A design methodology for reliable software systems*

by B. H. LISKOV**

The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

Any user of a computer system is aware that current systems are unreliable because of errors in their software components. While system designers and implementers recognize the need for reliable software, they have been unable to produce it. For example, operating systems such as OS/360 are released to the public with hundreds of errors still in them.[1]

A project is underway at the MITRE Corporation which is concerned with learning how to build reliable software systems. Because systems of any size can always be expected to be subject to changes in requirements, the project goal is to produce not only reliable software, but readable software which is relatively easy to modify and maintain. This paper describes a design methodology developed as part of that project.

Rationale

Before going on to describe the methodology, a few words are in order about why a design methodology approach to software reliability has been selected.† The unfortunate fact is that the standard approach to building systems, involving extensive debugging, has not proved successful in producing reliable software, and there is no reason to suppose it ever will. Although improvements in debugging techniques may lead to the detection of more errors, this does not imply that all errors will be found. There certainly is no guarantee of this implicit in debugging: as Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence." [3]

In order for testing to guarantee reliability, it is necessary to insure that all relevant test cases have been checked. This requires solving two problems:

(1) A complete (but minimal) set of relevant test cases must be identified.
(2) It must be possible to test all relevant test cases; this implies that the set of relevant test cases is small and that it is possible to generate every case.

The solutions to these problems do not lie in the domain of debugging, which has no control over the sources of the problems. Instead, since it is the system design which determines how many test cases there are and how easily they can be identified, the problems can be solved most effectively during the design process: The need for exhaustive testing must influence the design.

We believe that such a design methodology can be developed by borrowing from the work being done on proof of correctness of programs. While it is too difficult at present to give formal proofs of the correctness of large programs, it is possible to structure programs so that they are more amenable to proof techniques. The objective of the methodology presented in this paper is to produce such a program structure, which will lend itself to *informal* proofs of correctness. The proofs, in addition to building confidence in the correctness of the program, will help to identify the relevant test cases, which can then be exhaustively tested. When exhaustive testing is combined with informal proofs, it is reasonable to expect reliable software after testing is complete. This expectation is borne out by at least one experiment performed in the past.[4]

The scope of the paper

A key word in the discussion of software reliability is "complex"; it is only when dealing with complex sys-

* This work was supported by Air Force Contract No. F19(628)-71-C-0002.
** Present Address—Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.
† The material in this section is covered in much greater detail in Liskov and Towster.[2]

tems that reliability becomes an acute problem. A two-fold definition is offered for "complex." First, there are many system states in such a system, and it is difficult to organize the program logic to handle all states correctly. Second, the efforts of many individuals must be coordinated in order to build the system. A design methodology is concerned with providing techniques which enable designers to cope with the inherent logical complexity effectively. Coordination of the efforts of individuals is accomplished through management techniques.

The fact that this paper only discusses a design methodology should not be interpreted to imply that management techniques are unimportant. Both design methodology and management techniques are essential to the successful construction of reliable systems. It is customary to divide the construction of a software system into three stages: design, implementation, and testing. Design involves both making decisions about what precisely a system will do and then planning an overall structure for the software which enables it to perform its tasks. A "good" design is an essential first step toward a reliable system, but there is still a long way to go before the system actually exists. Only management techniques can insure that the system implementation fits into the structure established by the design and that exhaustive testing is carried out. The management techniques should not only have the form of requirements placed on personnel; the organization of personnel is also important. It is generally accepted that the organizational structure imposes a structure on the system being built.[5] Since we wish to have a system structure based on the design methodology, the organizational structure must be set up accordingly.*

CRITERIA FOR A GOOD DESIGN

The design methodology is presented in two parts. This section defines the criteria which a system design should satisfy. The next section presents guidelines intended to help a designer develop a design satisfying the criteria.

To reiterate, a complex system is one in which there are so many system states that it is difficult to understand how to organize the program logic so that all states will be handled correctly. The obvious technique to apply when confronting this type of situation is "divide and rule." This is an old idea in programming and is known as modularization. Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas:[7] "The connections between modules are the assumptions which the modules make about each other." Modules have connections in control via their entry and exit points; connections in data, explicitly via their arguments and values, and implicitly through data referenced by more than one module; and connections in the services which the modules provide for one another.

