations of the implementation part.
    !

    here, there : recoverable replicate_range
                  ! for storing values of replicate numbers

    MAXBUCKET : const integer := 101 ! or whatever

    type hash_range is 1 .. MAXBUCKET

    type ptr_entry is -> symtable_entry

    type symtable_entry is ! just something for demo purposes
        record
            name : valstring ,
            next : ptr_entry
        end record

    type symtab_type is array [hash_range] of ptr_entry

    symtable : recoverable symtab_type

    symentry_lock : lock ( write : [],
                           read : [read] ) domain is hash_range
    ! The SYMENTRY lock allows locking of individual hash buckets in the
    ! symbol table. Several READ operations are allowed to proceed
    ! concurrently, but a WRITE operation blocks all other operations.

    symtable_lock : lock ( write : [write] ,
                           read : [read] )
    ! The SYMTABLE lock allows the entire symbol table to be locked.
    ! This lock is set in the EXACT_LIST operation for purposes of
    ! getting an exact listing of the state of the symbol table.
    ! Operations which change the state of the symbol table must wait for
    ! completion of any outstanding EXACT_LIST operations.

    partner : recoverable symtab ! Object pointer to the partner object

    master : boolean ! remember whether this instance is master or slave

    local_version : recoverable version_vector
    ! The LOCAL_VERSION vector is used to store version numbers of the local
    ! state (the HERE entry) and of the last version of the local state
    ! known to be consistent with the partner's state (the THERE entry).
    ! Note, however, that only one copy (per instance) of the actual state
    ! is maintained.
```

```

procedure hash ( name : valstring ) : hash_range is
    ! This HASH function is a local (nonpublic) procedure of
    ! the SYMTAB object.
    begin
        ! the usual type of stuff
    end procedure ! hash !

```

```

procedure send_state () is action
    ! The SEND_STATE operation is called by the partner to copy the local
    ! state to the partner.
    ! Here, knowledge of object semantics can be of great help in fine-tuning
    ! the state transfer process. For example, if the number of changes
    ! expected over an average expected partition period is small with respect
    ! to the total number of symbol table entries, it might be better to keep a
    ! log of those changes made during the partition period and then execute
    ! those changes on the partner's state. The method shown below for copying
    ! the entire local state to the partner assumes that the number of changes
    ! will be large with respect to total size. Also, the method shown --
    ! total reconstruction of the symbol table -- is used
    ! since the symbol table data structure uses physical pointers. Had the
    ! data structure used integer indices instead, the state could have been
    ! copied directly to the partner.

```

```

    next_entry : ptr_entry
    i          : index hash_range

    begin
        for i := 1 to MAXBUCKET loop
            next_entry := symtable [i]
            while next_entry <> NIL loop
                partner @ sym_insert (next_entry -> .name, local_version [here])
                next_entry := next_entry -> .next
            end loop
        end loop
        local_version [there] := local_version [here]
    end procedure ! send_state !

```

```

procedure receive_state () is action
    ! The RECEIVE_STATE operation is called by the partner's SEND_STATE
    ! operation to receive a copy of the partner's state.
    ! This operation doesn't do anything because of the symbol table data
    ! structure and method of state transfer used above. If the data structure
    ! used indices instead of pointers, this operation would install the copy of
    ! the state sent by the partner.
    begin
        null
    end procedure ! receive_state !

```

```

procedure transfer_state (! partner_version : version_vector ! ) is action
    ! The TRANSFER_STATE operation is called by the MERGE_STATE operation to
    ! have the partner transfer its state to us. If the partner becomes
    ! unavailable, make this instance the master.
    begin
        aid := action (partner @ send_state ())
        if Committed (aid) then
            local_version [here], local_version [there] := partner_version [there]
        else
            partner_available := FALSE
            master             := TRUE
        end if
    end procedure ! transfer_state !

```

```

procedure reconcile_states ( partner_version : version_vector ) is action
! The RECONCILE_STATE procedure is called locally (by the MERGE_STATE
! operation) to merge the local state and the partner's state in
! the case where both states have been updated since a partition
! occurred.
begin
    null ! later
end procedure ! reconcile_states !

```

