Specification Techniques for Data Abstractions

BARBARA H. LISKOV AND STEPHEN N. ZILLES, MEMBER, IEEE

Abstract—**The main purposes in writing this paper are to discuss the importance of formal specifications and to survey a number of promising specification techniques. The role of formal specifications both in proofs of program correctness, and in programming methodologies leading to programs which are correct by construction, is explained. Some criteria are established for evaluating the practical potential of specification techniques. The importance of providing specifications at the right level of abstraction is discussed, and a particularly interesting class of specification techniques, those used to construct specifications of data abstractions, is identified. A number of specification techniques for describing data abstractions are surveyed and evaluated with respect to the criteria. Finally, directions for future research are indicated.**

Index Terms—**Data abstractions, programming methodology, proofs of correctness, specifications, specification techniques.**

I. INTRODUCTION

IN THE past, the advantages of formal specifications have been outweighed by the difficulty of constructing them for practical programs. However, recent work in programming methodology has identified a program unit, supporting a data abstraction, which is both widely useful, and for which it is practical to write formal specifications. Some formal specification techniques have already been developed for describing data abstractions. It is the promise of these techniques, some of which are described later in this paper, which leads us to believe that formal specifications can soon become an intrinsic feature of the program construction process. By writing this paper, we hope to encourage research in the development of formal specification techniques, and their application to practical program construction.

In the remainder of the introduction we discuss what is meant by formal specifications, and then explain some advantages arising from their use. In Section II a number of criteria are presented which will permit us to judge techniques for constructing formal specifications. Section III identifies the kind of program unit, supporting a data abstraction, to which the specification techniques described later in this paper apply. Section IV discusses properties of specification techniques for data abstractions and in Section V some existing techniques for providing specifications for data abstractions are surveyed and compared. Finally, we conclude by pointing out areas for future research.

Manuscript received December 4, 1974; revised February 5, 1975. This work was supported in part by the IBM funds for research in computer science.

B. H. Liskov is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

S. N. Zilles is with IBM Corporation, San Jose, Calif.

Proofs of Correctness

Of serious concern in software construction are techniques which permit us to recognize whether a given program is correct, i.e., does what it is supposed to do. Although we are coming to realize that correctness is not the only desirable property of reliable software, surely it is the most fundamental. If a program is not correct, then its other properties (e.g., efficiency, fault tolerance) have no meaning since we cannot depend on them.

Techniques for establishing the correctness of programs may be classified as to whether they are formal or informal. All techniques in common use today (debugging, testing, program reading) are informal techniques; either the investigation of the properties of the program is incomplete, or the steps in the reasoning place too much dependence on human ingenuity and intuition. The continued existence of errors in software to which such techniques have been applied attests to their inadequacy. Formal techniques, such as the verification condition [1], [2] and fixed-point [3] methods, attempt to establish properties of a program with respect to all legitimate inputs by means of a process of reasoning in which each step is formally justified by appeal to rules of inference, axioms and theorems. Unfortunately, these techniques have been very difficult to apply, and have therefore not yet been of much practical interest. However interest in formal techniques can be expected to increase in the future; economic pressure for reliable software is growing [4] and the domain of applicability of formal techniques is also growing because of the development of programming methodologies leading to programs to which formal techniques are more readily applied. Indeed, application of proof techniques to practical programs is being attempted in the area of operating system security [5]–[7], where the need for absolute certainty about the correct functioning of software is very great.

To study techniques which establish program correctness, it is interesting to examine a model of what the correctness of a program means. What we are looking for is a process which establishes that a program correctly implements a *concept* which exists in someone's mind. The concept can usually be implemented by many programs—an infinite number, in general—but of these only a small finite number are of practical interest. This situation is shown in Fig. 1. In current practice, the concept is stated informally and, regardless of the technique used to demonstrate the correctness of a program (usually testing), the result of applying the technique can be stated only in informal terms.

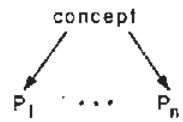

Fig. 1. Concept and all the programs which implement the concept correctly.

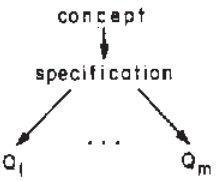

Fig. 2. Concept, its formal specification, and all programs which can be proved equivalent to the specification.

With formal techniques, a *specification* is interposed between the concept and the programs. Its purpose is to provide a mathematical description of the concept, and the correctness of a program is established by proving that it is equivalent to the specification. The specification will be provably satisfied by a class of programs (again, often an infinite number of which only a small finite number are of interest). This situation is shown in Fig. 2.

Proofs of large programs do not consist of a single monolithic proof with no interior structure. Instead, the overall proof is divided into a hierarchy of many smaller proofs which establish the correctness of separate program units. For each program unit, a proof is given that it satisfies its specification; this proof makes use of the specifications of other program units, and rests on the assumption that those program units will be proved consistent with their specifications.[1] Thus a specification is used in two ways: as a description against which a program is proved correct, and as a set of axioms in the proof of other programs. At the top of the proof hierarchy is a program unit which corresponds to the entire program. At the bottom is the programming language, and the hierarchy is based on the axioms for the programming language and its primitives.

The proof methodology can fail in two ways. First, a proof may incorrectly establish some program (or program unit) P as equivalent to the specification when, in fact, it is not. This is a problem which can be eliminated by using a computer as, at least, a proof checker. (Observe that one advantage of using formal specifications is that such specifications can be processed by a computer.)

The second way the methodology can fail is if the specification does not correctly capture the meaning of a concept. We will say a specification *captures* a concept if every Q_i in Fig. 2 is some P_j in Fig. 1. There is no formal way of establishing that a specification captures a concept, but we expect to have gained from using the proof methodology because (hopefully) a specification is easier to understand than a program, so that "convincing oneself" that a specification captures a concept is less error prone than a similar process to a program. Furthermore, any distinction between concept and specification may be irrelevant because of the hierarchical nature of the proof process. If a program P is proven equivalent to its specification, and every program using P is proven correct using that specification, then the concept which P was intended to implement can safely be ignored.

[1] Special techniques [3] must be used if the program units are mutually recursive.

Advantages of Formal Specifications

Proving the correctness of programs is described above as a two step process: first, a formal specification is provided to describe the concept, and second, the program is proven equivalent to the specification by formal, analytic means. Formal techniques are not necessarily limited to axiomatic methods. For example, it may also be possible to develop testing methodologies that are based on a comparison of the formal specification and the implementation. The output of a methodology would be a set of critical test cases which, if successfully executed, establish that the program correctly implements the specification. The formality of the specification means that the computer can aid in the proof process, for example, by checking the steps of a program proof, or by automatically generating test cases.

Clearly, the specification must be present before a proof can be given. However, formal specifications are of interest even if not followed by a formal proof. Formal specifications are very valuable in conjunction with the idea of making code "public" [8] in order to encourage programmers to read one another's code. In the absence of a formal specification, a programmer can only compare a program he is reading with his intuitive understanding of what the program is supposed to do. A formal specification would be better, since intuition is often unreliable. With the addition of formal specifications, code reading becomes an informal proof technique; each step in the proof process now rests on understanding a formal description rather than manipulating the description in a formal way.[2] As such, it can be a powerful aid in establishing program correctness.