Traditionally, modularity was chosen as a technique for system production because it makes a large system more manageable. It permits efficient use of personnel, since programmers can implement and test different modules in parallel. Also, it permits a single function to be performed by a single module and implemented and tested just once, thus eliminating some duplication of effort and also standardizing the way such functions are performed.

The basic idea of modularity seems very good, but unfortunately it does not always work well in practice. The trouble is that the division of a system into modules may introduce additional complexity. The complexity comes from two sources: functional complexity and complexity in the connections between the modules. Examples of such complexity are:

(1) A module is made to do too many (related but different) functions, until its logic is completely obscured by the tests to distinguish among the different functions (functional complexity).
(2) A common function is not identified early enough, with the result that it is distributed among many different modules, thus obscuring the logic of each affected module (functional complexity).
(3) Modules interact on common data in unexpected ways (complexity in connections).

The point is that if modularity is viewed only as an aid to management, then any ad hoc modularization of the system is acceptable. However, the success of modularity depends directly on how well modules are chosen. We will accept modularization as the way of organizing the programming of complex software systems. A major part of this paper will be concerned with the question of how good modularity can be achieved, that is, how modules can be chosen so as to minimize the connections between them. First, however, it is necessary to give a definition of "good" modularity. To emphasize the requirement that modules be as disjoint as possible, and because the term "module" has been used so often and so diversely, we will discard it and define modularity as the division of the system into

* Management techniques intended to support the design methodology proposed in this paper are described by Liskov.[6]

"partitions." The definition of good modularity will be based on a synthesis of two techniques, each of which addresses a different aspect of the problem of constructing reliable software. The first, levels of abstraction, permits the development of a system design which copes with the inherent complexity of the system effectively. The second, structured programming, insures a clear and understandable representation of the design in the system software.

Levels of abstraction

Levels of abstraction were first defined by Dijkstra.[8] They provide a conceptual framework for achieving a clear and logical design for a system. The entire system is conceived as a hierarchy of levels, the lowest levels being those closest to the machine. Each level supports an important abstraction; for example, one level might support segments (named virtual memories), while another (higher) level could support files which consist of several segments connected together. An example of a file system design based entirely on a hierarchy of levels can be found in Madnick and Alsop.[9]

Each level of abstraction is composed of a group of related functions. One or more of these functions may be referenced (called) by functions belonging to other levels; these are the external functions. There may also be internal functions which are used only within the level to perform certain tasks common to all work being performed by the level and which cannot be referenced from other levels of abstraction.

Levels of abstraction, which will constitute the partitions of the system, are accompanied by rules governing some of the connections between them. There are two important rules governing levels of abstraction. The first concerns resources (I/O devices, data): each level has resources which it owns exclusively and which other levels are not permitted to access. The second involves the hierarchy: lower levels are not aware of the existence of higher levels and therefore may not refer to them in any way. Higher levels may appeal to the (external) functions of lower levels to perform tasks; they may also appeal to them to obtain information contained in the resources of the lower levels.*

* In the Madnick and Alsop paper referenced earlier, the hierarchy of levels is strictly enforced in the sense that if the third level wishes to make use of the services of the first level, it must do so through the second level. This paper does not impose such a strict requirement; a high level may make use of a level several steps below it in the hierarchy without necessarily requiring the assistance of intermediate levels. The 'THE' system[8] and the Venus system[10] contain examples of levels used in this way.

Structured programming

Structured programming is a programming discipline which was introduced with reliability in mind.[11,12] Although of fairly recent origin, the term "structured programming" does not have a standard definition. We will use the following definition in this paper.

Structured programming is defined by two rules. The first rule states that structured programs are developed from the top down, in levels.* The highest level describes the flow of control among major functional *components* (major subsystems) of the system; *component names* are introduced to represent the components. The names are subsequently associated with code which describes the flow of control among still lower-level components, which are again represented by their component names. The process stops when no undefined names remain.

The second rule defines which control structures may be used in structured programs. Only the following control structures are permitted: concatenation, selection of the next statement based on the testing of a condition, and iteration. Connection of two statements by a *goto* is not permitted. The statements themselves may make use of the component names of lower-level components.