```

procedure sym_insert (! newname : valstring, newversion : integer !) is action
! The SYM_INSERT operation may be invoked as an action.
! It is called either by the INSERT operation below (if this instance is
! the master), or by the partner as an update operation (if this instance
! is the slave).
! Locks on the symbol table and the particular hash entry concerned are
! obtained by the caller.
! If NEWVERSION is greater than 0 (that is, the SYM_INSERT operation
! was called remotely), then this new version number is installed in the
! LOCAL_VERSION array.

```

```

entry      : ptr_entry
bucket_num : hash_range

```

```

begin
    bucket_num := hash (newname)
    new (entry)
    using ent := entry -> do
        ent.name := newname
        ent.next := symtable [bucket_num] -> .next
    end using
    region symtable [bucket_num] do
        symtable [bucket_num] := entry
    end region
    if newversion > 0 then
        local_version [here], local_version [there] := newversion
    end if
end procedure ! sym_insert !

```

```

procedure insert (! newname : valstring !) is action
! The INSERT operation may be called as an action.
! If this instance is the master, it sets write locks on the symbol table
! and on the hash entry to be changed, and then calls the SYM_INSERT
! operation both locally and at the slave partner (if available) to do
! the actual insertion.
! If this instance is the slave, it calls the master (if available) to do
! the insertion just as if the call had originated there. If the master
! is not available, the slave makes itself the master and performs the
! insertion.
! Note that, since only the master can call the SYM_INSERT operation
! (which actually performs the insertion), it is not necessary to set
! locks at the slave.

newversion : integer
aid        : action_id

begin
  if master then
    newversion := local_version [here] + 1
    SetLock (symtable_lock, write)
    SetLock (symentry_lock, write, hash (newname))
    sym_insert (newname, 0)
    local_version [here] := newversion
    if partner_available then
      aid := action (partner @ sym_insert (newname, newversion))
      Await (aid) ! block until the nested action commits or aborts
      if Committed (aid) then
        local_version [there] := newversion
      else
        partner_available := FALSE
      end if
    end if
  else ! this instance is the slave
    aid := action (partner @ insert (newname))
    Await (aid)
    if not Committed (aid) then ! become master
      partner_available := FALSE
      master := TRUE
      newversion := local_version [here] + 1
      SetLock (symtable_lock, write)
      SetLock (symentry_lock, write, hash (newname))
      sym_insert (newname, 0)
      local_version [here] := newversion
    end if
  end if
end procedure ! insert !

.....

merge_lock : lock ( busy : [] )

```

```

procedure merge_state (! partner_version : version_vector !) is action
! The MERGE_STATE operation is invoked by the master partner after it{
! receives a REINIT invocation after a partition or crash.
! This operation compares the local version vector (LOCAL_VERSION)
! with that of the partner (PARTNER_VERSION), and takes appropriate action
! to merge its state with that of the partner into a single,
! consistent state.
begin
  SetLock (merge_lock, busy)
  master := FALSE
  if local_version [here] = partner_version [there] then
    ! The local and partner states are already consistent
    return .
  elsif local_version [here] = partner_version [here] then
    ! The local state hasn't changed since the last time the
    ! states were consistent; copy the partner's state here
    transfer_state (partner_version)
  elsif local_version [there] = partner_version [there] then
    ! The partner's state has not changed since the last time the
    ! states were consistent, so just copy over the local state
    ! to the partner
    partner @ transfer_state (local_version)
  else
    ! Both the local and the partner's state have changed since
    ! the two states were last consistent (partition case),
    ! so we must merge them
    reconcile_states (partner_version)
  end if
end procedure ! merge_state !

```

```

procedure reinit () is action
! The REINIT operation is invoked by the object manager in charge of this
! instance when the site on which it is running comes back up after a
! crash, or when a partition ends and this instance's state must be merged
! with its partner's state.
! If the MERGE lock is set, the partner has already initiated the merge
! process, and this instance is made the slave.
! If the MERGE lock is not set, this instance is the first to have the
! REINIT operation invoked by the object manager, so it becomes the master,
! invokes the partner's MERGE_STATE operation and passes it the local
! LOCAL_VERSION array; the partner (if available) then determines what
! needs to be done to merge the two states into a consistent state.

```

```

aid : action_id
begin
  if TestLock (merge_lock, busy) then
    return .
  end if
  SetLock (merge_lock, busy)
  master := here = 1 ! arbitrary choice
  if master then
    aid := action (partner @ merge_state (local_version))
    Await (aid)
    partner_available := Committed (aid)
  else
    aid := action (partner @ reinit ())
    Await (aid)
    if not Committed (aid) then
      master := TRUE
      partner_available := FALSE
    end if
  end if
end procedure ! reinit !

```