Formal specifications can also play a major role while a program is being constructed. It is widely recognized that a specification of what a program is intended to do should be given before the program is actually coded, both to aid understanding of the concept involved, and to increase the likelihood that the program, when implemented, will perform the intended function. However, because it is difficult to construct specifications using informal techniques, such as English, specifications are often omitted, or are

[2] The relationship between proofs and understanding is a major motivating factor in structured programming. For example, the "go to" statement is eliminated because the remaining control structures are each associated with a well-known proof technique, and therefore the programs are intellectually manageable [9].

given in a sketchy and incomplete manner. Formal specification techniques, like the ones to be described later in this paper, provide a concise and well-understood specification or design language, which should reduce the difficulty of constructing specifications.

Formal specifications are superior to informal ones as a communication medium. The specifications developed during the design process serve to communicate the intentions of the designer of a program to its implementors, or to communicate between two programmers: the programmer implementing the program being specified, and the programmer who wishes to use that program. Problems arise if the specification is ambiguous: that is, fails for some reason to capture the concept so that two programs with different conceptual properties both satisfy the specification. Ambiguities can be resolved by mutual agreement, provided those using the specification realize that an ambiguity exists. Often this is not realized, and instead the ambiguity is resolved in different ways by different people. Formal specifications are less likely to be ambiguous than informal ones because they are written in an unambiguous language. Also, the meaning of a formal specification is understood in a formal way, and therefore ambiguities are more likely to be recognized.

The above paragraphs have sketched a program construction methodology that could lead to programs which are correct by construction. Formal specifications play a major role in this methodology, which differs from standard descriptions of structured programming [9] primarily in the emphasis it places on specifications.[3] Specifications are first introduced by the designer to describe the concepts he develops in a precise and unambiguous way. Each concept will be supported by a program module. The specifications are used as a communication medium among the designers and the implementors to insure both that an implementor understands the designer's intentions about a program module he is coding, and that two implementors agree about the interface between their modules. Finally, the correctness of the program is proved in the hierarchical fashion described earlier. The method of proof may be either formal or informal, and the proofs can be carried out as the modules are developed, rather than waiting for the entire program to be coded. Progress in developing formal specification techniques will enhance the practicality of applying this methodology to the construction of large programs.

II. CRITERIA FOR EVALUATING SPECIFICATION METHODS

An approach to specification must satisfy a number of requirements if it is to be useful. Since one of the most important goals of specification techniques is to permit the writing of specifications for practical programs, the criteria described below include practical as well as theoretical considerations.

[3] See the paper by Hoare [10] for a structured programming example in which specifications are emphasized.

We consider that the first criterion must be satisfied by any specification technique.

1) Formality: A specification method should be formal, that is, specifications should be written in a notation which is mathematically sound. This criterion is mandatory if the specifications are to be used in conjunction with proofs of program correctness. In addition, formal specification techniques can be studied mathematically, so that other interesting questions, such as the equivalence of two specifications, may be posed and answered. Finally, formal specifications are capable of being understood by computers, and automatic processing of specifications should be of increasing importance in the future.

The next two criteria address the fundamental problem with specifications—the difficulty encountered in using them.

2) Constructibility: It must be possible to construct specifications without undue difficulty. We assume that the writer of the specification understands both the specification technique and the concept to be specified. Two facets of the construction process are of interest here: the difficulty of constructing a specification in the first place, and the difficulty in knowing that the specification captures the concept.

3) Comprehensibility: A person trained in the notation being used should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is intended to describe. Here (and in criterion 2) we have a subjective measure in mind in which the difficulty encountered in constructing or reading a specification is compared with the inherent complexity (as intuitively felt) of the concept being specified. Properties of specifications which determine comprehensibility are size and lucidity. Clearly small specifications are good since they are (usually) easier to understand than larger ones. For example it would be nice if a specification were substantially smaller than the program it specifies. However, even if the specification is large, it may still be easier to understand than the program because its description of the concept is more lucid.

The final three criteria address the flexibility and generality of the specification technique. It is likely that techniques satisfying these criteria will meet criteria 2 and 3 as well.

4) Minimality: It should be possible using the specification method to construct specifications which describe the interesting properties of the concept and *nothing more*. The properties which are of interest must be described precisely and unambiguously but in a way which adds as little extraneous information as possible. In particular, a specification must say *what* function(s) a program should perform, but little, if anything, about *how* the function is performed. One reason this criterion is desirable is because it minimizes correctness proofs by reducing the number of properties to be proved.

5) Wide Range of Applicability: Associated with each specification technique there is a class of concepts which

the technique can describe in a natural and straightforward fashion, leading to specifications satisfying criteria 2 and 3. Concepts outside of the class can only be defined with difficulty, if they can be defined at all (for example, concepts involving parallelism will not be describable by any of the techniques discussed later in the paper). Clearly, the larger the class of concepts which may be easily described by a technique, the more useful the technique.

6) *Extensibility:* It is desirable that a minimal change in a concept results in a similar small change in its specification. This criterion especially impacts the constructibility of specifications.

III. THE SPECIFICATION UNIT

The quality of a specification (the extent to which it satisfies the criteria of the preceding section) is dependent in large part on the program unit being specified. If a specification is attached to too small a unit, for example, a single statement, what the specification says may be uninteresting, and furthermore there will be more specifications than can conveniently be handled. (The specification could express no more than the following comment, sometimes seen in programs:

$$x := x + 1; \qquad \text{"increase } x \text{ by 1."})$$

A specification of too small a unit does not correspond to any useful concept. What is wanted is a specification unit which corresponds naturally to a concept, or abstraction, found useful in thinking about the problem to be solved.

The most commonly used kind of abstraction is the functional or procedural abstraction in which a parameterized expression or collection of statements is treated as a single operation. The specification for a functional abstraction is normally given by an *input–output specification* which describes the mapping of the set of input values into the set of output values.

Recent work in the area of programming methodology, however, has identified another kind of abstraction, the *data abstraction*. This comprises a group of related functions or operations that act upon a particular class of objects, with the constraint that the behavior of the objects can be observed only by applications of the operations [11].[4] A typical example of a data abstraction is a "push down stack"; the class of objects consists of all possible stacks, and the group of operations includes the ordinary stack operations, like push and pop, an operation to create new stacks, and an operation to test whether a stack is empty.

Data abstractions are widely used in large programs, although the constraint on observable object behavior has not always been followed.[5] Some examples are segments, processes, files, and abstract devices of various sorts, in addition to the more ordinary stacks, queues, and symbol tables. In each case the implementation of the abstraction is given in the form of a multiprocedure module [14]. Each procedure in the module implements one of the operations; the module as a whole may provide a single object (for example, there is a single system data base), some fixed maximum number of objects (for example, there is a fixed maximum number of segments), or as many objects as users require (for example, a new stack is provided whenever a user asks for one).