Structured programming and proofs of correctness

The goal of structured programming is to produce program structures which are amenable to proofs of correctness. The proof of a structured program is broken down into proofs of the correctness of each of the components. Before a component is coded, a specification exists explaining its input and output and the function which it is supposed to perform. (The specification is defined at the time the component name is introduced; it may even be part of the name.) When the component is coded, it is expressed in terms of specifications of lower level components. The theorem to be proved is that the code of the component matches its specifications; this proof will be given based on axioms stating that lower level components match their specifications.

The proof depends on the rule about control structures in two important ways. First, limiting a component to combinations of the three permissible control structures insures that control always returns from a component to the statement following the use of the

* The levels in a structured program are not (usually) levels of abstraction, because they do not obey the rule about ownership of resources.

component name (this would not be true if *goto* statements were permitted). This means that reasoning about the flow of control in the system may be limited to the flow of control as defined locally in the component being proved. Second, each permissible control structure is associated with a well-known rule of inference: concatenation with linear reasoning, iteration with induction, and conditional selection with case analysis. These rules of inference are the tools used to perform the proof (or understand the component).

Structured programming and system design

Structured programming is obviously applicable to system implementation. We do not believe that by itself it constitutes a sufficient basis for system design; rather we believe that system design should be based on identification of levels of abstraction.* Levels of abstraction provide the framework around which and within which structured programming can take place. Structured programming is compatible with levels of abstraction because it provides a comfortable environment in which to deal with abstractions. Each structured program component is written in terms of the names of lower-level components; these names, in effect, constitute a vocabulary of abstractions.

In addition, structured programs can replace flowcharts as a way of specifying what a program is supposed to do. Figure 1 shows a structured program for the top level of the parser in a bottom-up compiler for an operator precedence grammar, and Figure 2 is a flowchart containing approximately the same amount of detail. While it is slightly more difficult to write the structured program, there are compensating advantages. The structured program is part of the final program; no translation is necessary (with the attendant possibility of introduction of errors). In addition, a structured program is more rigorous than a flowchart. For one thing, it is written in a programming language and therefore the semantics are well defined. For another, a flowchart only describes the *flow of control* among parts of a system, but a structured program at a minimum must also define the data controlling its flow,

```
begin
integer relation;
boolean must_scan;
string symbol;
stack parse_stack;
must_scan := true;
push(parse_stack, eof_entry);
while not finished(parse_stack) do
  begin
  if must_scan then symbol := scan_next_symbol;
  relation := precedence_relation(top(parse_stack), symbol);
  perform_operation_based_on_relation(relation, parse_stack,
                                       symbol, must_scan)
  end
end
```

Figure 1—A structured program for an operator precedence parser

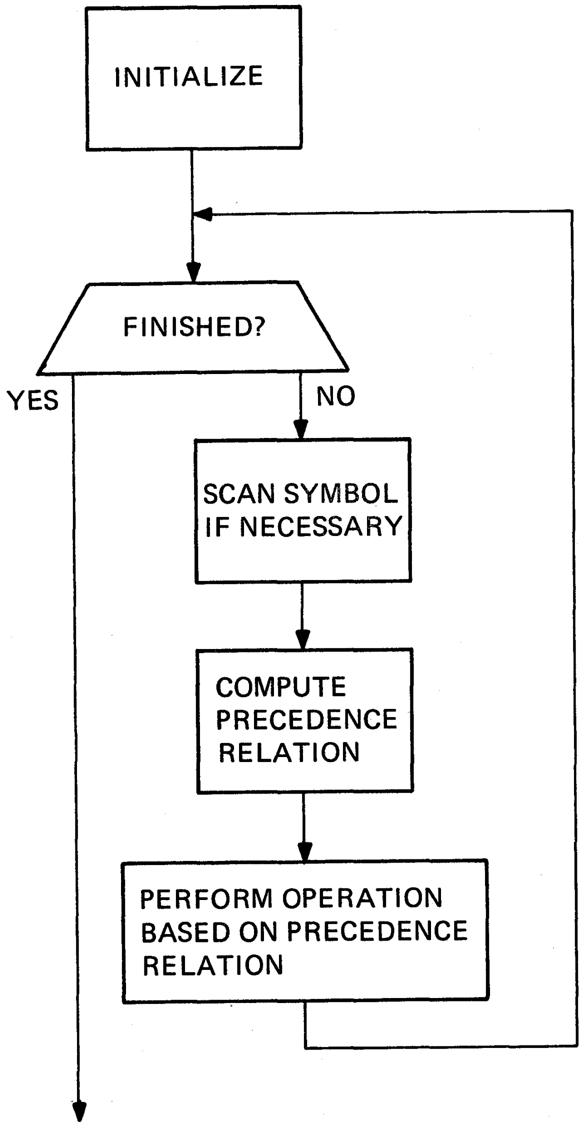
INITIALIZE
FINISHED?
YES
NO
SCAN SYMBOL IF NECESSARY
COMPUTE PRECEDENCE RELATION
PERFORM OPERATION BASED ON PRECEDENCE RELATION