```
procedure set_partner (! p : symtab, rep_number : replicate_range !) is
  ! The SET_PARTNER operation is used by the creating process to
  ! initialize the PARTNER object pointer.
begin
  partner := p
  there   := rep_number
end procedure ! set_partner !

begin ! initialization section
  here      := replicate_number
  master    := here = 1 ! arbitrary choice
  local_version := version_vector[ 0 : MAXREPLICATE ]
  symtab    := symtab_type[ NIL : MAXBUCKET ]
             ! symbol table is initially empty
end implementation.
```

3.2.2. Replication Using Permanent and Per-Action Variables

Since the use of a *recoverable* symbol table data structure requires the creation of a complete copy of the symbol table on the version stack for each action which modifies the data structure, the implementation of the replicated SYMTAB object presented in the preceding subsection can become inefficient as the size of the symbol table increases. Fortunately, we can use semantic knowledge about the object to simulate the effect of recoverable variables at a fraction of their cost. The technique which we will use is introduced in [McKe84b].

An implementation of the SYMTAB object which uses this new technique is shown in Example 2. Rather than require the system to maintain a version of the symbol table per action, we will maintain lists of those elements inserted or deleted by each action. These lists are maintained in *per-action variables*, copies of which are created for each action during its BOA (beginning of action) phase; an action may access not only its own per-action variables, but those of its parent (if any). We provide handlers for the *abort* and *nested commit* events of actions, which clean up after action aborts and propagate the values of the per-action variables to the parent of the current action. Only one copy of the entire symbol table data structure is maintained, as a *permanent variable*. Permanent variables are maintained in a special area of per-object storage, and are managed by a shadowing mechanism provided by the Clouds kernel storage management system [Pitt84]. The shadows are created during the *precommit* action event; thus, we provide a handler for the *oplevel precommit* event which performs the actual insertions and deletions on the permanent symbol table, using the INSERTED and DELETED lists which we have propagated up the action tree.

As in Example 1, only the INSERT operation is shown for sake of brevity; however, the form of the DELETE operation would be very similar to that of INSERT, except that items would be added to the DELETED per-action list rather than to the INSERTED list. The processing of the DELETED list during action events is shown in the code for the alternate event handlers.

Example 2. Replicated SYMTAB object using permanent and per-action variables

implementation of object symtab (replicate_number : integer) is

```
! Two-copy master/slave symbol table object using the action management
! facilities of Aeolus/Clouds for recovery firewalls and the lock
! mechanisms for synchronization, and demonstrating optimistic
! (non-blocking) site crash and partition recovery methods.
! Permanent variables, rather than recoverable variables, are used for
! for efficiency.
```

```
! Names are given here for alternate handlers provided for some of the action
! events.
```

action events

```
    abort is sym_abort,
    nested_commit is sym_nested_commit,
    toplevel_precommit is sym_top_precommit
```

```
! The definitions of MAXREPLICATE, REPLICATE_RANGE, and VERSION_VECTOR
! actually appear in the definition part of SYMTAB, but are shown here for
! convenience.
```

```
MAXREPLICATE : const integer := 2
```

```
type replicate_range is 1 .. MAXREPLICATE
```

```
type version_vector is array [replicate_range] of integer
```

```
!
! The actual declarations of the implementation part.
!
```

```
here, there : permanent replicate_range
! for storing values of replicate numbers
```

```
MAXBUCKET : const integer := 101 ! or whatever
```

```
type hash_range is 1 .. MAXBUCKET
```

```
type ptr_entry is -> permanent symtable_entry ! allocate in perm. heap
```

```
type symtable_entry is ! just something for demo purposes
record
    name : valstring ,
    next : ptr_entry
end record
```

```
type symtab_type is permanent array [hash_range] of ptr_entry
! the array of pointers is in perm. storage
```

```
symtable : symtab_type
```

```
symentry_lock : lock ( write : [],
    read : [read] ) domain is hash_range
! The SYMENTRY lock allows locking of individual hash buckets in the
! symbol table. Several READ operations are allowed to proceed
! concurrently, but a WRITE operation blocks all other operations.
```