The realization that a multiprocedure module is important in system design preceded the identification of the multiprocedure module as an implementation of a data abstraction.[6] It is illuminating to examine the arguments in favor of the multiprocedure module as an implementation unit. The procedures are grouped together because they interact in some way: they share certain resources (for example, a data base which only they use, and possibly some real resource, like the real-time clock owned by the process abstraction in [15]); and they also share information (for example, about the format and meaning of the data in the shared-data base, and the meaning of the states of the shared resource). Considering the entire group of procedures as a module permits all information about the interactions to be hidden from other modules [16]: other modules obtain information about the interactions only by invoking the procedures in the group [14]. The hiding of information simplifies the interface between modules, and leads directly to simpler specifications because it is precisely the interface which the specifications must describe.

As an example of the problems which arise when the data abstraction is ignored and the operations in the group are given input–output specifications independently of one another, consider the following specification for the operation push. Assuming the push operation is a function,

$$\text{push: stack} \times \text{integer} \rightarrow \text{stack}$$

the input–output specification must define the information content of the output value of push (the stack object returned by push) in terms of the input values of push (a stack object and an integer). This can be done by defining a structure for stack objects, and then describing the effect of push in terms of this structure. A typical stack structure might be (in PASCAL [17])

```
type stack = record top:  integer,
                    data: array [1···100] of integer
             end
```

and then the meaning of

$$t := \text{push}\ (s,i)$$

could be stated (using notation developed by Hoare [2])[7]

[4] Morris has discussed some criteria for determining what constitutes a sufficient set of operations [12].

[5] The constraint has been followed in the Venus system [13].

[6] It is an open question whether every multiprocedure module implements a data abstraction. We believe that the correspondence holds. In the Venus system [13], which was built entirely from such modules, every module did correspond to a data abstraction.

[7] This specification ignores the behavior of push if the stack is full, that is if s.top = 100.

$$\underline{\text{true}}\ \{t := \text{push}\ (s,i)\}\ \forall j[1 \leq j \leq \text{s.top} \supset \text{t.data}[j] = \text{s.data}[j] \ \&\ \text{t.data}[\text{t.top}] = i \ \&\ \text{t.top} = \text{s.top} + 1].$$

A similar specification could be given for pop.

There are several things wrong with such a specification. A serious flaw is that it does not describe the concept of stack-like behavior, but instead specifies a lot of extraneous detail. Concepts of stack-like behavior—for example, a theorem stating that pop returns the value most recently pushed on the stack—can only be inferred from this detail. The inclusion of extraneous detail is undesirable for two reasons. First, the inventor of the concept must get involved in the detail (which is really implementation information), rather than stating the concept directly. Second, the inclusion of the detail detracts from the minimality (as defined in the criteria) of the specification, and it is likely that a correctness proof of an implementation of push and pop based on a different representation for stack objects would be difficult. Another problem is that the independence of the specifications of push and pop is illusory; a change in the specification of one of them is almost certain to lead to a change in the specification of the other. For example, in addition to being related through the structure chosen for stack objects, the specifications of push and pop are also related in their interpretation of this structure: the decision to have the selector "top" point to the topmost piece of data in the stack (rather than to the first available slot).

If a data abstraction such as stack is specified as a single entity, much of the extraneous detail (concerning the interactions between the operations) can be eliminated, and the effects of the operations can be described at a higher level. Some specification techniques for data abstractions as a unit use input–ouptut specifications to describe the effects of the operations, but these specifications are expressed in terms of abstract objects with abstract properties instead of the very specific properties used in the example above. In other techniques, it is not even necessary to describe the individual operations separately, but instead, the effects of the operations can be described in terms of one another. As an example, just to convey a feeling for the latter approach, the effect of pop might be defined in terms of push by

$$\text{pop}\ (\text{push}\ (s,v)) = v$$

which states that pop returns the value most recently pushed.

In the remainder of the paper, we will concentrate on specification techniques for data abstractions. In doing this we will not ignore input–output specifications since these form a part of some of the techniques we will discuss, but we will also discuss techniques, like the one illustrated above, that are applicable only to data abstractions. We limit our attention in this way because the specification techniques for data abstractions are all fairly recent, and have received relatively little attention so far. Also, the information-hiding aspect of data abstractions, discussed above, promises that specification techniques focused on such units will satisfy the criteria very well.

CREATE	:	→ STACK
PUSH	:	STACK X INTEGER → STACK
POP	:	STACK → STACK
TOP	:	STACK → INTEGER

Fig. 3. Operations of the stack abstraction and their functionality.

IV. PROPERTIES OF SPECIFICATIONS OF DATA ABSTRACTIONS

Although the specification techniques to be described in the next section differ from one another in many particulars, there are also ways in which they are similar. All the techniques must convey the same information—information about the meaning of data abstractions—and this information is conveyed in a mathematical way. In this section, we discuss a mathematical view of the specification techniques, and the information contained in the specifications. We also discuss some of the problems arising from discrepancies between the mathematical and programming views of data abstractions.

All the specification techniques for data abstractions can be viewed as defining something very like a mathematical *discipline*; the discipline arises from the specification of the data abstraction in a manner not unlike the way in which number theory arises from specifications, like Peano's axioms, for the natural numbers. The *domain* of the discipline—the set on which it is based—is the class of objects belonging to the data abstraction, and the operations of the data abstraction are defined as mappings on this domain. The theory of the discipline consists of the theorems and lemmas derivable from the specifications.

The information contained in a specification of a data abstraction can be divided into a semantic part and a syntactic part. Information about the actual meaning or behavior of the data abstraction is described in the semantic part; the description is expressed using a vocabulary of terms or symbols defined by the syntactic part.

The first symbols which must be defined by the syntactic part of a specification identify the abstraction being defined and its domain or class of objects. Usually, an abstraction has a single class of defined objects, and, in this case, it is conventional to use the same symbol to denote both the abstraction and its class of objects. Thus the objects belonging to the data abstraction, stack, are referred to as stacks. (It is possible for an abstraction to have more than one class of defined objects, but this presents no mathematical difficulties, and we will not consider it further [18].)

The remaining symbols introduced by the syntactic part name the operations of the abstraction, and define their functionality—the domains of their input and output values. An example is shown in Fig. 3, where the functionality of the operations of the data abstraction, stack, is described. (In Fig. 3, the operation, TOP, returns the

value in the top of the stack without removing it; POP removes the value without returning it.)

Several interesting observations can be made about this example. First, more than one domain appears in the specification in Fig. 3. In practice, the specifications for almost all interesting data abstractions include more than one domain. Normally, only one of these (the class of stacks in the example) is being defined; the remaining domains (integer in the example) and their properties are assumed to be known. Of course, the specifications must clearly distinguish between the domains assumed to be known and the ones to be defined.

A second observation is that, given this distinction, the group of operations can be partitioned into three blocks. The first block, the *primitive constructors*, consists of those operations that have no operands which belong to the class being defined, but which yield results in the defined class. This block includes the constants, represented as argumentless operations (for example, the CREATE operation for stacks). The second block, the *combinational constructors*, consists of those operations (PUSH and POP in the example) which have some of their operands in and yield their results in the defined class. The third block consists of those operations (TOP for stacks) whose results are not in the defined class.