Figure 2—Flowchart of an operator precedence parser

* A recent paper by Henderson and Snowden[13] describes an experiment in which structured programming was the only technique used to build a program. The program had an error in it which was the direct result of not identifying a level of abstraction.

so the description it provides is more concrete. In addition, it defines the arguments and values of a referenced component, and if a change in level of abstraction occurs at that point, then the data connection between the two components is completely defined by the structured program. This should help to avoid interface errors usually uncovered during system integration.

Basic definition

We now present a definition of good modularity supporting the goal of software reliability. The system is divided into a hierarchy of partitions, where each partition represents one level of abstraction, and consists of one or more functions which share common resources. At the same time, the entire system is expressed by a structured program which defines the way control passes among the partitions. The connections between the partitions are limited as follows:

(1) The connections in control are limited by the rules about the hierarchy of levels of abstraction and also follow the rules for structured programs.
(2) The connections in data between partitions are limited to the explicit arguments passed from the functions of one partition to the (external) functions of another partition. Implicit interaction on common data may only occur among functions within a partition.
(3) The combined activity of the functions in a partition support its abstraction and nothing more. This makes the partitions logically independent of one another. For example, a partition supporting the abstraction of files composed of many virtual memories should not contain any code supporting the existence of virtual memories.

A system design satisfying the above requirements is compatible with the goal of software reliability. Since the system structure is expressed as a structured program, it should be possible to prove that it satisfies the system specifications, assuming that the structured programs which will eventually support the functions of the levels of abstraction satisfy their specifications. In addition, it is reasonable to expect that exhaustive testing of all relevant test cases will be possible. Exhaustive testing of the whole system means that each partition must be exhaustively tested, and all combinations of partitions must be exhaustively tested. Exhaustive testing of a single partition involves both testing based on input parameters to the functions in the partition and testing based on intermediate values of state variables of the partition. When this testing is complete, it is no longer necessary to worry about the state variables because of requirement 2. Thus, the testing of combinations of partitions is limited to testing the input and output parameters of the external functions in the partitions. In addition, requirement 3 says that partitions are logically independent of one another; this means that it is not necessary when combining partitions to test *combinations* of the relevant test cases for each partition. Thus, the number of relevant test cases for two partitions equals the sum of the relevant test cases for each partition, not the product.

GUIDELINES FOR SYSTEM DESIGN

Now that we have a definition of good modularization, the next question is how a system modularization satisfying this definition can be achieved. The traditional technique for modularization is to analyze the execution-time flow of the system and organize the system structure around each major sequential task. This technique leads to a structure which has very simple connections in control, but the connections in data tend to be complex (for examples see Parnas[14] and Cohen[15]). The structure therefore violates requirement 2; it is likely to violate requirement 3 also since there is no reason (in general) to assume any correspondence between the sequential ordering of events and the independence of the events.

If the execution flow technique is discarded, however, we are left with almost nothing concrete to help us make decisions about how to organize the system structure. The guidelines presented here are intended to help rectify this situation. First are some guidelines about how to select abstractions; these guidelines tend to overlap, and when designing a system, the choice of a particular abstraction will probably be based on several of the guidelines. Next the question of how to proceed with the design is addressed. Finally, an example of the selection of a particular abstraction within the Venus system[10] is presented to illustrate the application of several of the principles; an understanding of Venus is not necessary for understanding the example.

