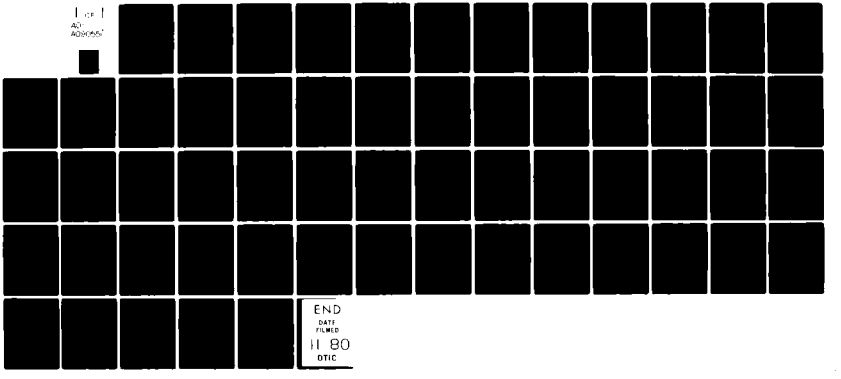
AD-A090 551 DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
PRACTICAL SUGGESTIONS FOR WRITING UNDERSTANDABLE, CORRECT FORMA--ETC(U)
AUG 80 R M WEISCHEDEL F49620-79-C-0131
UNCLASSIFIED TR-80-2 AFOSR-TR-80-1026 NL



Practical Suggestions for Writing Understandable, Correct Formal Specifications*

by

Ralph M. Weischedel

Aug 80

Technical Report #80-2

TR-80-2

2304

A2

*Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under contract no. F49620-79-C-0131 AFOSR-78-3539. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation herein.

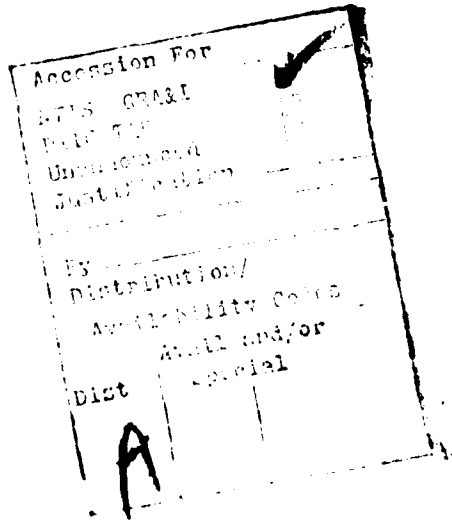
Abstract

Though formal specifications of software modules offer much toward the design problems of large software systems, creating formal specifications is very difficult, requiring much upfront effort. The first half of this report is tutorial. It describes the three major classes of formal specification languages. After arguing that understandability is critical for effective formal specifications, we discuss reasons why they are difficult to understand. Practical suggestions for making them more understandable follow from that; many of them are very familiar, since they carry over directly from structured programming. An example from pattern-matching is specified to illustrate the application of structured programming methodology to nonprocedural languages. Three practical suggestions for checking the correctness of formal specifications follow.

The second half of the paper raises and discusses three approaches to reducing the great effort in creating formal specifications. One is a library of formal specifications. One is using precise, but not mathematically rigorous, specifications. The third is the development of sophisticated tools for aiding in the creation of formal specifications.

Our suggestions have arisen from two studies. In one, we wrote several module specifications in the form that might appear in a library. The modules varied in complexity from a stack to the kernel of a text editor;

the text editor specifications ranged in length from 9 to 28 pages including copious comments as in-line documentation. The second study compared portions of the English and formal specifications of KSOS (Ford Aerospace, 1978), the Kernelized Secure Operating System.



1. Introduction

The technique of formal module specifications seems to offer much toward alleviating many problems of large software systems (those systems requiring at least 25 programmers for development and at least 30,000 lines of source code). The high cost of software maintenance, the predominance of design errors, the difficulty in modifying software, and the difficulty and cost of diagnosing and correcting design errors are some of the problems addressed by formal specifications based on the information-hiding principle. Yet, the creating of formal specifications is very difficult, requiring much upfront effort. For instance, Parnas (1976, p. 7) states, "Experience has shown that the effort involved in writing the set of specifications can be greater than the effort it would take to write one complete program."