```
symtable_lock : lock ( write : [write] ,
    read : [read] )
! The SYMTABLE lock allows the entire symbol table to be locked.
! This lock is set in the EXACT_LIST operation for purposes of
! getting an exact listing of the state of the symbol table.
! Operations which change the state of the symbol table must wait for
! completion of any outstanding EXACT_LIST operations.
```

```

partner      : permanent symtab  ! Object pointer to the partner object
master      : boolean  ! remember whether this instance is master or slave

local_version : recoverable version_vector
! The LOCAL_VERSION vector is used to store version numbers of the local
! state (the HERE entry) and of the last version of the local state
! known to be consistent with the partner's state (the THERE entry).
! Note, however, that only one copy (per instance) of the actual state
! is maintained.

! The per-action variables of the SYMTAB object.
! We will maintain lists of those entries inserted and deleted by
! each action. The per-action variables are headers of those lists,
! which are pointers to entries allocated in the permanent heap.
! There are two standard names, Self and Parent, which refer to the
! per-action records of the current action and its parent,
! respectively.
! The PER-ACTION declaration causes a record type with name PERACTION,
! as well as the names Self and Parent with that record type, to be added
! to the Aeolus compiler's symbol table.

per-action is
  record
    inserted, deleted : ptr_entry
  end record
  init peraction"[ NIL:2 ]  ! For initialization of Self at action start

procedure hash ( name : valstring ) : hash_range is
  ! Same as in Example 1.

procedure send_state ( ) is action
  ! Same as in Example 1.

procedure receive_state ( ) is action
  ! Same as in Example 1.

procedure transfer_state (! partner_version : version_vector ! ) is action
  ! Same as in Example 1.

procedure reconcile_states ( partner_version : version_vector ) is action
  ! Same as in Example 1.

```

```

procedure sym_insert (! newname : valstring, newversion : integer !) is action
! The SYM_INSERT operation may be invoked as an action.
! It is called either by the INSERT operation below (if this instance is
! the master), or by the partner as an update operation (if this instance
! is the slave).
! Locks on the symbol table and the particular hash entry concerned are
! obtained by the caller.
! If NEWVERSION is greater than 0 (that is, the SYM_INSERT operation
! was called remotely), then this new version number is installed in the
! LOCAL_VERSION array.
! The insertion is noted on the INSERTED per-action list of the current
! action, but is not actually performed until toplevel precommit.

```

```

entry      : ptr_entry
bucket_num : hash_range

```

```

begin
  bucket_num := hash (newname)
  new (entry)
  using ent := entry -> do
    ent.name := newname
    ent.next := Self.inserted -> .next
  end using
  Self.inserted := entry
  if newversion > 0 then
    local_version [here], local_version [there] := newversion
  end if
end procedure ! sym_insert !

```

```

procedure insert (! newname : valstring !) is action
! Same as in Example 1.
! Note that manipulation of the INSERTED per-action list is performed by
! the SYM_INSERT operation, which is called by the INSERT operation of the
! master.

```

.....

```

procedure sym_abort () is
! The SYM_ABORT procedure is the alternate handler for the ABORT action
! event for the SYMTAB object.
! It frees all space which was allocated in the permanent heap for
! items inserted by the action being aborted.

```

```

entry, next_entry : ptr_entry

```

```

begin
  entry := Self.inserted
  while entry <> NIL loop
    next_entry := entry -> .next
    dispose (entry)
    entry := next_entry
  end loop
end procedure ! sym_abort !

```

```

procedure sym_nested_commit () is
! The SYM_NESTED_COMMIT procedure is the alternate handler for the
! NESTED-COMMIT action event for the SYMTAB object.
! It adds the INSERTED and DELETED lists for the nested action being
! committed to the beginning of the respective lists of the action's parent.

entry : ptr_entry

begin
entry := Self.inserted
if entry <> NIL then
loop      ! find the end of the INSERTED list
  if entry -> .next = NIL then
    exit .
  end if
  entry := entry -> .next
end loop
entry -> .next := Parent.inserted
Parent.inserted := entry
end if

entry := Self.deleted
if entry <> NIL then
loop      ! find the end of the DELETED list
  if entry -> .next = NIL then
    exit .
  end if
  entry := entry -> .next
end loop
entry -> .next := Parent.deleted
Parent.deleted := entry
end if
end procedure ! sym_nested_commit !

```