Guidelines for selecting abstractions

Partitions are always introduced to support an abstraction or concept which the designer finds helpful in thinking about the system. Abstraction is a very valuable aid to ordering complexity. Abstractions are introduced in order to make what the system is doing clearer and more understandable; an abstraction is a conceptual simplification because it expresses what is being done without specifying how it is done. The purpose of this section is to discuss the types of abstractions which may be expected to be useful in designing a system.

Abstractions of resources

Every hardware resource available on the system will be represented by an abstraction having useful characteristics for the user or the system itself. The abstraction will be supported by a partition whose functions map the characteristics of the abstract resource into the characteristics of the real underlying resource or resources. This mapping may itself make use of several lower partitions, each supporting an abstraction useful in defining the functions of the original partition. It is likely that a strict hierarchy will be imposed on the group of partitions; that is, other parts of the system may only reference the functions in the original partition. In this case, we will refer to the lower partitions as "sub-partitions."

Two examples of abstract resources are given. In an interactive system, "abstract teletypes" with end-of-message and erasing conventions are to be expected. In a multiprogramming system, the abstraction of processes frees the rest of the system from concern about the true number of processors.

Abstract characteristics of data

In most systems the users are interested in the structure of data rather than (or in addition to) storage of data. The system can satisfy this interest by the inclusion of an abstraction supporting the chosen data structure; functions of the partition for that abstraction will map the structure into the way data is actually represented by the machine (again this may be accomplished by several sub-partitions). For example, in a file management system such an abstraction might be an indexed sequential access method. The system itself also benefits from abstract representation of data; for example, the scanner in a compiler permits the rest of the compiler to deal with symbols rather than with characters.

Simplification via limiting information

According to the third requirement for good modularization, the functions comprising a partition support only one abstraction and nothing more. Sometimes it is difficult to see that this restriction is being violated, or to recognize that the possibility for identification of another abstraction exists.

One technique for simplification is to limit the amount of information which the functions in the partition need to know (or even have access to). An example of such information is the complicated format in which data is stored for use by the functions in the partition (the data would be a resource of the partition). The functions require the information embedded in the data but need not know how it is derived from the data. This knowledge can be successfully hidden within a lower partition (possibly a sub-partition) whose functions will provide requested information when called; note that the data in question become a resource of the lower partition.

Simplification via generalization

Another technique for simplification is to recognize that a slight generalization of a function (or group of functions) will cause the functions to become generally useful. Then a separate partition can be created to contain the generalized function or functions. Separating such groups is a common technique in system implementation and is also useful for error avoidance, minimization of work, and standardization. The existence of such a group simplifies other partitions, which need only appeal to the functions of the lower partition rather than perform the tasks themselves. An example of a generalization is a function which will move a specified number of characters from one location to another, where both locations are also specified; this function is a generalization of a function in which one or more of the input parameters is assumed.

Sometimes an already existing partition contains functions supporting tasks very similar to some work which must be performed. When this is true, a new partition containing new versions of those functions may be created, provided that the new functions are not much more complex than the old ones.

System maintenance and modification

Producing a system which is easily modified and maintained is one of our primary goals. This goal can be aided by separating into independent partitions functions which are performing a task whose definition is likely to change in the future. For example, if a partition supports paging of data between core and some backup storage, it may be wise to isolate as an independent partition those functions which actually know what the backup storage device is (and the device becomes a resource of the new partition). Then if a new device is added to the system (or a current device is removed), only the functions in the lower partition will be affected; the higher partition will have been isolated from such changes by the requirement about data connections between partitions.

How to proceed with the design

Two phases of design are distinguished. The very first phase of the design (phase 1) will be concerned with defining precise system specifications and analyzing them with respect to the environment (hardware or software) in which the system will eventually exist. The result of this phase will be a number of abstractions which represent the eventual system behavior in a very general way. These abstractions imply the existence of partitions, but very little is known about the connections between the partitions, the flow of control among the partitions (although a general idea of the hierarchy of partitions will exist), or how the functions of the partitions will be coded. Every important external characteristic of the system should be present as an abstraction at this stage. Many of the abstractions have to do with the management of system resources; others have to do with services provided to the user.