Sections 6 and 7 examine two approaches to reducing the difficulty in writing formal specifications. One natural solution is to develop a library of formal specifications of modules. By reusing module specifications, rather than recreating them, the cost of the upfront effort would be dramatically cut. Section 6 examines the feasibility of building and using such a library.

A second approach to reducing the cost is to use semi-formal specifications that involve a mixture of formal and informal descriptions. Two possibilities for this are examined in section 7; the potentials and

drawbacks of a semi-formal approach are examined there.

In addition to the difficulty in creating formal specifications, they are very difficult to understand. In section 2 we list five reasons why they seem so difficult to understand. Section 3 includes six common sense suggestions for writing formal specifications that are more understandable; these are illustrated by a non-trivial example in section 4. Three suggestions for checking the correctness of formal specifications are given in section 5. Section 8 lists our conclusions. Rather than collecting citations to related work in one section, we have distributed references to closely related work throughout the paper.

In the remainder of this section, we clarify our terminology and describe the studies that led us to many of these suggestions.

1.1 Our Point of View and Use of Terminology

Since the meaning of module, design, specification, and library is often in the eye of the beholder, we outline our use of them here. By module we mean a collection of procedures and/or macros so closely related that they share implementation decisions and offer a single coherent interface to software using the module. The size of a module could be rather small such as the procedures implementing a stack, a symbol table, or other abstract data types (Guttag, et.al., 1978). Modules can be very large, themselves being hierarchically decomposed into other modules; an example is the provably

secure operating system (Robinson, et.al., 1977) which is a module whose suggested implementation involved 11 hierarchical levels of modules. With such large modules, the term abstract machine is often used since the implementation, according to the information-hiding principle, is a black box.

By design we mean the decomposition of a large system into precisely defined modules. In the design methodology advocated (Parnas (1972), Robinson, et.al. (1977) and others), a module specification defines precisely the interface of a module by detailing exactly what each function at the interface of a module does without committing oneself to any particular implementation. Research on these design methodologies usually involves a *formal language for* module specification because of the precise semantics, lack of ambiguity, and attention to detail of such languages. Specifications in English, no matter how well written, tend to be ambiguous and misleading. Parnas (1977) and Guttag (1980) are tutorials that argue for formal languages, rather than English, to specify a module's interface. Guttag, et.al. (1978), Robinson, et.al. (1977), Ford Aerospace (1978), and Neumann, et.al. (1977) present many examples of module specifications.

Our interest in formal specifications in this paper is in their use as a precise, unambiguous design document detailing the interfaces of modules of large systems. Therefore, our interest here is in their

communication aspects among people rather than in their potential use in program proving or automatic program synthesis. Consequently, one of the concerns of this paper is human factors issues, such as understandability and usability.

Table 1 summarizes our view of English and of formal languages as a communication device in software specification. From that comparison it is clear that both types are complementary and therefore that both are important to the design of large software systems.

Table 1

Characteristics of the Specification Languages

English	Formal
Easy to understand	Difficult to understand and write
Provide intuitive notion	Provide complete detail
Include both functional & nonfunctional requirements	Include only functional requirements (so far)
Frequently ambiguous	Unambiguous
Frequently imprecise	Precise
Frequently vague	Rigorous

The kind of library considered here would be a reference collection of module specifications for designers to browse through. For any given need, such as a file management module, the library would contain a number of alternative specifications ranging through the spectrum of functional capabilities a file management module might offer.

1.2 Basis for Suggestions

Our suggestions arise from two studies. The first involved specifying many modules in SPECIAL (Roubine and

Robinson, 1976). SPECIAL was chosen because of the availability of documentation, a syntax checker for it, and significant specifications of an operating system (Neumann, et.al. 1977) at the time of starting our work in 1977. In our study, several data structures including two kinds of stacks, four kinds of queues, and two binary trees were specified. In addition, the functional capabilities of three classes of text editors were specified by us. The specifications were prepared painstakingly with the goal that they would be patterns for the quality of entries to appear in a library of module specifications. The text editor specifications ranged from 9 to 28 pages including copious comments documenting them. In addition, implementations were written for each of the modules specified except one of the text editors.