```

procedure sym_top_precommit () is
! The SYM_TOP_PRECOMMIT procedure is the alternate handler for the
! TOPLEVEL-PRECOMMIT action event for the SYMTAB object.
! It inserts or deletes each item in the INSERTED or DELETED list for this
! action into (or out of) the permanent symbol table. At this point,
! the memory management system will create shadow versions of the pages
! in the permanent version affected by these changes.
! Note that, since the action management system promises that only one
! action can enter its PRECOMMIT stage at a time, no further locking for
! mutual exclusion is necessary.

entry, next_entry : ptr_entry
place, prev_place : ptr_entry
bucket_num       : hash_range

begin
  entry := Self.inserted
  while entry <> NIL loop
    next_entry := entry -> .next
    bucket_num := hash (entry -> .name)
    entry -> .next := symtable [hash]
    symtable [bucket_num] := entry
    entry := next_entry
  end loop

  entry := Self.deleted
  while entry <> NIL loop
    next_entry := entry -> .next
    bucket_num := hash (entry -> .name)
    place      := symtable [bucket_num]
    loop      ! find the entry in the permanent symbol table and remove it
      if place = NIL then ! not there, so don't worry about it
        exit .
      elsif place -> .name = entry -> .name then ! got it
        if place = symtable [bucket_num] then ! at start of bucket
          symtable [bucket_num] := place -> .next
        else
          prev_place -> .next := place -> .next
        end if
      else
        prev_place := place
        place      := place -> .next
      end if
    end loop
    entry := next_entry
  end loop
end procedure ! sym_top_precommit !

.....

merge_lock : lock ( busy : [] )

procedure merge_state (! partner_version : version_vector !) is action
! Same as in Example 1.

procedure reinit () is action
! Same as in Example 1.

procedure set_partner (! p : symtab, rep_number : replicate_range !) is
! Same as in Example 1.

```

```
begin ! initialization section
  here      := replicate_number
  master    := here = 1 ! arbitrary choice
  local_version := version_vector"[ 0 : MAXREPLICATE ]
  symtab    := symtab_type"[ NIL : MAXBUCKET ]
              ! symbol table is initially empty
end implementation.
```

3.2.3. The Permanent Heap

The design of the SYMTAB object presented in the preceding subsection requires the use of linked lists allocated in a heap in the permanent area of per-object storage, both for its per-action and permanent variables. This *permanent heap* will require special run-time support for its management, which must maintain the heap's consistency across failures.

In Example 3, we show a preliminary design for the permanent heap manager. To maintain the consistency of the heap, this PERMHEAP object uses the same techniques which we used in the SYMTAB object of Example 2 to implement recovery for the symbol table data structure, i.e., per-action variables and associated action-event handlers. (In fact, due to its compactness, this example may demonstrate the use of these techniques more clearly than does the SYMTAB example.) The actual management of memory is done by a HEAP object (whose definition is shown in Example 3 for clarity), which can allocate and free blocks of memory in both the permanent and the temporary heap areas. The HEAP object does not implement recoverability at present; however, once the PERMHEAP object is available, the HEAP object may be altered to use the permanent heap for its FREE list and bootstrapped.

The PERMHEAP object maintains lists of those areas of the heap allocated and freed by each action, in per-action variables. Since the HEAP object (which is at present nonrecoverable) does the actual management of the heap, allocations are visible to other actions immediately, thus maintaining the consistency of the heap. A call to the ALLOCATE operation of PERMHEAP will return a pointer to a block of memory allocated by HEAP in the permanent heap area of the object; a pointer to the block is also added to the ALLOCATED per-action list. A call to PERMHEAP's FREE operation will actually dispose the block of memory only if it was allocated by the action which is trying to free it; otherwise, a pointer to the block to be disposed is merely added to the FREED per-action list. Upon abort of an action which allocated permanent heap storage, the ALLOCATED list is used to clean up the heap via calls to HEAP's FREE operation. When a nested action enters its commit phase, its ALLOCATED and FREED per-action lists are propagated to its parent. Memory blocks on the permanent heap allocated by an action are actually disposed when the action's toplevel ancestor (to which the nested action's per-action lists have been propagated) enters its precommit phase; this is done by invoking the FREE operation of HEAP on all members of the toplevel action's FREED list:

Note that this implementation of the PERMHEAP object does not provide strict serializability. To see this, consider some action, A, which exhausts (or nearly exhausts) the permanent heap, causing other actions B and C trying to allocate permanent memory to fail. Action A may well be aborted itself. Actions B and C which failed because of A might not have failed had they been executed serially. However, such breaches of strict serializability do not affect the consistency of the permanent heap mechanism, and thus are of little concern in this context.

Example 3. Run-time support for the permanent heap