The second phase of system design (phase 2) investigates the practicality of the abstractions proposed by phase 1 and establishes the data connections between the partitions and the flow of control among the partitions. This latter exercise establishes the placement of the various partitions in the hierarchy. The second phase occurs concurrently with the first; as abstractions are proposed, their utility and practicality are immediately investigated. For example, in an information retrieval system the question of whether a given search technique is efficient enough to satisfy system constraints must be investigated.

A partition has been adequately investigated when its connections with the rest of the system are known and when the designers are confident that they understand exactly what its effect on the system will be. Varying depths of analysis will be necessary to achieve this confidence. It may be necessary to analyze how the functions of the partition could be implemented, involving phase 1 analysis as new abstractions are postulated requiring lower partitions or sub-partitions. Possible results of a phase 2 investigation are that an abstraction may be accepted with or without changes, or it may be rejected. If an abstraction is rejected, then another abstraction must be proposed (phase 1) and investigated (phase 2). The iteration between phase 1 and phase 2 continues until the design is complete.

Structured programming

It is not clear exactly how early structured programming of the system should begin. Obviously, whenever the urge is felt to draw a flowchart, a structured program should be written instead. Structured programs connecting all the partitions together will be expected by the end of the design phase. The best rule is probably to keep trying to write structured programs; failure will indicate that system abstractions are not yet sufficiently understood and perhaps this exercise will shed some light on where more effort is needed or where other abstractions are required.

When is the design finished?

The design will be considered finished when the following criteria are satisfied:

(1) All major abstractions have been identified and partitions defined for them; the system resources have been distributed among the partitions and their positions in the hierarchy established.
(2) The system exists as a structured program, showing how the flow of control passes among the partitions. The structured program consists of several components, but no component is likely to be completely defined; rather each component is likely to use the names of lower-level components which are not yet defined. The interfaces between the partitions have been defined, and the relevant test cases for each partition have been identified.
(3) Sufficient information is available so that a skeleton of a user's guide to the system could be written. Many details of the guide would be filled in later, but new sections should not be needed.*

An example from Venus

The following example from the Venus system[10] is presented because it illustrates many of the points made about selection, implementation, and use of abstractions and partitions. The concept to be discussed is that of external segment name, referred to as ESN from now on.

The concept of ESN was introduced as an abstraction primarily for the benefit of users of the system. The important point is that a segment (named virtual memory) exists both conceptually (as a place where a

* This requirement helps to insure that the design fulfills the system specifications. In fact, if there is a customer for whom the system is being developed, a preliminary user's guide derived from the system design could be a means for reviewing and accepting the design.

programmer thinks of information as being stored) and in reality (the encoding of that information in the computer). The reality of a segment is supported by an internal segment name (ISN) which is not very convenient for a programmer to use or remember. Therefore, the symbolic ESN was introduced.

As soon as the concept of ESN was imagined, the existence of a partition supporting this concept was implied. This partition owned a nebulous data resource, a dictionary, which contained information about the mappings between ESNs and ISNs. The formatting of this data was hidden information as far as the rest of the system was concerned. In fact, decisions about the dictionary format and about the algorithms used to search a dictionary could safely be delayed until much later in the design process. A collective name, the dictionary functions, was given to the functions in this partition.

Now phase 2 analysis commenced. It was necessary to define the interface presented by the partition to the rest of the system. Obvious items of interest are ESNs and ISNs; the format of ISNs was already determined by the computer architecture, but it was necessary to decide about the format of ESNs. The most general format would be a count of the number of characters in the ESN followed by the ESN itself; for efficiency, however, a fixed format of six characters was selected.

At this point a generalization of the concept of ESN occurred, because it was recognized that a two-part ESN would be more useful than a single symbolic ESN. The first part of the ESN is the symbolic name of the dictionary which should be used to make the mapping; the second part is the symbolic name to be looked up in the dictionary. This concept was supported by the existence of a dictionary containing the names of all dictionaries. A format had to be chosen for telling dictionary functions which dictionary to use; for reasons of efficiency, the ISN of the dictionary was chosen (thus avoiding repeated conversions of dictionary ESN into dictionary ISN).