The second study has involved comparing sections of the English specification and the formal specification of KSOS (Ford Aerospace, 1978), the kernel of a secure operating system. One emphasis here has been comparing the means of conveying information, its organization, and the type of information in the formal language (SPECIAL) and in English. KSOS was chosen, since it is one of the largest, most complex systems that has ever been formally specified and since both types of specifications were available for it. The KSOS specifications are a prodigious use of specification technology.

1.3 Classes of Formal Specification Languages

Three types of formal specification languages have emerged: operational ones, Hoare's axiomatic approach, and algebraic axioms. Surveys of the types of specification languages appear in Liskov and Berzins (1977) and Guttag (1980). Operational formal specifications are executable programs. They may be in a very high-level programming language, whose semantics is very different from conventional programming languages. Balzer and Goldman (1979) and Balzer (in preparation) develop one such language which is radically different from conventional languages. Operational specifications define the input-output behavior of an operation at the interface of a module by the input-output behavior of the executable specification. Naturally, the advantages are that one can actually check any aspect of the module's behavior by running it. A disadvantage is that one overspecifies the module. Consider defining a simple symbol table. A simple way to specify it is to write a linear search program in the operational language; yet, most assuredly one would not want to infer that linear search is a requirement for a symbol table. A second disadvantage is that the relations between operations at the interface are not stated explicitly; one must infer them from the various pieces of code.

Suppose we are specifying a queue; for expository convenience, suppose furthermore that the operational specification language has ALGOL-like semantics. Our operational specification must give some implementation

in the specification language. Let us choose the circular one, having two pointers: front, which designates the next item to leave the queue and rear, which points to the empty position for the next item to be added to the queue. A specification for the operation dequeue to remove an item from the queue is given in Figure 1. Notice that it takes an argument naming the queue and returns the value of the first element in it in addition to removing it from the queue.

Figure 1

An Operational Specification of Dequeue

```
INTEGER PROCEDURE dequeue (queue)
   BEGIN
   INTEGER ARRAY queue (1::n);
   IF front=rear THEN CALL response_to_empty_queue
      ELSE BEGIN
           dequeue <- queue (front);
           COMMENT queue names an array;
           front <- (front+1) MOD n;
           COMMENT n is the size of the array;
           END
   END
```

A second type of formal specification is more abstract than the operational ones, for the second type is not executable. Following the approach to semantics given by Hoare (1969), one specifies the preconditions (or input assertions) and the post-conditions (or output assertions) for each of the operations of the module. Several specification languages have been developed based on this approach (Ambler, et al., 1977; London, et al., 1978; and Roubine and Robinson, 1976). We will refer to this type as the Hoare axiomatic approach.

Now consider specifying a queue in the Hoare

axiomatic approach. A precondition on dequeue is that the queue is not empty. In SPECIAL (Roubine and Robinson, 1976) one would write this as an exception condition to detect that the queue is empty. The postcondition states that the first value is returned and that the queue is updated to exclude that element. Our definition appears in Figure 2. SPECIAL provides sets, vectors, and records as complex, primitive notions for use in defining entities. The construct q(queue) in lines 6 and 7 is the sequence of items held in the queue. The apostrophe in line six signifies the new value of q(queue) after dequeue has completed; lines 6 and 7 indicate that the sequence is unchanged except that the final item in the vector prior to dequeue is absent after dequeue. Empty would be defined as true if and only if the vector has length 0; front would be defined as the final item in the vector q(queue). Note that the equal sign does <u>not</u> mean assignment, but rather means equality. The EFFECTS are statements of fact; there is no significance to the order in which one states such facts.

