MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Computation Structures Group Memo 186

Programming Methodology Group Progress Report 1978-79

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01.

November 1979

PROGRAMMING METHODOLOGY

Academic Staff

B. H. Liskov, Group Leader
I. G. Greif

Research Staff

R. W. Scheifler	P. Johnson

Graduate Students

R. R. Atkinson	J. E. Moss
V. A. Berzins	J. C. Schaffert
T. Bloom	C. R. Seaquist
W. C. Gramlich	L. A. Snyder
M. P. Herlihy	M. K. Srivas
T. O. Humphries	E. W. Stark
D. Kapur	J. M. Wing

Undergraduate Students

M. D. Allen	B. J. Mirrer
L. R. Dennison	C. L. Mullendore
R. M. Knopf	J. L. Zachary
P. J. Leach	

Support Staff

S. Barefoot	J. Jones
	A. Rubin

Visitors

A. Merey	U. Montanari
	J. Peterson

A. Introduction

Our major research effort this year has been in the area of distributed systems. We have focussed on identifying linguistic primitives that would support the programming of distributed programs. We have also investigated the design of a new CLU compiler and runtime system, with the goals of efficiency and transportability. The primitives for distributed programs will be made available as extensions to CLU, and the extended language is intended to run on the nodes of a network of computers; the new CLU implementation supports this effort. Much of this work is being done jointly with the Computer Systems Research Group.

In conjunction with the Computer Systems Research Group, we sponsored an invitation-only workshop on Distributed Systems. This workshop was held at the Harvard University Faculty Club on October 12 and 13, 1978. Approximately 25 leading workers in the field assembled for two days to discuss research topics and direction. A report on the workshop has been published in Operating Systems Review [16].

We have continued our work on the current CLU system. A reference manual for CLU is now available, and the implementation has been moved to the PDP/20. In addition to this work on CLU, we have completed a study of synchronization primitives, the design of a new machine architecture to support object oriented languages, and the definition of a specification language that permits specification of mutable data abstractions.

In the next section we discuss the status of the present CLU implementation. Section C describes our work on primitives for distributed computing, while Section D discusses the new CLU implementation. The final three sections describe our work on synchronization primitives, object oriented machine architecture, and specification techniques.

B. Current CLU Implementation

During the past year substantial work has been done on the current CLU system. A real-time, display-oriented text editor, with a number of CLU-related features, has been written in CLU. Two improvements to the compiler have resulted in a reduction of the average module compilation time by a factor of four. Major portions of the CLU library, as well as a number of related utility programs, have been implemented. With the design of CLU essentially complete, a preliminary version of the reference manual has been published [14] and circulated for comments. Since its publication, several new data types and an own variable mechanism have been added, and we are in the process of revising the manual for final publication. In addition, the CLU system has been sent to a number of other research groups, both in the United States and Europe.

The first improvement to the compiler was a rewrite of the code generator. The previous code generator transformed the syntax tree for a module (as constructed by the parser and modified by the type-checker) into textual output in a macro language called CLUMAC, and ran a CLUMAC assembler as an inferior process to generate the actual object code. The new code generator produces object code directly from the syntax tree. This eliminates the costly intermediate translation to CLUMAC, and in practice reduces overall compilation time for a module by a factor of two.

The second improvement to the compiler was to augment the code generator to perform, as an option, inline substitution [18] (or open coding) of certain operations of the basic types (e.g., integers, records, and arrays). The only operations chosen were those that, when expanded inline, would result in at most a small increase in code size. The new code generator has been used to recompile the compiler itself, with the result that the new compiler runs twice as fast and is marginally smaller in size. Similar results have been found for other programs. An option was added to the compiler to restrict inline substitution to just those operations that would not increase code size, but in practice this option is not useful, in that simply performing all substitutions generally results in an overall decrease in code size.

In addition to work on the compiler, considerable progress has been made in implementing the CLU library [14]. The library is the repository for information about abstractions and their implementations. For each abstraction there is a description unit (DU) containing all system-maintained information about the abstraction, such as its interface specification. The DU also contains zero or more modules that implement the abstraction, and may additionally contain formal specifications, module interconnection information, and various documentation files. Each implementation contains source and object code, the compilation environment (CE) used to resolve external references in the source code, and other information.

CLU programs tend to be composed of many small modules. Thus the library will contain a great many DUs, and each DU, along with its associated implementations, will contain a number of relatively small pieces of information. Attempting to store this information in a conventional file system could be extremely inefficient in space. For example, one would probably choose to split the information in a DU into several small files: interface specification file, system information file, implementation files, and documentation files. Similarly, one would probably split an implementation into a number of files: source file, object file, system information file, and documentation files. Since files on most systems are composed of an integral number of fixed-size pages, where the page size is fairly large (e.g., 512 or 1024 words), a substantial amount of space can be lost due to breakage. Furthermore, such a library design would not be readily transportable, due to the wide variation in file naming schemes and protection mechanisms.

We have therefore designed and implemented a complete file system for storing large numbers of small files and directories efficiently. About half of this implementation is in assembly language, the rest in CLU. The file system is a tree structure, with the internal nodes being directories and directory-like objects (DUs, implementations, CEs), and the leaf nodes being files. Directory-like objects are implemented through abstract data types, with directories as the concrete representation. Each file is typed with one of five file types (e.g., text or object code). Each entry in a directory has a four part name, consisting of a two part string name of no more than 127 characters, a version number, and a generation (or edit) number. The protection mechanism employed is a variation of access control lists [17].

The CLU file system is contained in a single file of the host file system. This file is divided into logical pages, each page consisting of 1024 words. Each logical page is used to allocate blocks of some fixed size. The block sizes are powers of two, ranging from 8 to 1024 words. Each file (and directory) is composed of zero or more blocks, which need not all be the same size. If a file consists of more than one block, then the file has a header block containing a list of references to all subsidiary blocks. For each block size there is a global free-list chaining together all free blocks of that size. When no more free blocks of a particular size exist, a 1024-word block is broken up into blocks of the needed size.

A directory refers to a file (or subdirectory) with both a unique-id and a slot number. The slot number is used to index into a linear table (allocated from 1024-word blocks) to obtain a reference to the first block of the file, the number of data words in the block, and a flag indicating whether the block is the header of a multi-block file. The unique-id is redundant; it is stored as the first word of every block in the file, to aid recovery in the event of disastrous system crashes.

A command interpreter called SHELL has been written (in CLU) for interacting with the new file system. Many of the commands resemble those of DEC's TOPS-20 EXEC. The commands deal only with the file system; there are no facilities yet for running CLU programs directly from SHELL. In addition, a salvager and garbage collector have been written for use in case of system crashes, as well as an incremental dumper for backing up the file system onto magnetic tape. The CLU I/O facilities have been augmented to deal with this new file system, and hence the CLU-based text editor and the compiler can now access files there.

At present the file system contains only directories and files; DUs and related abstractions are still being designed. Although the file system is still only being used experimentally, the system appears to be nearly as fast as the host file system for most purposes.

C. Extended CLU

Distributed programs that run on nodes of a network are now technologically feasible, and are well-suited to the needs of organizations. However, our knowledge about how to construct such programs is limited. The distributed systems project has undertaken the study of the construction of distributed programs. This study involves

1. Identification of linguistic features that support distributed programming.

2. Experiments in constructing distributed programs.

The linguistic features will be used in the experiments, where they will reduce the effort needed to carry out the experiments. The experiments will provide an evaluation of the features; this evaluation will lead to refinement of existing features, and identification of new ones where appropriate.