When phase 2 analysis was over, we had the identification of a partition; we knew what type of function belonged in this partition, what sort of interface it presented to the rest of the system, and what information was kept in dictionaries. As the system design proceeded, new dictionary functions were specified as needed. Two generalizations were realized later. The first was to add extra information to the dictionary; this was information which the system wanted on a segment basis, and the dictionaries were a handy place to store it. The second was to make use of dictionary functions as a general mapping device; for example, dictionaries are used to hold information about the mapping of record names into tape locations, permitting simplification of a higher partition.

In reality, as soon as dictionaries and dictionary functions were conceived, a core of dictionary functions was implemented and tested. This is a common situation in building systems and did not cause any difficulty in this case. For one thing, extra space was purposely left in dictionary entries because we suspected we might want extra information there later although we did not then know what it was. The search algorithm selected was straight serial search; the search was embedded in two internal dictionary functions (a sub-partition) so that the format of the dictionaries might be changed and the search algorithm redefined with very little effect on the system or most of the dictionary functions. This follows the guideline of modifiability.

CONCLUSIONS

This paper has described a design methodology for the development of reliable software systems. The first part of the methodology is a definition of a "good" system modularization, in which the system is organized into a hierarchy of "partitions", each supporting an "abstraction" and having minimal connections with one another. The total system design, showing how control flows among the partitions, is expressed as a structured program, and thus the system structure is amenable to proof techniques.

The second part of the methodology addresses the question of how to achieve a system design having good modularity. The key to design is seen as the identification of "useful" abstractions which are introduced to help a designer think about the system; some methods of finding abstractions are suggested. Also included is a definition of the "end of design", at which time, in addition to having a system design with the desired structure, a preliminary user's guide to the system could be written as a way of checking that the system meets its specifications.

Although the methodology proposed in this paper is based on techniques which have contributed to the production of reliable software in the past, it is nevertheless largely intuitive, and may prove difficult to apply to real system design. The next step to be undertaken at MITRE is to test the methodology by conscientiously applying it, in conjunction with certain management techniques,[6] to the construction of a small, but complex, multi-user file management system. We hope that this exercise will lead to the refinement, extension and clarification of the methodology.

ACKNOWLEDGMENTS

The author wishes to thank J. A. Clapp and D. L. Parnas for many helpful criticisms.

REFERENCES

1 J N BUXTON B RANDELL (eds)
Software engineering techniques
Report on a Conference Sponsored by the NATO Science Committee Rome Italy p 20 1969

2 B H LISKOV E TOWSTER
The proof of correctness approach to reliable systems
The MITRE Corporation MTR 2073 Bedford Massachusetts 1971

3 E W DIJKSTRA
Structured programming
Software Engineering Techniques
Report on a Conference sponsored by the NATO Science Committee Rome Italy J N Buxton and B Randell (eds) pp 84-88 1969

4 F T BAKER
Chief programmer team management of production programming
IBM Syst J 11 1 pp 56-73 1972

5 M CONWAY
How do committees invent?
Datamation 14 4 pp 28-31 1968

6 B H LISKOV
Guidelines for the design and implementation of reliable software systems
The MITRE Corporation MTR 2345 Bedford Massachusetts 1972

7 D L PARNAS
Information distribution aspects of design methodology
Technical Report Department of Computer Science Carnegie-Mellon University 1971

8 E W DIJKSTRA
The structure of the "THE"—multiprogramming system
Comm ACM 11 5 pp 341-346 1968

9 S MADNICK J W ALSOP II
A modular approach to file system design
AFIPS Conference Proceedings 34 AFIPS Press Montvale New Jersey pp 1-13 1969

10 B H LISKOV
The design of the Venus operating system
Comm ACM 15 3 pp 144-149 1972

11 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven The Netherlands 1969

12 H D MILLS
Structured programming in large systems
Debugging Techniques in Large Systems R Rustin (ed) Prentice Hall Inc Englewood Cliffs New Jersey pp 41-55

13 P HENDERSON R SNOWDEN
An experiment in structured programming
BIT 12 pp 38-53 1972

14 D L PARNAS
On the criteria to be used in decomposing systems into modules
Technical Report CMU-CS-71-101 Carnegie-Mellon University 1971

15 A COHEN
Modular programs: Defining the module
Datamation 18 1 pp 34-37 1972