Figure 2

An Axiomatic Specification of Dequeue

```
1  OVFUN dequeue(queue) -> INTEGER;
2     EXCEPTIONS
3        empty(queue);

4     EFFECTS
5        i=front(queue)
6        'q(queue)=VECTOR(FOR j FROM 1 TO LENGTH(queue)-1|
7                                    q(queue)[j]);
```

The third type is algebraic axioms. Instead of having axioms exclusively for each operation at the interface, the algebraic axioms state (some of) the relationships between operations. Each axiom states the result of some sequence of operations by defining it to be equivalent to some value or some other sequence of operations. The axioms may be used for substitution to reduce some complex sequence of function calls to a simpler (shorter) sequence; this suggests the name "algebraic". Examples of languages based on algebraic axioms may be found in Guttag (1980) and Musser (1980).

Both axiomatic techniques are declarative in that they tell what is done without indicating how one might implement it. Nevertheless one can simulate the functional requirements of at least certain classes of modules specified axiomatically (Musser, 1980; Levitt et al., 1979). Both of the axiomatic approaches have been used as the basis of proving that a module conforms to its specifications (Robinson and Levitt, 1977; Guttag, et al., 1978).

In specifying a queue using algebraic axioms, one

must state axioms which imply the effect of any sequence of operations. The types of dequeue's arguments and results can be stated as dequeue(queue)->queue. The axioms involving dequeue are given in Figure 3. Line 1 states that trying to remove something from a queue that you just created is an error. One must infer from other axioms not stated here that any empty queue is equivalent to one just created. Lines 2 and 3 state that if one removes an item just after adding an item to the empty queue, that the result is equivalent to a queue that has just been created (i.e. an empty queue). Lines 2 and 4 state that if one tries to remove something from a queue to which an item was just added and which was not empty when that item was added, that the result is the same as performing dequeue first and then performing enqueue. One must infer from the two axioms of figure 3 plus others for the queue that in fact all cases are defined, e.g. that the two cover the case of doing two dequeues in a row.

Figure 3

Algebraic Axioms for Dequeue

```
1  dequeue(create( )):=ERROR
2  dequeue(enqueue(item,queue)):=
3              IF empty(queue) THEN create()
4                   ELSE enqueue(item,dequeue(queue))
```

Our suggestions are valid for any specification language based on the Hoare axiomatic approach, since its semantics is clearly the crucial aspect of specification, not its syntax. We will indicate where the

suggestions obviously carry over to other types of specification languages.

2. Reasons Why Formal Specifications are Difficult to Understand

Researchers in the area of formal module specifications and abstract data types generally agree that they are difficult to understand, though the degree of difficulty is argued.

The formal specification of a module must be understandable if it is to achieve its purpose, for it acts as a contract between designers and programming team, stating exactly what the programming team's product must do (Parnas, 1977). Unless they are understandable, 1) programmers will not know what the module they are to implement is to do nor how to use other modules, and 2) designers will not be able to detect design errors nor easily confirm that their design satisfies user requirements. Also, if one is to use a reference library of formal specifications, they must be understandable, for if the designer cannot understand the alternative specifications, how can an intelligent choice be made among the alternatives? If formal specifications of module interfaces are ever to become practical, they must be understandable.

We have found several reasons for the difficulty of understanding formal specifications, particularly as compared to natural language specifications.

1) Natural language specifications usually do not

contain the detail that formal specifications do. Though this is a major reason that natural language is inappropriate for module specification, the necessary, added detail in a formal specification makes them harder to understand. Liskov and Berzins (1977) agree, stating on p. 13-5, "Rigorous informal specifications are probably just as difficult to construct as formal ones; informal specifications appear easier to construct because they are usually incomplete."

2) Natural language has myriads of concepts already defined and familiar to us to succinctly state what a module does, but formal specifications do not as yet. For example, the concepts of a sorted sequence, a pointer, a line of text, and a shared segment of memory are all well-known and are referred to without further explanation. (Yet, the advantage this gives to understandability is simultaneously a serious drawback to natural language specifications, since the notion raised in each person's mind need not be the same.) There is no corresponding body of defined concepts which have been taught us and which we have frequently used. Therefore, concepts such as sorted order must be defined in the specification, thus adding to what must be understood. To the reader of a specification, English may appear like a very high level language, whereas the formal language appears like an assembly language without any significant collection of macros or subroutines to draw on. Of course, a library of formal specifications would

provide a body of past experience to study and use just as in natural language.