A third observation is that the mathematical description of the functionality of an operation does not necessarily correspond to the way the operation would be programmed. One difference is that the functions in the example have only one output value, while in practice it is often desirable for a program to return more than one result. For example, one might define a stack operation

$$\text{POP2: STACK} \rightarrow \text{STACK} \times \text{INTEGER}$$

which removes a value from a stack, and returns both the new stack and the value. This operation can be modeled mathematically by a pair of operations, one for each result. For example, the result of POP2 can be defined as the pair of results from POP and TOP, where both are applied simultaneously to the same stack value. When such an association is made, the specification must clearly indicate the relationship between the operation symbols.

A more serious discrepancy is that the operations are viewed by the specification as acting on time-invariant, mathematical values, but the objects found in most programming languages can be modified in some way. These modifications are the result of side effects in some of the applicable operations. For example, although the PUSH operation used above is purely functional, it would more likely be implemented so that no result is returned, and PUSH modifies (has a side effect upon) an existing stack object.

The now conventional solution to this difficulty is to factor a modifiable object into two components: an object identity (unique for each distinct object) and a current state. The modifications affect only the state component, so a given object (over time) is represented by a sequence of pairs of values in which the object identity is always the same. Each operation with a side effect is defined by a mapping which yields a new pair of values representing the same object and a new state.

There are two frequently occurring cases in which the identity component of an object can be omitted in the specifications. First, if there is only one object, such as in the KWIC index example described by Parnas [19], then the identity component is obviously redundant. Second, if, as is the case in certain programming languages, the identity of an object is uniquely given by the symbolic name of identifier that denotes the object, then a separate identity component is unnecessary. The symbolic name of an object becomes its identity, and the use of a new symbolic name implies that a new object is introduced.[8] This approach is unsatisfactory for the many languages in which a given object may have two or more distinct symbolic names; for example, an object may be accessible both via a parameter and a global name. Then the approach fails because side effects will not appear under both names (see for example, [20]).

The semantic part of the specification uses the symbols defined in the syntactic part to express the meaning of the data abstraction. Two different approaches are used in in capturing this meaning: either an abstract model is provided for the class of objects and the operations defined in terms of the model, or the class of objects is defined implicitly via descriptions of the operations.

In following the abstract model approach, the behavior is actually defined by giving an abstract implementation in terms of another data abstraction or mathematical discipline, one whose properties are well understood. The data abstraction being used as the model also has a number of operations, and these are used to define the new operations. The complexity of the descriptions depends on how closely the new operations match the old ones. Sometimes they match very closely; at other times the descriptions can be arbitrarily complex.

The approach of defining the objects implicitly via descriptions of the operations is much closer to the way mathematical disciplines are usually defined. The domain or class of objects is determined inductively. Usually it is the smallest set closed under the operations. Only those operations identified above as constructors are used in defining this closure. The closure is the smallest set which contains the results of the primitive constructors and the results of the combinational constructors when the appropriate operands are drawn from the set. For example, with stacks, the only primitive constructor is the constant operation CREATE which yields the empty stack, and the class of stacks consists of the empty stack and all stacks that result from applying sequences of PUSH's and POP's to it. One difficulty with the implicit definition approach is that if the specifications are not sufficiently complete, in the sense that all the relationships among the operations are indicated, several distinct sets

[8] See, for example, Hoare's rule of assignment [2].

may be closed under the operations. The distinct sets result from different resolutions of the unspecified relationships.

In the next section, specification techniques employing both the abstract model and the implicit definition approaches will be discussed.

V. SPECIFICATION TECHNIQUES

In this section we present a survey of selected techniques for giving formal specifications of data abstractions. This survey is not complete, but it is intended to be illustrative. We do not describe the techniques in enough detail for the reader to be able to immediately apply them; indeed, achieving such a description is a matter of research for at least some of the techniques. Rather, our intention is to introduce the most promising formal techniques, to indicate their strengths and weaknesses, and to provide pointers into the literature so that more information can be obtained.

Of the many techniques by which a data abstraction can be specified, most do not meet the criteria set forth in Section II because they are either too informal, or too low level. Thus, textual (English) specifications and specifications in terms of an implementation, such as the class definitions of SIMULA 67 [21], will not be considered. In addition, a number of techniques developed for specifying the semantics of programming languages—though relevant in varying degree—are not considered because of their specialized use. The techniques that are discussed and which seem most promising are those which use some form of abstraction to reduce the complexity of the specifications.

The techniques fall into five categories which are (in order of increasing abstractness of the specifications): use of a fixed domain of formal objects, such as sets or graphs; use of an appropriate, but otherwise arbitrary, known formal domain; use of a state machine model; use of an implicit definition in terms of axioms; and use of an implicit definition in terms of algebraic relations. Techniques in the first two categories use the abstract model approach, while those in the remaining categories use the implicit definition approach. Each of the categories is illustrated by one particular technique chosen to be typical of the category and, where possible, to be accessible in the literature. Following the description of the example, the technique is evaluated with respect to the criteria of Section II. Finally, we summarize the evaluations, and compare the categories with one another.

Use of a Fixed Discipline

We begin by discussing specification techniques in which a fixed language—that of some established mathematical discipline—is used for all specifications. The given discipline is used to provide a high-level (abstract) implementation or model of the desired data abstraction. The class of objects is represented by a subset of the mathematical domain and the operations are defined in terms of the operations on that domain. Although any mathematical discipline (number theory, analysis) might be used, practical usage has been restricted primarily to graphs [22]–[24], sets [25]–[27], and the theory developed around the Vienna Definition Language [28].

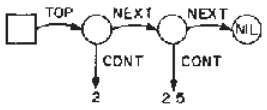

Fig. 4. *V*-graph representation for a stack.

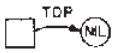

Fig. 5. *V*-graph representing the initial stack configuration.

As an example of using a fixed discipline, we will consider Earley's use of graphs in describing data structures [22]. Each instance of a data structure is represented by a graph or, as he called it, a *V*-graph. These are constructed from atoms, nodes, and links. Atoms represent data with no substructure. Links are given labels, called selectors, and are directed from nodes to nodes or atoms; the only requirement on links is that two links with the same selector can not emanate from the same node. The selectors can be any node or atom (strings, integers). Nodes have no significance other than as place holders in the structure being described; in our discussion, we will display nodes as circles, except that header nodes will be displayed as boxes. For example, a representation of a stack holding the integers 2 and 25 is shown in Fig. 4; the structure has a single header node, and the node labeled NIL is a special terminator. The values stored in the stack are accessible via the selector, CONT.

Once a *V*-graph representation has been chosen, two methods are available for defining the operations. First, operations may be defined by expressions written in terms of primitive *V*-graph operations. These operations provide the ability to use the selectors to access and modify the links and nodes. Thus, the stack operation TOP can be defined directly to access the contents of the node selected by the selector TOP.