```
implementation of object permheap is
! Support for the permanent heap, using per-action variables for
! recovery management.

uses heap
! The definition of the HEAP pseudo-object is shown here for clarity.
! The HEAP object implements a standard heap management discipline (i.e.,
! without recovery), but allows one to allocate memory in either the
! permanent or the temporary memory area.
!
! definition of object heap is
!   type heap_type is ( normal_heap, permanent_heap )
!   operations
!     procedure allocate ( size : unsigned ,
!                         kind : heap_type ) : address
!       -- the ALLOCATE operation returns a pointer to a block of
!       -- memory of the specified SIZE in the area of memory
!       -- indicated by KIND.
!     procedure free ( block : address )
!       -- the FREE operation disposes the block of memory pointed
!       -- to by BLOCK.
!   end definition.

! The local declarations of the PERMHEAP object.
!
! Give the names of alternate handlers for some of the action events.

action events
  abort is permheap_abort,
  nested_commit is permheap_nested_commit,
  toplevel_precommit is permheap_top_precommit

! The BLOCKLIST type is used in the declaration of the per-action variables
! below.

type ptr_blocklist is -> blocklist

type blocklist is
  record
    block : address,
    next : ptr_blocklist
  end record

! The per-action variables for permanent-heap recovery management.
! We will maintain lists of memory blocks allocated and freed by each action.

per_action is
  record
    allocated, freed : ptr_blocklist
  end record
  init peraction"[ NIL:2 ]
```

```

!
! The operations of the PERMHEAP object.
!

```

```

procedure allocate (! size : unsigned ! ) ! : address ! is
! Return a pointer to a block of memory of the given SIZE in
! permanent memory.

list : ptr_blocklist

begin
  new (list)          ! create a new entry for the ALLOCATED list
  using l := list -> do
    l.block           := heap @ allocate (size, permanent_heap)
    l.next            := self.allocated
    Self.allocated    := list ! put new entry at beginning of ALLOCATED list
    return l.block
  end using
end procedure ! allocate !

```

```

procedure free (! block : address ! ) is
! Dispose the block of memory indicated by BLDCK.

prev, list : ptr_blocklist

begin
  list, prev := Self.allocated ! First, scan the ALLOCATED list to see if
  loop          ! BLDCK was allocated by the current action
    if list = NIL then          ! Nope, go below
      exit .
    elsif list -> .block = block then ! Yes, so
      if prev = next then      ! remove it from ALLOCATED list;
        Self.allocated := NIL
      else
        prev -> .next := list -> .next
      end if
      heap @ free (list -> .block) ! go ahead and dispose it
      dispose (list)
      return .                    ! we're done
    else
      prev := list
      list := list -> .next
    end if
  end loop
  new (list)          ! If we get here, BLOCK wasn't allocated by the
  using l := list -> do ! current action, so put it on the FREED list
    l.block := block
    l.next  := Self.freed
  end using
  Self.freed := list
end procedure ! free !

```

```

procedure permheap_abort () is
! The alternate handler for the ABORT action event.
! We'll just free all the space allocated by this action as indicated
! by the ALLOCATED list, and clean up the FREED list for good measure.

list, old : ptr_blocklist

begin
list := Self.allocated
while list <> NIL loop
heap @ free (list -> .block)
old := list
list := list -> .next
dispose (old)
end loop

list := Self.freed
while list <> NIL loop
old := list
list := list -> .next
dispose (old)
end loop
end procedure ! permheap_abort !

procedure permheap_nested_commit () is
! The alternate handler for the NESTED_COMMIT action event.
! We'll propagate the items on the ALLOCATED and FREED lists of this
! action to the beginning of the corresponding lists of its parent action.