3) The semantics of specification languages is quite different than programmers are used to. Formal specifications of modules are to be implementation independent. Programming languages are designed to define implementation detail. Thus, the purpose and focus of attention of specification languages are often quite different from programming languages. (The operational specification languages are an exception, since they have semantics similar to programming languages. However, Guttag (1980) argues that they have additional handicaps for understandability: the irrelevant implementation detail and inferring the relations between functions at the interface.)

4) English specifications often provide summaries of detail, whereas formal ones have not, as yet, tending to provide detail in an isolated way. For instance, in the KSOS specification (Ford Aerospace, 1978), the following summarizing statement appears on page 9, "A SEID shall be returned as the result of new object creations (i.e. K_create, K_build_segment, K_create_device, K_fork, and K_spawn). This summarizes several things: 1) that the five functions listed are the ones that create new objects and 2) that all of them return a SEID as a value. In the formal specification of KSOS, for each function, the type of the value returned, the inputs, input assumptions, and side-effects are given with

each of the functions individually. However, there is no summarizing of either of the two facts as in the English; one must induce this information from the particulars. Understanding seems to require a "cognitive framework" around which to organize particulars. Formal languages could be devised and used for that purpose; however, none seem to be used for that purpose in module specification.

5) English specifications often explicitly state facts which are only mathematically implied in formal specifications. This holds for all three types of formal languages. For instance, the English specification of K_build_segment in KSOS goes beyond the description of only that function by spelling out the sequence of other KSOS functions to call to set up shared segments of execute-only code. While one may be able to infer all of the steps from the formal specification, the need to determine so much that is implicit decreases the understandability of the formal specification. Since algebraic axioms focus attention on the relations between operations at the interface of a module, they state explicitly many facts that are only implicit in the other two types of languages. Yet, examples specified using algebraic axioms can lack understandability also by leaving crucial facts implicit. For instance, Guttag (1980) specifies a stack using algebraic axioms. A key to understanding that specification is recognizing that any time that the stack is empty is equivalent to

the time when it was first created. The axioms do not state this explicitly, though one could presumably add one or more axioms to do so. Rather, one must first hypothesize the fact and then prove it true from the axioms.

These observations lead us to our practical suggestions for writing more understandable formal specifications.

3. Practical Suggestions for Understandability

It is interesting to note that principles developed for structured programming apply equally well to nonprocedural specification languages of the axiomatic type. This means that the principles deal not so much with managing control flow as with managing detail. We state our suggestions, particularly emphasizing documentation.

1) Complex definitions should be broken up into short definitions which can be analyzed and reviewed in a top-down way.

2) Long, descriptive mnemonics, such as "the_line_numbered," are critical to understandability. They enable the user to make assumptions about predicates or functions defined at a lower level, assumptions that can be checked after understanding and checking the present level.

3) The specification language should provide a rich set of primitive objects and operations. This will alleviate the problem cited in section 2, the lack of previously defined concepts. For instance, suppose we

are specifying a module which among other things, sorts a sequence. The definition would be much shorter and clearer if a concept permutation(a,b) were already defined in the language or in a library. For that matter, the concept of a sorted sequence is so common, that its precise definition should be primitive to the specification language.

4) In any given language, since there are several styles of writing even simple formal specifications, alternative styles should be examined and selected on the basis of understandability of the individual item's specification. Sometimes a simple composition of the primitive elements of the language suffices to define a concept. Recursive definitions can provide short, easy-to-understand specifications. Another alternative is to use the English definition of the object as a pattern for a formal one; this often leads to definitions in terms of sets and operations on them. We have found that the style that seems clearest depends on the item being defined and that walkthroughs can be very valuable to check not only the correctness of a specification but also its understandability. For the three text editors we have specified, walkthroughs proved very valuable in criticizing both the formal definitions and the documentation. Understandability is not easily attained. For each of the text editors, after completely specifying one, we were able to conceive of a much more understandable, but functionally equivalent specification by

significantly changing the style of the definitions. We have concluded that entries for a library of formal module specifications should be prepared with so much care for understandability that casting away the first attempt at a specification to create a more understandable one is not frowned upon. Writing clear nonfiction receives that much attention; so should creating a specification to reside in a library of module specifications for repeated use.