A second definition method is used to describe operations which modify the structure of the representing *V*-graphs. These operations are defined by means of pictures of *V*-graph transformations. The operations could be described by complicated expressions in terms of the primitive operations; however by using pictures, a more minimal description, containing less extraneous detail, can be achieved. For example, the stack operations PUSH, POP, and CREATE are defined via transformations. First, an initial configuration is defined to represent the empty stack produced by CREATE; this is shown in Fig. 5. Then, PUSH and POP are defined by giving before and after pictures for the corresponding transformations. The left-hand *V*-graph dis-

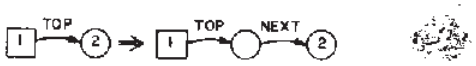

Fig. 6. *V*-graph specification for PUSH.

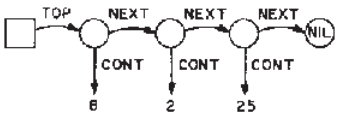

Fig. 7. *V*-graph resulting from PUSHing 8 onto the stack shown in Fig. 4.

plays a pattern, in the form of a path of selectors from a header node to other nodes, to match against the operands of the transformation. Some of the nodes in the left-hand *V*-graph are given labels which can be used to identify the new position of these nodes in the rearrangement defined by the right-hand *V*-graph, which represents the result of the transformation. For example, Fig. 6 describes the operation PUSH as follows: for any arbitrary stack object, PUSH causes a new node to be inserted between the header node and the node previously connected to it via the link labeled TOP; the value being PUSHed will be on the CONT link of the newly added node. Fig. 7 displays the result of PUSHing 8 onto the stack shown in Fig. 4. A similar definition can be given for POP; it would show POP to be the inverse of PUSH (the arrow in Fig. 6 would be reversed).

The technique of using a fixed discipline to express the specifications satisfies many of the criteria set forth in Section II. Certainly, it can be made sufficiently formal. For someone familiar with the given discipline, the specifications are usually easily understood and easily constructed if they describe concepts within the range of applicability of the chosen discipline. Extensibility presents no problem provided that the representation selected for the class of objects of the abstraction is adequate to express the properties of the extension. Even proofs of correctness of the uses of the specifications are simplified by using the multitude of theorems which exist for established disciplines.

However, techniques using a fixed discipline are deficient with respect to the criteria of minimality and range of applicability. Using such a technique to express specifications is similar to writing programs in a programming language which provides a single data structuring method;[9] although a single method can be powerful enough to implement all user-defined data structures, it does not follow that all data structures are implemented with equal facility. Similarly, we cannot expect that all data abstractions can be specified equally well in terms of a fixed discipline. For example, the graphical representation is very suitable for showing the paths by which the content of a data structure can be accessed. But, if the access path is not relevant, such as when testing whether an object is in a given set, then the graphical representation over-specifies the desired structure; that is, the abstract representation introduces details which need not be preserved in an implementation capturing the specifier's intentions. The use of extra details violates the criterion of minimality and places a practical limit on the range of applicability of a fixed discipline.

Use of an Arbitrary Discipline

The unwanted representational detail which results from using a fixed discipline can be reduced by allowing the specifications to be written in any convenient discipline. This approach is particularly useful when the class of objects of the desired data abstraction is a subset of some established mathematical domain. Hoare has used this approach to specify sets [29], [30] and certain subsets of the integers [30]. The operations on the data abstraction are defined by expressions in the chosen discipline. For example, an operation to insert an integer in a set might be defined by

$$\text{insert } (s,i) \equiv s := s \cup i$$

where assignment is used to show that s is updated with a side effect.

Many of the properties of specifications in which an arbitrary discipline is chosen are the same as when a fixed discipline is used. Allowing the specifier to choose a convenient discipline removes some of the limitations of a fixed discipline, but not all. Actually, the number of disciplines available for use is not large, and, in addition, if a completely free choice of discipline could be made it is doubtful that the resulting specifications would be comprehensible. Thus, in reality, the specifier must choose among a small number of disciplines; some of these might be existing mathematical disciplines, while others would be disciplines developed especially for use in specifications. This situation is analogous to writing programs in a language providing several data structuring facilities; programming experience indicates that there will always be (problem oriented) abstractions which cannot be ideally represented by any of the data structuring methods. Thus, it appears unlikely that all data abstractions can be given minimal specifications by choosing among a number of disciplines.

Use of a State Machine Model

As was noted in Section IV, the class of objects can be defined implicitly rather than by means of an explicit model. If the class of objects is viewed as states of an abstract (and not necessarily finite) state machine,[10] then the class can be defined implicitly by characterizing the states of the machine. Parnas [31] has developed a technique and notation for writing such specifications. The basic idea is to separate the operations into two groups: those which do not cause a state change but allow some aspect

[9] In fact, Earley defined a programming language, VERS, in which *V*-graphs were the data structuring method [22].

[10] In this case, the set of states of the state machine is the set of time-invariant mathematical values that we discussed in Section IV.

```
V-operation: TOP
possible values: integer ; initially undefined
parameters: none
effect: error call if 'DEPTH' = 0

O-operation: PUSH (a)
possible values: none
parameters: integer a
effect: error call if 'DEPTH' = MAX
        else [TOP = a ; DEPTH = 'DEPTH' + 1]
```

Fig. 8. Partial state-machine specification for the stack abstraction.

of the state to be observed—the value returning or V-operations—and those which cause a change of state—the operate or O-operations. The O-operations correspond to the constructors of Section IV. The specifications are given by indicating the effect of each O-operation on the result of each V-operation. This implicitly determines the smallest class of states necessary to distinguish the observable variations in the values of the V-operations. It also determines the transitions among these states caused by the O-operations.

We again use the integer stack data abstraction as an example, and consider the operations TOP and PUSH. TOP is a V-operation which is defined as long as the stack is not empty, and PUSH is an O-operation which affects the result of TOP. Looking at just these two operations, the state machine specifications might read as shown in Fig. 8, where DEPTH is another V-operation whose definition is not shown here, but which is intended to reflect the number of integers on the stack, and MAX represents the maximum number of integers which can be stored on the stack. Quotes around an operation name are used to indicate its value before the O-operation is executed.

This type of specification is different from those previously considered because it is free of representational details. No extra information is introduced if the specifications are expressed entirely in terms of the names of operations, types, and possibly some initial values (like MAX in the definition of PUSH). Thus, one might expect to achieve quite reasonable minimality. In practice, however, it is not always easy to build a simple description of the effect of an O-operation. The problem is that certain O-operations may have "delayed effects" on the V-operations: some property of the state will be observable by the V-operation only after some other O-operation has been applied. For example PUSH has a delayed effect on TOP, in that the former top-of-stack element is no longer directly observable by TOP, but will again be observable after POP is applied. Parnas used an informal language to describe this delayed effect [31]. Delayed effects can be described formally by introducing "hidden functions" to represent aspects of the state which are not immediately observable. Users of the state-machine model [6], [7] have made extensive use of such hidden functions. However, adding hidden functions can also add representational detail, and thus detract from the minimality of the specification.