In our research, we are influenced by some assumptions about hardware, and about the way in which that hardware will be used. We assume that distributed programs run on a collection of computers, called *nodes*, that are connected by means of a communications network. Each node consists of one or more processors, and one or more levels of memory. The nodes are heterogeneous, e.g., they may contain different processors, come in different sizes and provide different capabilities, and be connected to different external devices. The nodes can communicate only via the network; there is no (other) shared memory. This assumption is in contrast to multiprocessor systems such as CM* [8].

We make no assumptions about the network itself other than that it supports communication between any pair of nodes. For example, the network may be longhaul or shorthaul, or some combination with gateways in between; these details are invisible at the programmer level.

We assume that each node has an owner with considerable authority in determining what that node does. For example, the owner may control what programs can run on that node. Furthermore, if the node provides a service to programs running on other nodes, that service may be available only at certain times (e.g., when the node is not busy running internal programs) and only to certain users. We refer to such nodes as *autonomous*.

The principal consequence of the assumption of autonomy is that the programmer, not the system, must control where programs and data reside. The system may not breach the autonomy of a node by moving processing to it for purposes of load sharing. This attitude distinguishes our approach from others, such as the Actor system [9], where the mapping of a program to physical locations is entirely under system control. Work in the same general area includes [4] and [7], although autonomy is not explicitly addressed.

Our approach to the study of linguistic features is to extend an existing sequential language with primitives to support distributed programs. Our base language is CLU [14, 15]. Although the primitives are mostly independent of the base language, CLU is a good choice for two main reasons. It supports the construction of well-structured programs through its abstraction mechanisms, especially data abstractions; it is reasonable to assume that distributed programs will require such mechanisms to keep their complexity under control. Secondly, CLU is an object-oriented language, in which programs are thought of as operating on long-lived objects, such as data bases and files; this view is well-suited to the applications of interest, e.g., banking systems, airline reservation systems, office automation.

Our work this year has concentrated on primitives in two main areas, namely, modularity and communication. Below we discuss a new linguistic construct intended to support modular construction of distributed programs, and provide an example illustrating its use. More information about our work can be found in [13].

For distributed programs, a modular unit is needed that

1. Can be used to model the tasks and subtasks being performed in a reasonably natural way.

2. Can be realized efficiently, i.e., gives the programmer a realistic model of the underlying architecture.

A major issue in point (2) is control of direct sharing of data. An object that is shared directly (i.e., many entities know its location in the distributed address space) is a problem for three reasons. The object can be a bottleneck because of the contention for its use. It is a storage management problem, since to deallocate it while avoiding dangling references requires detection and invalidation of all references to it. Finally, to coordinate its use correctly can lead to increased program complexity. The main conclusion that can be drawn from considering these problems is that a linguistic mechanism that encourages the programmer to think about controlling the direct sharing of data is desirable. Note that a synchronization mechanism such as a monitor [5, 11] helps with the synchronization problem but not with the other two, since the monitor itself is a shared datum.

1. Guardians

We provide a construct called a *guardian* to support modular distributed programs. A guardian is a local address space containing objects and processes. A *process* is the execution of a sequential program. *Objects* contain data; objects are manipulated (accessed and possibly modified) by processes. Examples of objects are integers, arrays, queues, documents (in an office automation system), bank accounts and procedures. Objects are strongly typed: they may be directly manipulated only by operations of their type. The types may be either built-in or user-defined.

A computation consists of one or many guardians. Within each guardian, the actual work is performed by one or many processes. The processes within a single guardian may share objects, and communicate with one another via these shared objects.

Processes in different guardians can communicate only by sending messages. (Message passing is discussed in [13].) Messages will contain the *values* of objects, e.g., "2" or " 176538 $173.72" (the value of a bank account object). An important restriction ensures that the address space of a guardian remains local: it is impossible to place the *address* of an object in a message. It is possible to send a *token* for an object in a message; a token is an external name for the object, which can be returned to the guardian that owns the object to request some manipulation of the object. (A token is a sealed capability that can be unsealed only by the creating guardian.) The system makes no guarantee that the object named by the token continues to exist; only the guardian can provide such a guarantee. Thus a guardian is entirely in charge of its address space, and storage management can be done locally for each guardian.

A guardian exists entirely at a single node of the underlying distributed system: its objects are all stored on the memory devices of this node and its processes run on the processors of the node. During the course of a computation, the population of guardians will vary; new guardians will be created, and existing guardians may self-destruct. The node at which a guardian is created is the node where it will exist for its lifetime. It must have been created by (a process in) a guardian at that node. Each node comes into existence with a *primal* guardian, which can (among other things) create guardians at its node in response to messages arriving from guardians at other nodes. This restriction on creation of new guardians helps preserve the autonomy of the physical nodes.

A guardian is an abstraction of a physical node of the underlying network: it supports one or more processes (abstract processors) sharing private memory, and communicates with other guardians (abstract nodes) only by sending messages. In thinking about a distributed program, a programmer can conceive of it as a set of abstract nodes. Intra-guardian activity is local and inexpensive (since it all takes place at a single physical node); inter-guardian processing is likely to be more costly, but the possibility of this added expense is evident in the program structure. The programmer can control the placement of data and programs (a consequence of autonomy as discussed above) by creating guardians at appropriate nodes. Furthermore, each guardian acts as an autonomous unit, guarding its resource and responding to requests as it sees fit.

2. Robustness

A major problem in distributed programs is how to achieve robust execution of atomic operations in spite of failures. (An atomic operation is either entirely completed or not done at all.) This is an area where distributed programs are likely to differ significantly from centralized programs. Not that the need for robustness is new; rather, the issue has been largely ignored in centralized systems, with the exception of some work in data base systems.

One requirement for robustness is *permanence of effect*. Permanence means that the effect caused by a completed atomic operation (e.g., a change in the state of the resource owned by the guardian that performs the operation) will not be lost due to node failures.

To achieve permanence requires a finer grain of backup and recovery than is provided by occasional system dumps and automatic system restart. We believe that permanence must be provided by each guardian for the resource it guards. We expect that backup and recovery will be provided on a per guardian basis: processes in the guardian save recovery data as needed (by, e.g., logging it in storage that will survive a node crash), and the guardian provides a recovery process that is started after a node crash to interpret the recovery data.

3. Discussion and Examples

The guardian construct was invented to satisfy the modularity criteria given above. The purpose of a guardian is to provide a service on a resource of a distributed program, but in a safe manner, i.e., it guards the resource by properly coordinating accesses to it, by protecting the resource from unauthorized access, and by providing backup and recovery for the resource in case of node failures. The resources being so guarded may be data, devices or computation.

For example, the flight data for an airline might be guarded by a single guardian that handles reservations for all flights and also provides a number of administrative functions such as deleting or archiving information about flights that have occurred, collecting statistics about flight usage, etc. It responds to requests such as "reserve," "cancel," "list passengers," and so on. For such requests, it checks that the requestor has the right to request the access (perhaps using some sort of access control list mechanism [17]). For example, only a manager can request a passenger list, and a reservation request from some other airline might not be permitted to reserve the last seat on a flight. The guardian guarantees that requests are properly coordinated, for example, performed in an order approximating the externally observable order in which they were requested. It performs the reserve and cancel requests as atomic operations, and logs them so that information will not be lost if the node fails.