list : ptr_blocklist

begin
list := Self.allocated
if list <> NIL then
loop ! find the end of the ALLOCATED list
if list -> .next = NIL then
exit .
end if
list := list -> .next
end loop
list -> .next := Parent.allocated
Parent.allocated := list
end if

list := Self.freed
if list <> NIL then
loop ! find the end of the DELETED list
if list -> .next = NIL then
exit .
end if
list := list -> .next
end loop
list -> .next := Parent.freed
Parent.freed := list
end if
end procedure ! permheap_nested_commit !

```

```

procedure permheap_top_precommit () is
  ! The alternate handler for the TOPLEVEL_PRECOMMIT action event.
  ! We'll use the normal HEAP operation FREE to dispose of the memory blocks
  ! on the FREED list, but we'll just dispose the ALLOCATED list -- it's only
  ! used to free up storage allocated by an aborting action.

  list, old : ptr_blocklist

begin
  list := Self.freed
  while list <> NIL loop
    heap @ free (list -> .block)
    old := list
    list := list -> .next
    dispose (old)
  end loop

  list := Self.allocated
  while list <> NIL loop
    old := list
    list := list -> .next
    dispose (old)
  end loop
end procedure ! permheap_top_precommit !

begin ! Object initialization
  null
end implementation.

```

4. RELATED WORK

As with most of the topics involved in the study of distributed systems, the synchronization and recovery of replicated data was first studied in the area of distributed database systems. The history of these efforts is summarized by Wright [Wrig83]. He classifies these methods as *conservative* (*pessimistic*, *blocking*) and *optimistic* (*non-blocking*). Examples of conservative methods are voting schemes [Giff79, Thom78], primary copy methods [Ston79], and token-passing schemes [LeLa78]. The intent of these methods is to ensure consistency of the replicated data by requiring access to a special copy or set of copies of the data during partitions. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods; a token is passed among sites holding a copy of data, and that copy at the site currently holding the token is considered the primary copy. Yet another extension of primary copy methods are the voting schemes. Each copy of the data object is assigned a (possibly different) number of votes; a partition possessing a majority of the votes for that object may access it. The conservative schemes are called *blocking* since a data object is not available at a site in a partition which does not possess the primary copy (or token or majority of votes); thus, the access must block until the partition is ended, even if a copy of the data is available in the partition. Indeed, under these schemes it is possible that no partition may have access to the data object.

The optimistic methods do not seek to ensure global consistency of replicated data during partitions [Davi81, Davi82]. Thus, accesses are not blocked if a replicate of the data is available in the partition in question. Rather, inconsistencies in the data replicates are resolved during a merge process once the partition is ended, by use of backouts or compensatory actions. It is assumed that the number of such inconsistencies will be small (hence, *optimistic*). However, tradeoffs may be made between consistency and availability. For example, the *Data-Patch* tool for designing replicated databases [Blau82, Garc83] assumes that, rather than strict consistency, a "reasonable" view of the database should be maintained to enhance availability.

Wright develops enhancements to both the conservative and the optimistic methods. Conservative schemes are extended by the notion of compatibility among classes of transactions, which allow increased efficiency and availability with these methods of concurrency control. He also considers the computational complexity of the problem of backing out transactions under optimistic schemes, and shows that (in general) the problem is NP-complete. He then develops efficient heuristic solutions to this problem.

However, Wright's work (as is most of the work previously discussed) assumes a simple data model based on reads and writes. He does speculate that the semantic knowledge about objects available in object-oriented systems may bring about the possibility of gains in efficiency over his model, since the read-write model places unnecessary restrictions on availability and concurrency when used with more complex objects. The *Data-Patch* tool mentioned above takes advantage of semantic knowledge through its YACC-like approach to the construction of partition-merge routines for databases.

Previous work in the area of replication of data in distributed operating systems includes work on the LOCUS system at UCLA [Walk83] as well as the Argus system at MIT [Herl84] and the ISIS system at Cornell [Birm84]. The LOCUS system supports replicated files and directories using an optimistic approach; inconsistencies are allowed to develop among the separate partitions which are resolved (except in the case of simple read-write file objects) by application-dependent measures. No mention is made of system support for the applications' recovery methods. Herlihy's work at MIT uses semantic knowledge of Argus objects to enhance a conservative (voting) method. Analysis of the algebraic structure of data types is used in the choice of appropriate intersections of voting quorums. The ISIS system supports *k-resilient* objects (objects replicated at $k+1$ sites and which can withstand up to k failures) by means of checkpoints and the "available copies" voting algorithm. This system provides both availability and *forward progress*, that is, even after up to k site failures, enough information is available at the remaining sites possessing an object replicate that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme similar to the master-slave discipline shown in the previous section.