The state-machine specifications are slightly deficient with respect to the other criteria of Section II. Because of

```
1  CREATE (STACK)
2  STACK(S) & INTEGER (I)  ⊃ STACK (PUSH (S, I)) &
                             [POP(S) ≠ STACKERROR ⊃ STACK (POP(S))] &
                             [TOP(S) ≠ INTEGERERROR ⊃ INTEGER (TOP(S))]
3  (∀A) [A(CREATE) &
            (∀S)(∀I) [STACK (S) & INTEGER (I) & A(S)
                  ⊃ A(PUSH (S, I)) & [S ≠ CREATE ⊃ A(POP (S))]]
          ⊃ (∀S) [STACK (S) ⊃ A(S)]]
4  STACK(S) & INTEGER (I) ⊃ PUSH (S, I) ≠ CREATE
5  STACK(S) & STACK(S') & INTEGER(I)
          ⊃ [PUSH (S, I) = PUSH (S', I) ⊃ S = S']
6  STACK(S) & INTEGER (I) ⊃ TOP (PUSH (S, I)) = I
7  TOP(CREATE) = INTEGERERROR
8  STACK (S) & INTEGER (I) ⊃ POP (PUSH (S, I)) = S
9  POP(CREATE) = STACKERROR
```

Fig. 9. Axiomatic specification of the stack abstraction.

the problem of delayed effects noted above, they are sometimes difficult to construct. Because the O-operations which change the result of a V-operation are totally separated from that V-operation, the specifications are sometimes difficult to read. The separation also affects extensibility since adding a new V-operation may require updates to a large portion of the O-operation specifications.

With respect to the criterion of formality, we expect that state-machine specifications can be given an adequate formalization but much work remains to be done. In particular, it is necessary to develop a formal (not necessarily effective) construction for the state machine specified by a given set of specifications. This will necessitate defining the language which can be used to describe the effects of an O-operation. In addition, work on developing the proof methodology to use with state-machine specications is needed. Price [6] has proven a number of properties of a particular data abstraction, but the methodology for proving the correctness of an implementation still needs to be developed. Some of the needed formalization is being done in an ongoing project at SRI [7], [32].

Use of Axiomatic Descriptions

An alternative to using state machines to implicitly determine a data abstraction is to give a list of properties possessed by the objects and the operations upon them. This approach can be formalized by expressing the properties as axioms for the data abstraction. Axiomatization has been used by Hoare [2], [33] to define the built-in data types of a programming language. The technique can also be used to give specifications for user-created data abstractions.

An axiomatization of the integer stack abstraction in which popping the top element off the stack (POP) and examining the top element (TOP) are separate operations, is given in Fig. 9. In this example, STACK and INTEGER are predicates; STACK is being defined, but INTEGER is assumed to be defined elsewhere. The axioms are written in a form analogous to Peano's axioms for the natural numbers. Axioms 1 and 2 define the range of the applicable operations. Axiom 3 is the induction axiom which limits the class of stacks to those that can be constructed with the given operations. Axioms 4 and 5 insure the distinctness of the results of the PUSH operation. Axioms 6 and 7

define the result of the TOP operation and Axioms 8 and 9 define the result of POP. Axioms 7 and 9 capture the fact that neither TOP nor POP may be legally applied to an empty stack (the result of CREATE).[11]

The axioms determine an abstract representation for stacks in the following manner. Consider the set of all legal expressions that can be constructed from the given operations. This set of expressions names every possible member of the class of stacks. Some pairs of expressions may name the same stack, however; for example, both

PUSH (CREATE, 7) and

POP (PUSH (PUSH (CREATE, 7), 25))

denote the same stack. Therefore, the class of stack objects is represented by equivalence classes over the set of all expressions. These equivalence classes are determined (noneffectively, in general) by the axioms.

If the axioms are sufficiently well chosen, the equivalence classes are unique. If not, then several sets of equivalence classes may satisfy the axioms. If, for example, Axiom 4 is omitted, then two distinct sets of equivalence classes—one in which the result of PUSH is always distinct from the empty stack and one in which it is not—would both satisfy the axioms.

The axiomatic specifications can almost always be minimal and widely applicable, in part because there are so few limitations on the form of the axioms. In addition, the approach seems to support extensibility, since, in most cases, it suffices to add new axioms to describe the extended concept, or at most, to modify a few existing axioms. The formalization of the axiomatic technique is borrowed directly from existing mathematics. Proving the correctness of an implementation of a data abstraction specified by axioms means showing that the implementation is a model of the axioms.

The axiomatic approach is most seriously deficient with respect to the criteria of comprehensibility and constructibility. As discussed in Section IV, the approach does not directly define a model for the class of objects; instead the class is defined only implicitly. It is sometimes difficult to see that the axioms really define the set of values of interest. In addition, the possibility that several very different sets of values may satisfy the axioms is disturbing.

Use of Algebraic Definitions

It is reasonable to expect that all data abstractions one might be interested in implementing on a computer would have finitely constructible, countable domains. In view of this, the first three axioms in Fig. 9 can be omitted, providing suitable notation is developed to indicate the group of applicable operations and their functionality. Algebraic specifications [18] provide such a notation.

[11] In these axioms, we are using the standard mathematical technique for making a partial function total: the output domain of the function is extended by one special, recognizable value which will be the result of the function in all cases where it was previously undefined.

```
Functionality:
    CREATE :                  → STACK
    PUSH   : STACK × INTEGER  → STACK
    TOP    : STACK            → INTEGER ∪ INTEGERERROR
    POP    : STACK            → STACK ∪ STACKERROR

Axioms:
  1' TOP(PUSH(S,I)) = I
  2' TOP(CREATE) = INTEGERERROR
  3' POP(PUSH(S,I)) = S
  4' POP(CREATE) = STACKERROR
```

Fig. 10. Algebraic specification of the stack abstraction.

The algebraic specification technique is based on a generalization of the algebraic construction known as a *presentation*. A presentation of the stack abstraction is shown in Fig. 10. Only four axioms are now needed (labeled with primes to avoid confusion with the axioms in Fig. 9). Axioms 1–3 are replaced by the definition of functionality; this is sufficient to define the set of legal, finitely constructible expressions in these operations. In the usual algebraic terminology, the legal expressions are called *words*. Next, it is necessary to specify which of these expressions are to yield equivalent results, through a set of defining axioms referred to as *relations* or relation schemata: this is done by Axioms 1'–4' (which correspond to Axioms 6–9 in Fig. 9). The construction which gives meaning to a presentation automatically forces all expression pairs which cannot be shown to be equivalent to be distinct. This simplifies the expression of the specifications and is why Axioms 4 and 5 are not needed.