5) A formal definition which is patterned after an English description of an entity is often easy to understand. Frequently there are different styles available for specifying a particular function of a module. These correspond to different views in defining it. The way one would describe it in English is a particular view which at times suggests a very understandable definition.

For instance, in section 4 we define a particular string pattern-matching feature of a module. The matched sequence is to be identified by its left and right boundaries in the string. If multiple matches could occur, the longest of the leftmost matches is to be selected. The first definition we wrote was a recursive one, which computes whether the match desired starts at the first position. If it does not, it uses the definition recursively on the remainder of the string after the first position.

However, that was very difficult to understand. A

second was developed modelling the phrase, "the longest of the leftmost matches." In it, the set of all matches is defined; then the leftmost matches are defined; then the longest is identified. The second definition is far easier to understand, and is presented in section 4. English descriptions of the expressions corresponding to each of the three parts mentioned above make the definition particularly convincing.

6) An English description of the purpose of a module and of each function available at the interface of the module provides a general notion or conceptualization for understanding the formal specification. Though that description will be vague, incomplete, and ambiguous, it conveys a toplevel view around which the complete, unambiguous, precise, formal description can crystallize into understanding in the reader's mind. Of course, the formal specification alone is the arbiter of all questions about the module. Such a high level description is essential documentation for management personnel.

7) The principle for deciding whether to include a comment for a line of the formal specification is whether its purpose and implications would be obvious to the average reader without a comment. (We are indebted to David Crocker for stating this criterion for us.) In general, formal specifications in Hoare's approach will consist of a large number of formulas, some of which can be very long. Each formula should be documented,

preferably with the documentation intermixed in the formulas, so that the purpose and implications of each subformula are made obvious. The reason is simple: the author of the specification when writing down a long formula had specific reasons for writing each subformula. Those reasons or implications should be succinctly stated.

Not only will following (6) and (7) make formal specifications more understandable, but also following them gives each reader of the specification the ability to verify that every aspect and subformula of it corresponds to the author's intent. This is an informal means of design validation; for a library of such specifications, the means is very powerful since more and more designers will be reading and verifying a specification as time goes by.

8) Certain English constructions can be very difficult to understand and should be avoided. The English specification of KSOS is generally well-written and well-organized. However a cryptic style by omitting words in defining variables or by overuse of parenthesized descriptions makes understanding difficult on certain points. An example exhibiting both features is the following definition (p. 27 of Ford Aerospace, 1978) of an exception condition of one of the functions: "unable to create new process (possible information channel)". It is unclear to us whether a new process cannot be created because of a possible information

channel or a new process which is a possible information channel cannot be created or some other explanation is intended. One can avoid these two features simply by stating everything as complete thoughts in complete sentences. For the example, this could be "A new process cannot be created because of a possible information channel".

9) All but very small specifications need an index. To pick an arbitrary figure, formal specifications whose complete hierarchical definition is at least six pages need an index. English ones at least twelve pages in length also need an index. The parser for the formal specification can easily create an index for all items referred to more than some threshold level of times.

10) If an English description accompanies the formal specification, as opposed to being embedded within the body of the formal one as comments, then a cross-reference between the two is needed. As indicated in section 1, one is likely to have an English specification in addition to the formal one for "large" module specifications. (Let us define "large" as above, at least 12 pages of English or 6 pages of formal.) The English one may be in the style of a proposed manual, DOD B-5 specifications, etc. The cross-reference will not only aid understandability by relating the two but will also provide an informal means for people to check that the formal specification does what its authors claim.

These practical suggestions are not new ideas. However, some computer scientists dispute our recommendations on documentation; they argue that the English descriptions play too central a role in specifications as advocated here. However, the gain in understandability of the specifications and the potential for informally checking their correctness far outweigh the potential for misinterpretation due to the documentation. Without the documentation, there is still the potential of misinterpreting a formal specification. For any significant specification, such documentation is crucial to understandability. KSOS (Ford Aerospace, 1978) has a 72 page formal specification that is almost devoid of comments. Though it has 62 pages of documentation according to principle (6), the lack of comments according to principle (7) prevent the formal specification from being very understandable. Neither an index nor a cross-reference is provided.