Internally, the airline guardian might make use of a guardian for each flight: The top level guardian simply dispatches requests to the appropriate flight guardian, which does the actual work and logs results. A flight guardian might be organized in several different ways, for example:

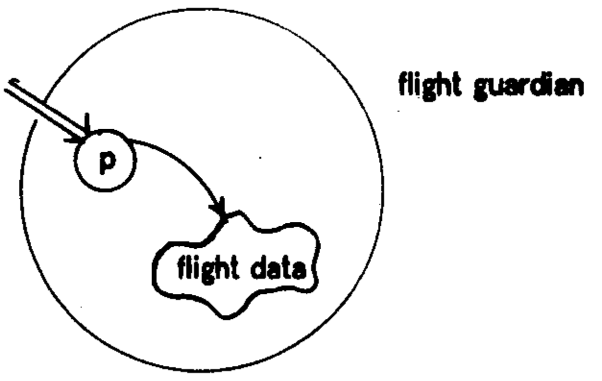
1. A single process handles requests one at a time (Figure 1a).

2. Requests for different dates are permitted to proceed in parallel. A single process synchronizes requests; it hands them off to other processes that perform the actual work (Figure 1b) when the flight data of interest are available. Such a structure is similar to that provided by a serializer [10].

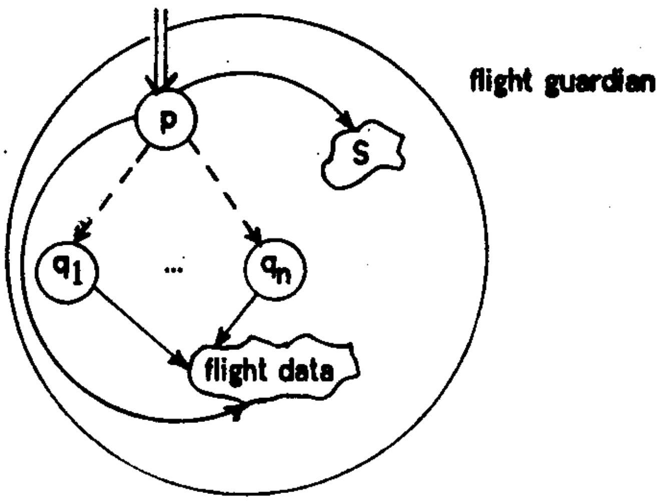
3. A single process receives a request and immediately creates a process to handle it (Figure 1c). The forked processes synchronize with each other to ensure that only one process is manipulating the data for a particular date at a time. The processes synchronize using shared data, e.g., a monitor [11] providing operations *start_request(date)* and *end_request(date)*.

Organizations 2 and 3 can provide concurrent manipulation of the data base, while organization 1 cannot.

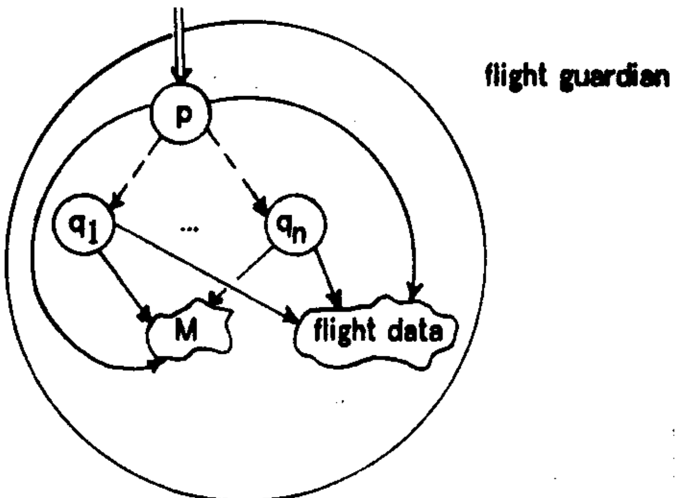
Fig. 1. Possible organizations for a flight guardian.



a. One-at-a-time solution: process p handles requests sequentially.



b. Serializer solution: process p uses synchronization data S to determine when requests should be performed. It forks processes q_i to do the actual requests.

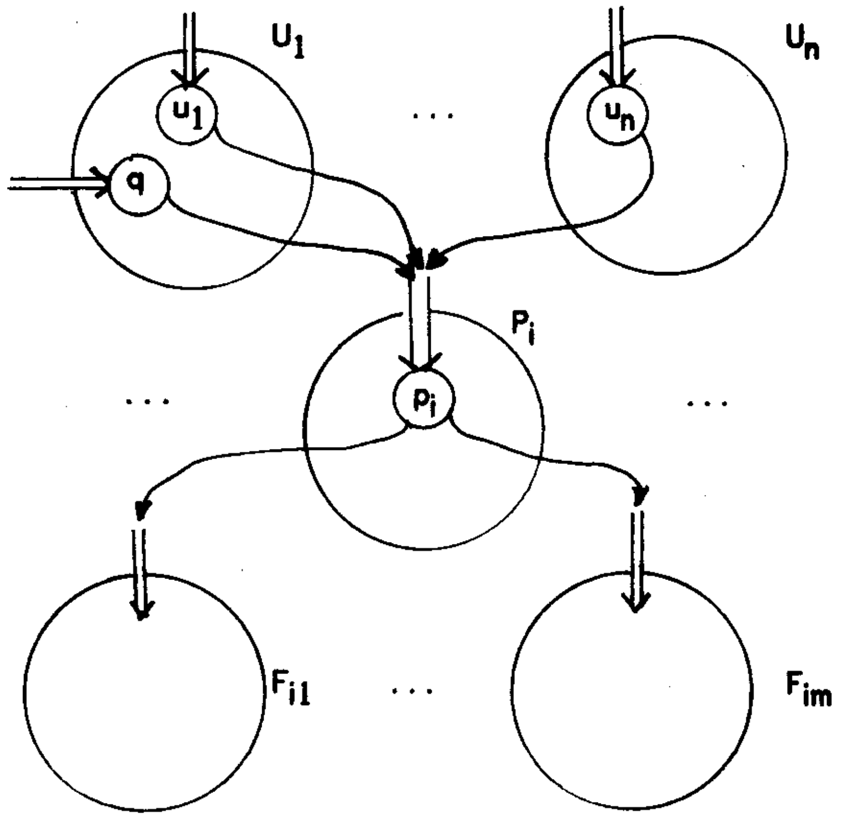


c. Solution using a monitor: process p forks a process q_i upon receipt of a request. The processes q_i synchronize with each other using monitor M and perform the requests on the data base.

The airline data base discussed above had a single top level guardian. Alternatively the data base might be distributed; for example, it might be divided into partitions for different geographical regions, each residing at a distinct node, and the guardian for a flight assigned to the region containing the flight's destination. Such a structure is shown in Figure 2. Here each node belonging to the airline has one guardian, P_i, for the region in which it resides, and one guardian, U_i, to provide an interface to the airline data base for that node's users (e.g., reservation clerks and administrators). A user makes a request to the U_j at his node; some checking for access rights would be done here, and then one or more requests sent to the appropriate P_i. The P_i would dispatch these requests to the flight guardians for its region. The P_i and U_j would

Fig. 2. Distributed airline system example.

There are n front ends (guardians U_1 to U_n) and n regional managers (only one, guardian P_i is shown) that communicate with the guardians of flights in its region (guardians F_{i1}, ..., F_{im}). Process q in U_1 is carrying out a transaction for a user. Processes u_i are ready to accept requests from new users.



coordinate as needed by means of some protocol established for that purpose. A possible organization for the U_j might be to fork a process to handle a transaction consisting of many requests; this process would carry out U_j's end of the coordination protocol. This process might, for example, interact with a clerk to make a number of reservations for the same customer.