Thus, recalling the issues detailed in Section 3.1, we believe that the proposed research will lead to contributions in several of the areas mentioned. The use of semantic knowledge of objects in the programming of non-blocking replication methods has not been the subject of much previous study, especially in the context of systems programming problems. The trade-offs mentioned, between consistency and availability and between blocking and non-blocking replication methods, have been the focus of some work in the database realm, but again the issues have not been treated in operating systems. The issue of appropriate programming models for availability in action/object systems has not been treated before. Rather, in systems such as Argus, the applications language has been designed *ab initio*, and the system as well as programming models for it have been cut to fit. Finally, the study of support needed for availability and for forward progress should provide valuable insights.

5. OUTLINE OF DISSERTATION

A proposed outline for the dissertation resulting from the proposed research is presented here. Since this is an exploratory thesis, the answer to the perennial question "how can we tell when you are done?" is a difficult one. There will be some tangibles; in particular, designs for the run-time support system for Aeolus and for the interfaces to the action manager, object manager, and job scheduling systems should be forthcoming. However, completion of several portions of the work will depend on our satisfaction with the completeness of the set of case studies and with the insights into the design of language features which these case studies may yield.

Introduction

Background and terminology for the research to be discussed will be presented in terms of an overview of the Clouds project and of the Aeolus language. The goals and plan of the research will be described.

Contributions

The contributions of the research will be summarized.

Related Work

Previous work in this area will be discussed and compared to this research.

Case Studies

The results of the explorations in programming methodology for replicated data will be presented and discussed.

Language Features for Resilience and Availability

Those features whose designs result from the case studies will be presented and discussed; in particular, comparisons will be made with features provided in other languages for distributed applications.

Run-Time and Operating System Support for Replicated Data

Designs or suggestions for support features resulting from the case studies will be presented and compared to support provided by other systems.

Conclusions and Further Work

The work done and its contributions are summarized; ideas for further work beyond the scope of this research which may develop are presented.

6. REFERENCES

- [Allc82] Allchin, J. E., and M. S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982
- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also available as technical report GIT-ICS-83/23)
- [Birm84] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El Abbadl, "Implementing Fault-Tolerant Distributed Objects," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984
- [Blau82] Blaustein, B., R. M. Chilenskas, H. Garcia-Molina, D. R. Ries, and T. Allen, "Partition Recovery Using Semantic Knowledge," Computer Corporation of America, Cambridge, Massachusetts, November 1982
- [Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [Davi81] Davidson, S., and H. Garcia-Molina, "Protocols for Partitioned Distributed Database Systems," *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, Pennsylvania, July 1981
- [Davi82] Davidson, S., "An Optimistic Protocol for Partitioned Distributed Database Systems," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1982
- [Garc83] Garcia-Molina, H., T. Allen, B. Blaustein, R. M. Chilenskas, and D. R. Ries, "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition," *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, Clearwater Beach, Florida, October 1983
- [Giff79] Gifford, D. K., "Weighted Voting for Replicated Data," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979
- [Herl84] Herlihy, M. P., "Replication Methods for Abstract Data Types," Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1984 (available as MIT/LCS/TR-319)
- [LeB185] LeBlanc, R. J., and C. T. Wilkes, "Systems Programming with Objects and Actions," to appear in *Proceedings of the Fifth International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)
- [LeLa78] LeLann, G., "Algorithms for Distributed Data-Sharing Systems which use Tickets," *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, California, August 1978
- [McKe84a] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," School of Information and Computer Science, Georgia Institute of Technology, May 1984
- [McKe84b] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)

- [Pitt84] Pitts, D., "Storage Management for an Action-Based Operating System," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, November 1984 (also available as Technical Report GIT-ICS-85/02)
- [Ston79] Stonebreaker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering* 5, 3, May 1979
- [Thom79] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases," *ACM Transactions on Database Systems* 4, 2, June 1979
- [Walk83] Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 1983 (available as *Operating Systems Review* 17, 5)
- [Wrig83] Wright, D. D., "Managing Distributed Databases in Partitioned Networks," Ph.D. Thesis, Department of Computer Science, Cornell University, January 1984 (available as Technical Report 83-572, September 1983)