Our suggestions for writing understandable formal specifications are illustrated in the next section.

4. An Example

Rather than specifying completely a small example such as a stack, we present a portion of a rather complex definition which is one function at the interface of a text editor. Because a text editor is a realistically-sized example, only a small slice of it can be presented. Consequently, the function defined here refers to parts of the editor's specification not

present in the paper.

The function is the Find feature taken from Kernighan and Plauger (1976), chapter five. It is particularly appropriate for several reasons. One is that it is complex enough that the English description provided by Kernighan and Plauger, pages 136-138 of their text, is very hard to follow. This does not seem to be a result of expository style, but rather seems to indicate that intricate concepts are difficult to understand whether described in English or in a formal language. (They give no abstract formal definition; a concrete implementation is included in the text.)

A second reason for selecting this feature is the following paragraph (Kernighan and Plauger, 1976, page 149) where they are commenting on some of the intricacies of FIND.

> "We emphasize that these 'features' are ad hoc decisions made as we implemented the pattern builder. A number of curious situations turned out to be unspecified, as is often the case, and had to be resolved during coding. We chose to complete the specification in what appeared to be the most convenient way for the user."

The quotation points out exactly why one needs formal specifications in large software systems. In a small group of good programmers, working closely together, omissions of the type mentioned in the paragraph can be readily corrected and communicated to all. However, on large software systems, such omissions can lead to design errors which are quite costly to fix.

In section 4.1, we present our English description complementing the formal specification of Find, according to our recommendation (6) in section 3. Identifiers of significance in the formal definition are enclosed in single quotes. In section 4.2, the top-level definition of Find is given. Section 4.3 discusses the example.

4.1 Our Description of Find

'Find' searches a range of lines from 'starting_line' to 'stopping_line' in the file for a match of 'pattern'. 'Find' gives as a value the lines having a match. Its definition has been designed so that 'one_line_search' may be used in string replacement; 'one_line_search' returns a triple indicating the line matched, the left boundary of the match, and the right boundary. If there are several matches on the line in question, the boundaries for the longest of the leftmost matching strings are given by `one-line-search'.

A 'pattern' that a user of the module passes to 'find' is a string of symbols consisting of a sequence of the following logical elements:

a) a literal character - The literal sequence of characters STRUCTURE matches strings identical to it.

b) a character class - Any character in a character class can appear next in the matched string. They are enclosed in square brackets.

1) A list of alternatives may be

given. [DS] matches either a D or an S.

2) A range of letters or digits may be given, by a shorthand notation using a hyphen. [Ø-9] matches any digit.

3) A question mark matches any single character.

4) The complement of the desired class can be indicated by placing ¬ as the first character within brackets. [¬A] matches anything except an A.

c) The Kleene star may be applied to any literal character or any character class, allowing zero or more repetitions of the literal or class. Algol identifiers may be defined then by the pattern [A-Z][Ø-9A-Z]*.

d) The special nature of any character can be overridden by preceding it with an at-sign. For instance, @* matches a single asterisk.

e) If matches are required to begin at the start of the line, the pattern must start with a percent sign.

f) If matches must end at the right boundary of lines, the pattern must end with a dollar sign.

g) All special characters have a "normal position." If they are used in other than their normal position, they are treated as literals. For instance, *$% can only match an identical string of three characters.

4.2 A Formal Specification

In this section, we describe the top-level definition of Find, given in figure 4. It is written in SPECIAL (Roubine and Robinson, 1976), which falls into

the class of Hoare's axiomatic approach. In SPECIAL, comments are preceded by a dollar sign. (I have used parentheses to delimit the comments following the dollar sign.) Words in upper case are reserved words of the language.