In the organization shown in Figure 2, each guardian U_j guards the entire airline data base and provides transactions that consist of sequences of requests. Each guardian P_i guards the data for a geographical region, while each flight guardian guards the data for a single flight. Thus, access to the entire distributed data base is provided by a group of guardians, but each guardian in that group guards a discernable resource.

It is appealing to imagine a system structure in which processes do not share any data. Although multi-process guardians are not necessary for computational power, we permit many processes in a guardian for two main reasons: concurrency (e.g., Figures 1b and 1c) and conversational continuity. Concurrency could be obtained by having guardians that guard very small resources (e.g., the information about a single flight and date, or a single record in a data base), but we felt such a structure would often be unnatural. Conversational continuity is illustrated in Figure 2: process q carries on a conversation with the user and the "state" of this conversation (e.g., the identity of the passenger for whom reservations are being made and the reservations made so far) is captured naturally in the state of process q.

D. New CLU Implementation

With the current CLU implementation in a fairly stable state, we have begun work on a new implementation for the LCS Advanced Node. This effort has four major goals: to produce a more efficient implementation, to produce a much more portable implementation, to provide a basis for implementing the extensions to CLU discussed in the preceding section, and to provide a modern programming system for laboratory-wide use.

To produce a more space-efficient implementation, we have decided to use a static linker and loader, at least initially, rather than copy the dynamic (re)linking and (re)loading capabilities of the current implementation [1]. Space is saved this way because the linker, loader, and associated data structures are not in memory at execution time. Although this decision eliminates a very useful debugging aid, it should be possible eventually to reintroduce some form of dynamic relinker/reloader running as a separate process.

In an attempt to produce a more time-efficient implementation, we have carefully redesigned the procedure call mechanism, the iterator mechanism, the parameter mechanism, and the exception handling mechanism. Whereas these mechanisms were designed for the current implementation primarily to allow simple, uniform code generation, for the new implementation we have tried to optimize the most common cases, without sacrificing speed in other cases. For the most part, these optimizations reduce the number of registers that must be maintained across procedure and iterator boundaries. In addition, we have decided not to maintain run-time type codes, which exist in the current implementation primarily for redundant run-time type-checking.

A major problem with the current implementation is that most of the run-time support system is written in assembly language. This not only complicates maintainance of the system, but also makes the system difficult to transport to a different machine, or even the same machine under a different operating system. We hope to rectify this problem with the new implementation. Although we have been proceeding under the assumption that the initial target machine is the Zilog Z-8000, it should also be possible to bring the system up on a VAX 11/780, an M68000, a 360/168, or a PDP-10 with a minimum of effort. The primary assumptions are that the machine supports values of at least 32 bits (for object references), a fairly large (2^{16} values or more) linear address space, an efficient stack mechanism for object references, and a small number of registers (for object references).

Easy portability is obtained by defining a small number of data types (object references, integers, tagged cells, byte vectors, and vectors of object references) that are not completely type-safe, and a few I/O primitives, and then implementing everything else in CLU on top of them. (With certain extensions to CLU, even records and oneofs can be implemented in CLU.) To bring up an implementation on a different machine, it should only be necessary to implement a few modules in assembly language, produce a new code generator for the compiler, and perhaps modify a few code-dependent modules of the linker.

The major components of the new implementation that need to be constructed are the basic data types of CLU, the garbage collector, the linker and loader, the code generator, and various debugging tools. All of these components have been designed and, with the exception of the debugging tools, are partially implemented and tested for the Z-8000.

E. Evaluating Synchronization Constructs

When facilities for concurrent programming are added to a language, it is essential that these extensions support the construction of reliable, easily maintained software. Much attention has been given to developing high-level programming language synchronization constructs, such as monitors [11], path expressions [6], and serializers [10], for specifying and controlling access to shared resources. Unfortunately, the requirements that these constructs must meet are not fully understood. Properties such as expressive power, ease of use, modularity, and modifiability are agreed to be important, but the definitions of these terms are so vague that evaluation according to these criteria is difficult.

Most attempts at evaluating these constructs have centered around attempting to implement solutions to various examples of synchronization problems, and then deriving intuitive judgments about the expressiveness and usability of a construct. Bloom [3] has developed evaluation methods that allow such information to be deduced from examples in more structured ways, and has applied the method to several existing mechanisms. First, synchronization problems have been examined and categorized according to properties that affect how easily these problems can be handled by synchronization constructs. This categorization defines the range of problems synchronization mechanisms must be able to handle, and provides a means of selecting a set of test cases adequate for evaluating the mechanisms. Making use of these test cases, mechanisms can be evaluated, with the methods developed, for expressive power, ease of use, and modifiability. Because the properties of interest are so vague, completely objective methods of evaluation are impossible. However, the categorization makes it possible to obtain information from test cases that will provide substantial assistance in locating and correcting weaknesses, as well as in generalizing results from one test case to sets of problems with similar properties.

In the following sections, we first discuss modularity requirements. Then the categorization of synchronization problems is presented, followed by an explanation of techniques for evaluating synchronization constructs based on this categorization. These techniques have been applied to monitors, path expressions, and serializers, and a summary of this evaluation is given in the final section.

F. Modularity Requirements

The first property to be defined is modularity, and how modularity applies to the structure of shared resources; the remainder of the analysis assumes shared resources are properly modularized. The model of shared resources assumed here is based on the use of abstract data types [15]. Resources are considered to be objects of abstract types. A resource will therefore have a set of operations associated with it, and access to the resource will be possible only by invoking one of those operations.

There are two modularity requirements that should be satisfied by concurrent programs accessing shared resources. The first follows from the principle that the definition of an abstraction should be made independent of its use. The shared resource abstraction should thus contain all of the synchronization code, as well as the resource definition. This structure will guarantee that users of the resource can assume it to be properly synchronized; no synchronization code need be located at each point of access to the resource.

The other modularity requirement governs the structure of the shared resource definition. The module implementing the shared resource serves two purposes: to define the abstract behavior of the resource, independent of whether concurrent access is allowed, and to provide the synchronization for controlling access. These two parts actually serve different functions and should be separable into two subsidiary abstractions, the unsynchronized resource, and the synchronization.

1. Categorizing Synchronization Problems

Synchronization mechanisms serve two main functions with respect to shared resources. One is excluding certain processes from the resource under given circumstances; the other is scheduling access to the resource according to given priorities. Synchronization schemes are thus composed of two types of constraints. Exclusion constraints take the form:

if condition then process A is excluded from the resource

and priority constraints take the form:

if condition then process A has priority over process B

Within these two main classes, constraints differ in the kinds of information used in the conditional clause. The information that can appear falls into several categories:

1. The access operation requested. Stating, for instance, that readers of a data base have priority over writers involves giving a constraint in terms of the types of operations requested. In contrast, a strict first-come first-serve ordering uses no information about the operations requested.

2. The times at which requests were made. Though it is rarely necessary to know the exact time of a request, the order of requests relative to other events is often important. Time information is used for this purpose.

3. Request arguments. In many cases, the arguments passed with a request are needed to determine the order in which processes should be admitted to the resource.

4. Local resource state. Local state includes information that would be present independent of access control. For example, whether a buffer is full or empty.

5. Synchronization state. Synchronization state includes only state information needed for synchronization purposes. This information would not be part of the resource state were the resource not being accessed concurrently. Included in this category is information about the processes currently accessing the resource, and the operations those processes are executing.