Almost all the comments about how axiomatic definitions satisfy the criteria apply equally well to algebraic definitions. Algebraic and axiomatic definitions are equally good with respect to the criteria of minimality, wide range of applicability, and extensibility. (Algebraic definitions are shorter than axiomatic ones, but they are not more minimal because they express the same information.) The algebraic approach can be easily formalized by borrowing from existing mathematics; most results carry over in a straightforward manner, although some generalization is needed to treat several existing domains simultaneously. For algebraic specifications, proving the correctness of an implementation means showing that it defines an isomorphic image of the presented algebra. This isomorphism can be established implicitly by showing that the defining axioms hold in the implementation and that the mapping is one–one [18], [34].

The algebraic approach is superior to the axiomatic approach with respect to the criteria of constructibility and comprehensibility, because the approach is more structured. However, algebraic specifications are still deficient with respect to these criteria. Although use of the algebraic approach precludes the possibility of more than one set of values satisfying the axioms, it is still possible that the set of values defined is not the one intended. We believe this difficulty can be eased if a methodology is developed which can be applied to constructing and understanding such specifications. Some progress in this direction has been made [18], [35], but more work is needed.

Summary of Analyses

The analyses given in this section indicate that there is no single specification technique that is universally better than the others. One major difference among the techniques is the extent to which they exhibit a *representational bias*, that is, the extent to which the specifications suggest a representation or implementation for the abstractions being defined. The representational bias of a technique determines, in large measure, its range of applicability. Techniques having a representational bias will be limited primarily to those abstractions which are naturally expressed in the representation; however, within the range, specifications will be fairly easy to construct and comprehend, and reasonably minimal. Those techniques which make use of an existing mathematical discipline to specify an abstract model for the class of defined objects have a representational bias. Such techniques will be preferred for abstractions which fit nicely into the discipline (for example, where the objects of the abstraction are elements of an existing domain).

The techniques providing an implicit definition of the class of objects have no representational bias, and will clearly be preferable for those abstractions not well matched to an existing discipline. They may sometimes be preferred even when one of the abstract model approaches could be used. The abstract model approaches tend to suggest an implementation for the abstraction, and this may be undesirable, not because it precludes very different implementations, but because it may be hard for the implementor to find a different but better implementation.

All the implicit definition techniques, with their lack of representational bias, have a wide range of applicability, but they vary in the extent to which they satisfy the criteria of minimality, constructibility, and comprehensibility. The difficulty in the state-machine approach of coping with delayed effects reduces the minimality and constructibility of the specifications, though not necessarily the comprehensibility. The introduction of hidden V-functions may impact the free choice of an implementation, since the implementor may feel the need to implement these hidden functions, which is not necessary. Algebraic and axiomatic specifications are more minimal than state-model specifications, but they may be more difficult to construct and understand.

The state-machine technique appears to be least satisfactory with respect to the criterion of extensibility, because introducing a new V-operation is likely to necessitate changes to the definitions of many O-operations. However, the criterion of extensibility, based on the notion of a "small" change to the concept, is really quite vague. Perhaps a small change is one requiring only a minor modification to the specification. Also, the different specification techniques may tolerate different kinds of changes, and this could be a factor in choosing a technique.

The criterion of formality is not entirely satisfied by any of the techniques, although the state-machine model is the least formalized. There are two important aspects to formalization. First, the syntax and semantics of the language in which the specifications are written must be fully defined. Defining the semantics involves more than just defining the meaning of each symbol; a construction (it may be noneffective) of the defined class of objects from the specification must also be provided. This is only difficult in the implicit definition approaches; in the abstract model approaches the specification describes the objects explicitly. Second, a methodology for proving that an implementation satisfies a specification must be provided. Additional work on formalization would expand the usefulness of the techniques. Unless a technique is adequately formalized, it will be difficult, if not impossible, to train people to use it correctly and coherently.

We conclude by discussing one previously unmentioned aspect of specification techniques: the extent to which they capture all interesting properties of a data abstraction. For example, consider the treatment of errors in the various specification techniques. In some techniques, errors are completely ignored. In others, notably the axiomatic and algebraic techniques, the presence of errors is acknowledged, but not in a particularly illuminating way. The solution of adding an extra error element to the output domain, while mathematically sound, does not provide the kind of information that a user of the abstraction requires. A more realistic approach is taken by the state-machine technique; here, error cases are prominently displayed, different errors can be given meaningful names (although this was not shown in the example), and even the order in which errors will be recognized by a given operation can be specified. It is noteworthy that this technique is based on a model of the way errors will be handled in running programs; such a model may be necessary if errors are to be specified in a realistic manner. The treatment of errors is not the only example where the specification techniques are deficient (e.g., performance requirements are also missing). Much more work is needed to identify the interesting properties of data abstractions, and to develop the specification techniques to express those properties.

VI. CONCLUSIONS

A major premise of this paper has been that formal specifications should come to play a fundamental role in the construction of reliable software. Two reasons were given for this: 1) The growing economic pressure for reliable programs indicates that increased effort in this direction is justified, and 2) the recognition of a new kind of module—the multiprocedure module—has led to the identification of a specification unit for which specifications are practical. This kind of module is helpful in the construction of software, because it permits data abstractions to be used in building programs. Since data are the fundamental concern of programs, we can expect the use of data abstractions to be widespread.

To indicate the form such specifications might take, Section V discussed several specification techniques. The

techniques discussed were promising in that they did succeed in describing data abstractions at a reasonably abstract level. However, none of the techniques are ready to be applied to practical programs. Some techniques have not yet been put on a firm mathematical basis (although we believe that all the techniques surveyed are capable of being adequately formalized). Other techniques ignore a fundamental aspect of data abstractions: how to cope with errors and exceptions. Finally, none of the techniques has been applied widely enough that its expressive power can be evaluated. Recent uses of the state-machine technique of Parnas to specify operating systems [7] or parts thereof [6] may indicate that that technique is suitable for systems of interesting size, but the complexity of at least one of those specifications [6] indicates the specification technique requires further refinement. It is reasonable to expect deficiencies in the other specification techniques to emerge when they are likewise applied to large programs.

Some deficiencies in the techniques are already apparent. The range of applicability of the various techniques is often smaller than we would like; examples were discussed in Section V. Since the range of applicability is different for the different techniques, we may expect that using a combination of techniques when describing a large program would be a profitable approach. However, there are programs whose meaning cannot be captured by any of the described techniques. For example, specifications using the techniques cannot be given for programs involving parallel activity. We chose not to survey work going on in developing specification techniques to handle parallelism because the work is very recent and quite preliminary. However, one promising approach uses data abstractions as the specification units [36].

The specification techniques discussed in this paper can adequately describe modules—the blocks out of which systems are built—but it is not clear that they can describe the entire system. For example, Parnas has shown how a KWIC sytem can be modularized [16], and each module was described using his specifications, but the specification of the system as a whole was given in English. It seems unlikely that an entire system can be viewed as a single, top-level module, so perhaps a different kind of specification technique is desirable here.

Even if we are not able to describe an entire system using the specification techniques, the ability to define most of the modules used in constructing a system in a precise, formal way would be a major advance in the construction of reliable software. The specification techniques discussed in this paper are all quite recent; much is being accomplished by concentrating on the data abstraction as a specification unit. This general area appears to be a very promising one for further study: work in applying existing techniques to large programs, in extending and formalizing existing techniques, and in proposing new techniques, for both sequential and parallel programs, is of the utmost importance.