6. History information. This information differs from synchronization state in that it refers to resource operations that have already completed, as opposed to those still in progress. It is often interchangeable with local state information, since the interesting past events are most likely to be those that leave some noticeable change in the state of the resource.

By way of two simple examples, a first-come first-serve ordering scheme has a single priority constraint of the form:

```
process A has priority over process B
                if time_of_request(A) < time_of_request(B)
```

Thus, only request time information is used. A readers priority constraint has the form:

```
process A has priority over process B
                if request_type(A) = read & request_type(B) = write
```

More complete examples can be found in [3].

This identification of properties of synchronization problems enables more informative conclusions to be drawn from the evaluation of synchronization problems. In addition, the categorization aids in the selection of test cases. By selecting a set of problems that cover all the constraint types, and examining the ways in which each kind of information is handled by a particular mechanism, general conclusions can be drawn about the mechanisms' ability to implement synchronization problems that make use of various types of information.

2. Evaluation Criteria

To be sufficiently powerful, a synchronization mechanism must satisfy two basic requirements. First, it must be expressive, by providing a straightforward means of stating individual constraints, and by providing the ability to express those constraints in terms of any of the information types described earlier. Second, when implementing complex synchronization schemes composed of many constraints, it must be possible to implement each constraint independently. In the following sections we explain these criteria more precisely, and discuss methods for assessing how well they are supported by various synchronization constructs.

2.a. Expressive Power

One way to test expressive power of a synchronization mechanism is to use it to implement solutions to a set of examples that cover all information classes. A sample set is presented in [3]. If there is no direct way to use a certain kind of information, it should become obvious when an attempt is made to implement a solution requiring it. By examining how various types of information are handled in each solution, conclusions can be drawn as to the ease with which the mechanism can access each type of information.

A more general way to measure expressive power is simply to examine the mechanism and attempt to determine what features it has that will enable each type of constraint to be dealt with. For example, monitor queues are a construct for handling request time information, while serializer crowds retain synchronization state information. Some data manipulation technique must be available for each type of information. The ability to identify the particular way in which each information type is handled will also make a mechanism easier to use, because the structure of a solution will be indicated by the kinds of information used in the specification.

2.b. Combining Constraints in Complex Solutions

Whether or not a mechanism is easy to use depends not only on the ability to easily construct solutions to individual constraints, but on the ability to easily construct complex synchronization schemes made up of many such constraints.

Complex schemes will be easy to implement only if they can be decomposed into individual constraints that can then be realized independently. As the number of constraints increases, solutions quickly become difficult to construct if the implementation of any one constraint depends on another; the complexity of constructing the solution increases more than linearly with the number of combinations of constraints present. In addition to the difficulty of initially constructing solutions, the solutions will be difficult to modify if this constraint independence property is not met: a change in the specification of one constraint may necessitate reimplementation of the entire solution.

One way to test whether a mechanism allows independent implementation of constraints is to examine solutions to two similar synchronization problems. If the problems share some constraints, but differ in others, then the implementation of the common constraints should be similar in both solutions. If differences appear in the way a given constraint is implemented in two different synchronization problems, or if the implementation of each individual constraint is not even identifiable as a separate part of the solution, then this indicates that the independence criterion is being violated. If constraint implementations are really independent, a given implementation should remain the same when other constraints are modified to use different types of information. A complete evaluation involves checking all possible pairs of the six information types for conflicts. Two sample problems for use in this analysis are given in [3].

3. Evaluation of Existing Mechanisms

The three existing mechanisms that seem most likely to satisfy the requirements of good software engineering are monitors, path expressions, and serializers. Based on our evaluation, we have drawn the following conclusions about these three mechanisms. While the approach taken by path expressions seems very attractive, our analysis has revealed some serious shortcomings. Path expressions do not provide easy access to several types of information needed in synchronization constraints, and thus lack sufficient expressive power. In particular, it is difficult to use the local resource state and the arguments of operations. To maintain information about time of request, or to express priority constraints in general, requires additional synchronization procedures, thus increasing the solution's complexity. In addition, the constraint independence requirements necessary to ensure ease of use are not well supported by the mechanism.

Both monitors and serializers satisfy our criteria reasonably well. Based on our evaluation, we prefer serializers over monitors. Though certain tradeoffs are involved in selecting one of these mechanisms over the other, serializers seem superior in two important respects. First, serializers meet our modularity requirements more closely. The proper use of monitors requires a special protected-resource module in addition to the synchronizer and resource modules; the resource implementor must also follow specific guidelines for defining monitor operations. Serializers depend less on such rules: the protected-resource module is not needed, and serializer operations are precisely the user-accessible operations on the protected resource. The other important distinction between these two mechanisms is the use of automatic signalling in serializers. Though proof rules for the monitor signal construct have been developed, an automatic signalling feature is more likely to aid in constructing correct programs, and in easing the burden placed on the verifier.

G. CLU Machine Architecture

The design of computers is strongly influenced by the characteristics of available technology. Until recently, computers have been designed under the constraint that processing hardware is expensive. However, the cost of hardware is continually decreasing and the significant cost of software has become even more apparent. Therefore, it seems appropriate to consider how hardware technology can be used to implement modern programming languages and to reduce the complexity of computer systems.

Snyder [19] has designed a computer system that directly supports an object-oriented machine language. Unlike most implementations of object-oriented languages on conventional machines, which provide a separate and usually small space of objects for each process, the proposed system provides a single, very large space of objects shared by all processes in the system. This space of objects would include not only temporary objects used during the execution of programs, but also such long-lived objects as the procedures and data normally stored in a file system, with uniform access to all objects.

The primary goal of the research was to design a machine that supports such a large universe of objects effectively. A second goal was to minimize the complexity of the design. To accomplish these goals, two assumptions were made about future technology.

The first assumption is that processors are sufficiently inexpensive that several processors can be used where one is used today. Multiple processors are used both to obtain greater modularity, and to improve performance. When necessary, processor utilization may be decreased in order to increase system throughput.

The second assumption is the existence of secondary storage devices with access times on the order of 100 microseconds (about 100 times faster than current disks). This assumption is motivated by the expected small average size of objects (on the order of 4 to 20 words), based on measurements of existing programs. Fast-access devices are needed to obtain good multi-level memory performance without introducing undue complexity.

Both of these assumptions appear to be reasonable. It is widely predicted that LSI processors equivalent to current main frames will be developed in the next decade; the cost of these processors will be quite low compared to the total cost of a computer system. The access time figure of 100 microseconds is within the predicted range for charge-coupled devices and electronic beam memories; however, projections do show that such memories may be an order of magnitude more expensive than disk memories.

Below we provide a brief description of this work. The first section focusses on the overall system structure, and the second section explains the storage reclamation mechanism.

1. System Structure

The system is constructed hierarchically out of a number of specialized processor modules communicating via messages. Each module performs well-defined functions. At the top-most level, the system is divided into two major modules, the processing module (PM) and the memory module (MM). The PM interprets procedures and implements multiple processes. It consists of a number of instruction processors, which interpret procedures, plus a control processor, which performs scheduling and controls the multiplexing of the instruction processors. Each instruction processor has its own local memory which it uses in the interpretation of instructions. In addition, this local memory is used as a cache to reduce the number of accesses to the MM. For example, much of the evaluation stack would certainly be held in this local memory.

The MM implements the universe of objects with a multi-level memory system. An object is simply a vector of references to other objects (although certain values, such as booleans and integers, are stored directly in a reference). The MM encapsulates all knowledge of how objects are implemented, including storage allocation and automatic storage reclamation. The interface between the PM and the MM basically consists of invocations of operations of the vector data type.

Briefly, each object is represented by a single "page"; the system supports a number of different page sizes. Objects (pages) are identified by their secondary storage addresses and are transferred individually between primary and secondary storage. A large set associative memory maps from the secondary storage addresses of objects in primary storage to their primary storage addresses. The set associative memory is implemented using ordinary random access memory, with a small, fast, expensive associative memory used as a cache. Physical storage is divided into fixed-size blocks; each block is (statically or dynamically) divided into pages of a single size. Storage is further divided into a number of zones; each zone provides pages of a single size and contains its own list of free pages.

2. Storage Reclamation

To maximize secondary storage utilization, the storage used by an object should be reclaimed as soon as possible after the object becomes inaccessible. An important contribution of this research has been the development of a simple, efficient automatic storage reclamation scheme that is performed continuously, without requiring frequent or unpredictable interruptions of service.

To facilitate storage reclamation, there is one additional interaction between the PM and MM. The PM must cooperate with the MM in order for the MM to determine which objects are needed and which can be reclaimed. In particular, at certain times the MM will request the PM to discard all of its object references. When there are no object references outside the MM, the system is said to be in *quiescence*. During quiescence, the MM can examine the entire collection of accessible objects without interference from the PM. When the MM is finished, the PM reads back all needed data and resumes normal operation.

The storage reclamation algorithm is based on reference counts. The basic idea of reference counts is to associate with each object a counter to record the number of existing references to that object. When an object is created, a single reference to the object is created, and the reference count of the object is set to one. Whenever a reference to the object is copied or destroyed, the reference count is incremented or decremented, respectively. The object can be reclaimed whenever the reference count reaches zero (destroying all contained object references and decrementing the corresponding reference counts). Of course, if a group of objects contains a cycle of references, none of the objects in the group can be reclaimed in this way, even if the entire group is inaccessible. Similarly, if a bounded reference count ever reaches its maximum value, it must remain there, lest the object be reclaimed prematurely.

The biggest problem with a conventional reference count scheme is that reference count events occur at an enormous rate: each time a reference to a storage object is copied or destroyed, a reference count must be updated. A reference count scheme can be made very efficient by not requiring every reference in the system to be accounted for. Instead, reference counts only count references stored as components of objects in the MM; references outside the MM, or on their way in or out of the MM, are not counted. This substantially reduces the number of events that cause reference count operations: only events that change the contents of objects in the MM cause reference count operations. Manipulations of references within the local memory of the PM do not change reference counts. Since most of a process' evaluation stack can be cached into such local memory, changes to the evaluation stack generally cause no reference count operations.

Of course, under this scheme an object cannot be reclaimed just because its reference count has become zero; rather, the system must be forced into quiescence in order to reclaim objects. In quiescence, one can locate objects with zero reference counts by scanning the entire memory, but a much more efficient algorithm is possible.

During normal operation (not during quiescence), whenever the reference count of an object X becomes zero, an entry "discard(X)" is added to a queue of suspected garbage GQ. Further, whenever the reference count goes from zero to non-zero, an entry "resurrect(X)" is added to the GQ. Then, when quiescence is established, the GQ can be used to determine precisely which objects can be reclaimed: an object X can be reclaimed only if the last entry for it on the GQ is of the form "discard(X)". This information is totally contained within the GQ; there is no need to examine the actual reference counts of objects. More importantly, it is not necessary to keep the system in quiescence while the GQ is being processed. Once quiescence is established, normal system operation can be resumed with a new, empty GQ, with the old GQ being processed concurrently.

Forcing the system into quiescence is similar to swapping out all running processes. In general, there will be some minimum rate of process switching needed anyway to maintain interactive response with the users. As long the the cycle time between quiescent states is longer than the process time quantum, there need be little performance degradation.

In addition to the storage reclamation process just described, it will be necessary to perform infrequent, periodic garbage collection for the purpose of reclaiming cyclic garbage. Suitably designed, the garbage collection time should be on the order of ten minutes [19]. Clearly any time of this magnitude is acceptable for infrequent garbage collections scheduled on the order of once per day or once per week.

H. Specifications of Mutable Abstractions

Berzins [2] has investigated specifications for data abstractions based on the *explicit* or *abstract model* appproach. In this technique, a data abstraction is defined in terms of simple, well-known abstractions (e.g., integers, mathematical sets), and possibly some other user-defined abstractions. The main innovation of this work is that it extends previous work to handle both potentially shared, mutable data objects, and operations that can raise exceptional conditions.

In the sections below, we give an informal description of this work, concentrating on the extension to mutable objects. We first give an example specification of a mutable data type, integer sets, to present the specification language and explain the major concepts of the technique. Following the example, we discuss the standard model defined by a specification. We then present an implementation of integer sets, and show how to prove that an implementation model is equivalent to the standard model. We conclude with a discussion of how the approach fits in with program verification.

1. Example Specification

A specification of finite, mutable sets of integers is shown in Figure 3. The first line of the specification states that we are defining a type named "intset," and introduces "I" as an abbreviation for "intset." The with clause gives the names and functionality of the operations. For example, "empty" takes no arguments and returns an *intset*, and "remove" takes two arguments and returns no results (it mutates its first argument).

Because we are specifying a mutable data type, the actual model is based on *system states*, which are indexed sets of *data states*. Each data state represents the state of some *intset* object. Formally, if *s* is a system state and *x* is an *intset token* (a formal identifier for an *intset* object), then *s*(*x*) is the state of the particular *intset* object *x* in the system state *s*. The actual functionality of each operation has an extra argument (the input system state) and an extra result (the output system state) besides those shown in the with clause of the specification. The definitions of immutable data abstractions are, of course, simpler.

As types are built up in a hierarchy, the system state becomes a set of indexed sets, with one indexed set of data states for each mutable type used in the type's operations. The formal structure that is built is a heterogeneous algebra.

The data states for *intset* are represented by mathematical sets of integers; the restrictions clause indicates that only finite sets can be used. The identity clause defines an equivalence relation on the restricted data state domain; the objects of the

Fig. 3. Specification for Intset

```
type intset as I

with     empty:                  --> I
         insert:      I x int    -->
         remove:      I x int    -->
         has:         I x int    --> bool

data states   D = sets of integers
restrictions  d such that cardinality (d) ∈ N
identity      equal

operations    empty(s)( ) = extend (s, { })
              insert(s) (x, i) = update (s, x, s(x) ∪ {i})
              remove (s) (x, i) = update (s, x, s(x) ∪ {i})
              has (s) (x, i) = <s, i ∈ s(x)>

end intset
```

algebra are precisely the equivalence classes so formed. In this case the identity is trivial (set equality).

The operations clause of the specification defines the effects of the operations. Here the system state is represented explicitly. Also, to emphasize the special nature of the system state, the operations are written in "curried" form; e.g., we write "insert(s)(x, i)" instead of "insert(s, x, i)". The operations are defined in terms of functions on the system state: "extend" creates a new object with the given initial data state, and returns both a system state with the new object added and a token for that new object; "update" returns a new system state with the data state of the object named by its second argument changed to be the data state given as the third argument.