ACKNOWLEDGMENT

The authors greatfully acknowledge the helpful suggestions made by J. Dennis and the referees.

REFERENCES

[1] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Mathematics*, vol. XIX, Mathematical Aspects of Computer Science, American Mathematical Society, Providence, R.I., 1967, pp. 19–32.
[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Compt. Mach.*, vol. 12, pp. 576–580, 583, Oct. 1969.
[3] Z. Manna, S. Ness, and J. Vuillemin, "Inductive methods for proving properties of programs," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 491–502, Aug. 1973.
[4] B. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, vol. 19, pp. 48–59, May 1973.
[5] M. Schroeder, "Certification of computer systems," Mass. Inst. Technol., Cambridge, Project MAC Progress Rep. 11, to be published.
[6] W. R. Price, "Implications of a virtual memory mechanism for implementing protection in a family of operating systems," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1973.
[7] P. G. Neumann *et al.*, "On the design of a provably secure operating system," in *Proc. Int. Workshop on Protection in Operating Systems*, IRIA, Rocquencourt, France, 1974, pp. 161–175.
[8] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 2, pp. 56–73, Jan. 1972.
[9] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming* (APIC Studies in Data Processing, no. 8). New York: Academic, 1972, pp. 1–81.
[10] C. A. R. Hoare, "Proof of a program: FIND," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 39–45, Jan. 1971.
[11] B. H. Liskov and S. Zilles, "Programming with abstract data types," in *Proc. Ass. Comput. Mach. Conf. Very High Level Languages*, SIGPLAN Notices, vol. 9, Apr. 1974, pp. 50–59.
[12] J. H. Morris, "Toward more flexible type systems," in *Proc. Programming Symp.*, Paris, France, Apr. 9–11, 1974, *Lecture Notes in Computer Science*, vol. 19. New York: Springer-Verlag, pp. 377–384.
[13] B. H. Liskov, "The design of the Venus operating system," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 144–149, Mar. 1972.
[14] ——, "A design methodology for reliable software systems," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, N.J.: AFIPS Press, 1972, pp. 191–199.
[15] E. W. Dijkstra, "The structure of the 'THE'—multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 341–346, May 1968.
[16] D. L. Parnas, "Information distribution aspects of design methodology," in *Proc. Int. Fed. Inform. Processing Congr.*, Aug. 1971.
[17] N. Wirth, "The programming language PASCAL," *Acta Informatica*, vol. 1, pp. 35–63, 1971.
[18] S. N. Zilles, "Data algebra: A specification technique for data structures," Ph.D. dissertation (forthcoming), Project MAC, Mass. Inst. Technol., Cambridge, 1975.
[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1053–1058, Dec. 1972.
[20] R. M. Burstall, "Some techniques for proving correctness of programs which alter data structures," in *Machine Intelligence*, D. Michie, Ed., vol. 7. New York: Elsevier, 1972.
[21] O. J. Dahl, B. Myhrhaug, and K. Nygaard, "The SIMULA 67 common base language," Norwegian Computing Center, Oslo, Publication S-22, 1970.
[22] J. Earley, "Toward an understanding of data structures," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 617–627, Oct. 1971.
[23] A. W. Holt, "Mem-theory, a mathematical method for the description and analysis of discrete finite information systems," Applied Data Research, Inc., 1965.
[24] C. Christensen, "An example of the manipulation of directed graphs in the AMBIT/G programming language," in *Interactive Systems for Applied Mathematics*, Klerer and Reinfelds, Ed. New York: Academic, 1968.
[25] J. Earley, "Relational level data structures for programming languages," *Acta Informatica*, vol. 2, pp. 293–309, 1973.
[26] J. B. Morris, "A comparison of madcap and SETL," Los Alamos Sci. Lab., Univ. of California, Los Alamos, N. Mex., 1973.
[27] J. Schwartz, "On programming, an interim report on the SETL project," Dep. Comput. Sci., Courant Inst. Math. Sci., New York Univ., New York, 1973.

[28] A. Birman, "On proving correctness of microprograms," *IBM J. Res. Develop.*, vol. 18, pp. 250–267, May 1974.
[29] C. A. R. Hoare, "Proof of a structured program: 'The sieve of Eratosthenes'," *Comput. J.* vol. 15, pp. 321–325, Nov. 1972.
[30] ——, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271–281, 1972.
[31] D. L. Parnas, "A technique for the specification of software modules with examples," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 330–336, May 1972.
[32] P. G. Neumann, "Toward a methodology for designing large systems and verifying their properties," Gesellschaft fur Informatik, Berlin, Germany, 1974.
[33] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 335–355, 1973.
[34] S. N. Zilles, "Algebraic specification of data types," Mass. Inst. Technol., Cambridge, Project MAC Progress Rep. 11, to be published.
[35] J. Donahue, J. D. Gannon, J. V. Guttag, and J. J. Horning, "Three approaches to reliable software: Language design, dyadic specification, complimentary semantics," Comput. Sci. Res. Group, Univ. of Toronto, Toronto, Ont., Canada, Tech. Rep. 45, Jan. 1975.
[36] C. Hewitt and I. Greif, "Actor semantics of PLANNER-73," in *Proc. 2nd Ass. Comput. Mach. Symp. Principles of Programming Languages*, Palo Alto, Calif., Jan. 20–22, 1975, pp. 67–77.

Barbara H. Liskov received the B.A. degree in mathematics from the University of California, Berkeley, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, Calif.

From 1968 to 1972 she was associated with the Mitre Corporation, Bedford, Mass., where she participated in the design and implementation of the Venus Machine and the Venus Operating System. She is presently Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge. Her research interests include programming methodology and the design of languages and systems to support structured programming.

Stephen N. Zilles (S'71–M'73) was born in Toledo, Ohio, on July 1, 1941. He received the S.B. degree in economics and politital science in 1963 and in mathematics in 1967 from the Massachusetts Institute of Technology, Cambridge. In 1970, he received the S.M. and E.E. degrees in electrical engineering, also from the Massachusetts Institute of Technology.

From 1963 to 1968 he was employed by the IBM Corporation, San Jose, Calif., where he worked on the compilation of PL/I, an interactive system for building large programs and Formac, a system for algebraic manipulation. While completing the M.S. and E.E. degrees at the Massachusetts Institute of Technology he was a Teaching Assistant, and a Research Assistant in Programming Linguistics at MIT Project MAC. Upon returning to IBM in 1970 he was Advanced Programming Manager responsible for implementing a prototype language interpreter. Since 1971 he has been active in the design and evaluation of programming and command languages at IBM. His current interests are in the semantics and design of programming languages with particular emphasis on the description and representation of data types.

Mr. Zilles is the Secretary–Treasurer of the Association for Computing Machinery SIGPLAN and a member of the Association for Computing Machinery Board on Special Interest Groups and Committees. He is a member of Sigma Xi and the Association for Computing Machinery.