2. Standard Model

So far the presentation has been largely intuitive; we have not said much about the formal object that is defined by a specification. To the logician, a model is a mathematical structure satisfying some set of axioms; the existence of a model shows the set of axioms to be consistent. A specification can also be viewed as a set of axioms; the heterogeneous algebra defined by the specification is a model for that axiom system. This model is called the *standard model.*

Since the specification language is powerful enough to describe inconsistent specifications (any useful language has this property), we must, in general, prove the existence of a standard model. This is done by proving inductively that no operation constructs a data state outside the restricted data state domain (when provided with arguments in that domain), and that all operations preserve the equivalence relation on data states (i.e., equivalent arguments produce equivalent results). In our example, the proof is trivial, but in more complicated types this may require considerable effort.

3. Example Implementation

The specification language also has features for defining implementations. Figure 4 gives an implementation of *intset* in terms of arrays of integers.

The representation clause serves both to identify what type is being implemented, and to describe the representation domain. Arrays are mutable vectors to which items can be added or removed at either end, and they are indexed by a contiguous subset of the integers. For an array a, low(a) is its current low bound, high(a) is its current high bound, and if low(a) $\leq i \leq$ high(a), then $a[i]$ names a valid element of a.

The restrictions clause states which elements of the representation domain are legal representations of objects. Here the arrays are restricted to have a low bound of 1, and to not have any duplicate elements. The identity clause states that identity of an *intset* object is to be represented by the identity of the representing array. Note that this clause defines object identity, *not* state identity; two distinct objects may have the

Fig. 4. An Implementation of Intset

```
representation    intset = array[int]

restrictions      a such that: low(a) = 1 & (low(a) ≤ j < k ≤ high(a) ==> a(j) ≠ a(k))
identity          array$equal

operations        empty( ) = array[int]$create(1)
                  insert (a, i) = if ~has (a, i) then addh (a, i)
                  remove (a, i) = if has (a, i) then {store (a, find(a, i), a[high(a)]); remh(a)}
                  has (a, i) = find(a, i) > 0

definition  find(a, i) = if (∃j) [low(a) ≤ j ≤ high(a)  &  a[j] = i]
                                          then j : low(a) ≤ j ≤ high(a)  &  a[j] = i
                                          else 0
```

same state. For mutable objects, identity is almost always identity of representation objects, as it is in this case. Identity for immutable objects can be more complicated, particularly if the representataion is itself mutable.

The operations clause defines the operations in terms of the representation domain; each operation is defined by a program in a simple programming language. This language permits ∃, ∀, and ":" (meaning "such that") to be used for clarity and to avoid overspecification. The operations are implemented in terms of array operations: "create" returns a new, empty array with the given low bound; "addh" adds an element to the high end of an array; "remh" removes and returns the high element; "store" updates the element at the specified index with the given value. Some of these operations may raise exceptional conditions; in this example it must be proved that they do not. The definition clause is used for defining "helping routines" to simplify the description of the implementation.

4. Behavioral Equivalence

An implementation defines a model, in a manner very similar to the way a specification defines the standard model. To show that we have a valid implementation of an abstraction, we must demonstrate that the implementation simulates the required abstract behavior. The particular simulation desired is *behavioral equivalence*: informally, this means that if the implementation model is substituted for the standard model in any program, the program will produce the same results.

Behavioral equivalence is proved inductively on the length of the computations performed by the two programs (one with the standard model, one with the implementation model). At each step in the computation we must exhibit a correspondence relation between the simulated objects and the simulating ones. It is this relation that describes exactly how the implementation objects simulate the abstract objects. As such, its purpose is very similar to Hoare's abstraction function [12].

The simulation relation for our example is as follows, where "abstract" refers to the standard model and "concrete" refers to the implementation model:

For each	i	abstract integer,
	i', j'	concrete integer,
	s	intset abstract system state,
	s'	intset concrete system state,
	x	abstract intset,
	x'	concrete intset

$$s \Longleftrightarrow s' \;\&\; x \Longleftrightarrow x' \;\&\; i \Longleftrightarrow i') \Longrightarrow$$
$$(i \in s(x) \;\equiv\; (\exists j')\,[\mathrm{low}\,(s'(x')) \leq j' \leq \mathrm{high}(s'(x'))\;\&\; i' = s'(x')\,[j']])$$

This says that given corresponding system states (s, s'), *intset* objects (x, x'), and integers (i, i') in the two models, then an integer is an element of the *intset* in one model if and only if the corresponding integer is an element of the corresponding *intset* in the other model.

5. Program Verification

Proving the correctness of a data type implementation with respect to a standard model is only half of the process required to verify programs that use data abstractions. The other half of the process involves proving the correctness of programs that invoke the type's operations. The intended behavior of a program is typically described by inserting assertions at various points in the program. The assertions express the relations that must hold between the data objects manipulated by the program. For programs that use data abstractions, the assertions are written in terms of the primitive operations of the abstractions. For dynamic abstractions, the system state must be explicitly included in the assertions, so that the operations can be treated as pure functions.

The problem of showing that a program satisfies its assertions can be reduced to the problem of proving theorems about the data abstractions it uses, by eliminating the program text from the correctness requirements with an axiomatic definition of the control constructs in the programming language. The derived theorems, which must be proved in order to establish correctness, are called verification conditions. Proving verification conditions based on an abstract model approach presents no methodological problems. It is sufficient to prove the interpretations of the verification conditions in the standard models of the data abstractions used by the program, since behavioral equivalence guarantees the same results in all correct implementations of those abstractions.

REFERENCES

1. Atkinson, Russell R.; Liskov, Barbara; and Scheifler, Robert W. "Aspects of Implementing CLU." Proceedings of the ACM 1978 Annual Conference, December 1978, 123-129.

2. Berzins, Valdis A. Abstract Model Specifications for Data Abstractions. M.I.T., Laboratory for Computer Science, LCS/TR-221. Cambridge, Ma., July 1979.

3. Bloom, Toby. Synchronization Mechanisms for Modular Programming Languages. M.I.T., Laboratory for Computer Science, LCS/TR-211. Cambridge, Ma., April 1979.

4. Brinch Hansen, Per. "Distributed Processes: A Concurrent Programming Concept." Communications of the ACM, Vol. 21 No. 11 (November 1978), 934-941.

5. Brinch Hansen, Per. "The Programming Language Concurrent Pascal." IEEE Transactions on Software Engineering, Vol. 1 No. 2 (June 1975), 199-207.

6. Campbell, Roy H., and Habermann, A. Nico. "The Specification of Process Synchronization by Path Expressions." Lecture Notes in Computer Science, Vol. 16. Springer Verlag, 1974.

7. Feldman, Jerome A. A Programming Methodology for Distributed Computing. University of Rochester, Department of Computer Science, Technical Report 9. Rochester, N. Y., 1977.

8. Fuller, Samuel H., et al. A Collection of Papers on CM*: A Multi-microprocessor Computer System. Carnegie Mellon University, Department of Computer Science, February 1977.

9. Hewitt, Carl E. "Viewing Control Structures as Patterns of Passing Messages." Artificial Intelligence, Vol. 8, 1977, 323-364.

10. Hewitt, Carl E., and Atkinson, Russell R. "Specification and proof techniques for serializers." IEEE Transactions on Software Engineering, Vol. SE-5 No. 1 (January 1979), 10-23.

11. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." Communications of the ACM, Vol. 17 No. 10 (October 1974), 549-557.

12. Hoare, C. A. R. "Proof of Correctness of Data Representations." Acta Informatica, Vol. 1 No. 4 (1972), 271-281.

13. Liskov, Barbara. Primitives for Distributed Computing. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 175. Cambridge, Ma., May 1979.

14. Liskov, Barbara; Moss, Eliot; Schaffert, J. Craig; Scheifler, Robert W.; and Snyder, Alan. CLU Reference Manual. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 161. Cambridge, Ma., July 1978.

15. Liskov, Barbara; Snyder, Alan; Atkinson, Russell; and Schaffert, J. Craig. "Abstraction mechanisms in CLU." Communications of the ACM, Vol. 20 No. 8 (August 1977), 564-576.

16. Peterson, James L. "Notes on a Workshop on Distributed Computing." Operating Systems Review, Vol. 13 No. 3 (July 1979), 18-30.

17. Saltzer, Jerome H., and Schroeder, Michael D. "The Protection of Information in Computer Systems." Proceedings of the IEEE, Vol. 63 No. 9 (September 1975), 1278-1308.

18. Scheifler, Robert W. "An Analysis of Inline Substitution for a Structured Programming Language." Communications of the ACM, Vol. 20 No. 9 (Sept. 1977), 647-654.

19. Snyder, Alan. A Machine Architecture to Support an Object-Oriented Language. M.I.T., Laboratory for Computer Science, LCS/TR-209. Cambridge, Ma., March 1979.

Publications

1. Atkinson, Russell R.; Liskov, Barbara H.; and Scheifler, Robert W. "Aspects of Implementing CLU." Proceedings of the ACM 1978 Annual Conference, December 1978, 123-129.

2. Bloom, Toby. Synchronization Mechanisms for Modular Programming Languages. M.I.T., Laboratory for Computer Science, LCS/TR-211. Cambridge, Ma., April 1979.

3. Clark, David C.; Greif, I. G.; Liskov, Barbara H.; and Svobodova, Liba. Semantics of Distributed Computing, Progress Report of the Distributed Systems Group. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 171. Cambridge, Ma., October 1978.

4. Greif, Irene G., and Meyer, Albert. "Specifying Programming Language Semantics: A Tutorial and Critique of a Paper by Hoare and Lauer." Proceedings of the Principles of Programming Languages Conference, San Antonio, Texas, January 1979.

5. Greif, Irene G., and Meyer, Albert. "Specifying the Semantics of While-Programs: A Tutorial and Critique of a Paper by Hoare and Lauer." M.I.T., Laboratory for Computer Science, LCS/TM-130. Cambridge, Ma., April 1979.

6. Kapur, Deepak. "Specifications of Majster's Traversable Stack and Veloso's Traversable Stack." SIGPLAN Notices, Vol. 14 No. 5 (May 1979), 46-53.

7. Kapur, Deepak, and Mandayam, Srivas. Expressiveness of the Operation Set of a Data Abstraction. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 179. Cambridge, Ma.: June 1979.

8. Laventhal, Mark S. Synthesis of Synchronization Code for Data Abstractions. M.I.T., Laboratory for Computer Science, LCS/TR-203. Cambridge, Ma., July 1978.

9. Liskov, Barbara H.; Moss, Eliot; Schaffert, J. Craig; Scheifler, Robert W.; and Snyder, Alan. CLU Reference Manual. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 161. Cambridge, Ma., July 1978.

10. Liskov, Barbara H. Practical Benefits of Research in Programming Methodology. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 166. Cambridge, Ma.: August 1978.

11. Liskov, Barbara H. Primitives for Distributed Computing. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 175. Cambridge, Ma.: May 1979.

12. Liskov, Barbara. "Remarks on the Construction of Large Programs." The Impact of Research on Software Technology. Edited by P. Wegner. Cambridge, Ma.: M.I.T. Press, June 1979, 345-351.

13. Liskov, Barbara, and Berzins, Valdis. "An Appraisal of Program Specifications." The Impact of Research on Software Technology. Edited by P. Wegner. Cambridge, Ma.: M.I.T. Press, June 1979, 276-301.

14. Principato, Robert N., Jr. A Formalization of the State Machine Specification Technique. M.I.T., Laboratory for Computer Science, LCS/TR-202. Cambridge, Ma., July 1979.

15. Snyder, Alan. A Machine Architecture to Support an Object-Oriented Language. M.I.T., Laboratory for Computer Science, LCS/TR-209. Cambridge, Ma., March 1979.

16. Svobodova, Liba; Liskov, Barbara; and Clark, David. Distributed Computer Systems: Structure and Semantics. M.I.T., Laboratory for Computer Science, LCS/TR-215. Cambridge, Ma., April 1979.

Accepted for Publication

1. Liskov, Barbara H., and Snyder, Alan. "Exception Handling in CLU." To be published in the Proceedings of the IEEE Transactions on Software Engineering.

2. Peterson, James L. "Notes on a Workshop on Distributed Computing." To be published in Operating Systems Review.

Theses in Progress

1. Allen, Matthew D. "A Comparative Analysis of Programmming Languages." S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1979.

2. Atkinson, Russell R. "Verification of Serializers." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1980.

3. Berzins, Valdis. "Abstract Model Specification for Data Abstractions." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, July 1979.

4. Herlihy, Maurice. "Communicating Abstract Values in Messages." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1980.

5. Kapur, Deepak. "Towards a Theory of Data Abstractions." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1979.

6. Knopf, Ralph. "A Formatter for CLU." S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1979.

7. Leach, Paul. "Objects and Information Containers in a CLU Garbage Collector." S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1980.

8. Moss, Eliot. "Distributed Programming Environment." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1980.

9. Mullendor. Chris. "Performance Analysis of the CLU Implementation." S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1980.

10. Schaffert, J. Craig. "Specifications and Proofs in Object Oriented Languages." Ph.D Thesis, M.I.T.; Department of Electrical Engineering and Computer Science, expected date of completion, June 1980.

11. Srivas, M. K. "Automatic Generation of Implementations of Data Abstractions." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, June 1980.

Theses Completed

1. Bloom, Toby. Synchronization Mechanisms for Modular Programming Languages. S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, April 1979.

2. Snyder, Alan. A Machine Architecture to Support an Object-Oriented Language. Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, March 1979.

3. Zachary, Joseph. "A CLU Machine Design Evaluation." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1979.

Talks

1. Greif, Irene G. "Specifying the Semantics of While-Programs." University of Washington, Computer Science Department, Seattle, Wa., January 1979; IBM, Yorktown Heights, N. Y., March 1979.

2. Liskov, Barbara H. "Issues in Distributed Computing." Quality Software Workshop, Salt Lake City, Utah, October 1978.

3. Liskov, Barbara H. "Implementation Aspects of CLU." ACM National Conference, Washington, D.C., December 1978.

4. Liskov, Barbara H. "Use of Data Abstractions in Data Bases." ACM National Conference, Washington, D. C., December 1978.

5. Liskov, Barbara H. "Linguistic Support for Distributed Computing." Eidgenossische Technische Hochschule, Zurich, Switzerland, January 1979.

6. Liskov, Barbara H. "Introduction to CLU"; "An Example of Modular Program Development"; "Embedding Data Abstraction in Programming Languages." Copenhagen Winter School on Abstract Software Specifications, Copenhagen, Denmark, January 1979.

7. Liskov, Barbara H. "Introduction to CLU"; "An Example of Modular Program Development." Bell Laboratories, Piscataway, N. J., March 1979.

8. Liskov, Barbara H. "Linguistic Support for Distributed Programs." University of Rochester, Rochester, N.Y., April 1979.

9. Liskov, Barbara H. "Message Passing Primitives." Quality Software Workshop, Amherst, Ma., April 1979.

10. Liskov, Barbara H. "Communicating Abstract Values." National Computer Conference, New York, N. Y., June 1979.