In a module definition in SPECIAL, one of the most important parts is the list of function definitions at the interface of the module. Functions may compute a value and/or have side-effects. The side-effects are unordered, and may be viewed as occurring simultaneously. Any of the functions may have exception conditions under which they should not be called. These are checked in the order listed. If an exception occurs, control is returned to the caller without computing a value and without any side-effects occurring; however, a message indicating which exception occurred is sent to the caller.

In figure 4, the top-level definition of Find appears. The remaining paragraphs are presented since the syntax and semantics of SPECIAL are probably not familiar to the reader. These would not be part of the documentation. Lines 1 and 2 define find as a function of three arguments: an INTEGER called starting_line, an INTEGER called stopping_line, and a pattern whose data type is a user_pattern (defined elsewhere). The value returned is a vector or sequence of lines; the vector's name is vec. Following line 2 is a comment which succinctly indicates the purpose of the function.

```
 1 VFUN find(INTEGER starting_line; INTEGER stopping_line;
 2           user_pattern pattern) -> VECTOR OF lines vec;
   $ (Find searches for all lines from starting_line to
      stopping_line, including both starting_line and
      stopping_line, matching pattern and returns those
      lines as a value.)

 3 EXCEPTIONS
 4   NOT valid_line(starting_line);
 5   NOT valid_line(stopping_line);
 6   starting_line > stopping_line;
 7   ill_formed(pattern);

 8 DERIVATION
 9   LET SET_OF INTEGER
10         matching_lines = {INTEGER i |
11                             i >= starting line AND
12                             i <= stopping_line AND
13                             one_line_search(i, pattern,
14                                  internal_form_of(pattern))
15                               ~= no_match}

                                  $ (One_line_search has the
                                     value no_match only when
                                     i contains no match of
                                     pattern.)

                $ (Matching_lines is the set of all numbers of
                   lines in the specified range, which match
                   pattern.)

16       IN LET VECTOR OF INTEGER v |
17              LENGTH(v) = CARDINALITY(matching_lines) AND
18              v[j] INSET matching lines AND

                  $ (The vector consists of line numbers of
                     matching lines.)

19              (FORALL INTEGER i, j |
20                      i > 0 AND
21                      j > 0 AND
22                      j <= CARDINALITY(matching_lines) :
23                           (i < j => v[i] < v[j]))

                           $ (No line numbers are
                              repeated in the sorted
                              vector v of matching lines.)

24          IN VECTOR(FOR i FROM 1 TO CARDINALITY(matching_lines) |
25                    the_line_numbered(v[i]))

              $ (The lines themselves are returned in sequence,
                 not the line numbers.)
```

Figure 4

```
1  BOOLEAN valid_line(INTEGER line_number) IS
2          line_number > 0 AND
3          line_number <= number_of_lines;
```

Figure 5

Lines 3-7 list the exception conditions on using find. Lines 4 and 5 require that neither starting_line nor stopping_line is an invalid line. In reading lines 4 and 5, it is easy to see that these exception conditions are relevant, because of the clear mnemonics. The definition of valid_line given in figure 5 states that valid_line is either true or false and that it has an INTEGER argument called line_number. It is true if line_number is greater than zero and does not exceed the total number of lines. Since the formal specification of the file defines line numbers as ranging from one to number_of_lines in increments of one, the definition of valid_line correctly distinguishes between valid and invalid line numbers.

Line 6 of the specification of find requires that starting_line not exceed stopping_line.

The use of mnemonics in line 7 makes this exception condition easy to understand and accept. The value of the mnemonics in a top-down reading of the definition is that one can make certain assumptions about well-named subdefinitions such as valid_line and ill_formed, write them down, and check whether the top-level definition of find is correct given those assumptions. After ascertaining that, one may verify whether the subdefinitions

satisfy those assumptions.

The value returned by find is derived from other VFUN functions defined at the interface of the text editor. Lines 8 to 25 specify the value returned using two local definitions. These are given by a construct "LET definition IN expression," which signifies that the definition is in effect only during the expression.

The first definition, lines 9-15, specifies a set matching_lines which is a set of integers i having 3 properties. The first two, lines 11 and 12, indicate that the lines matched must be in the specified range. The third, given in lines 13-15, is a rather complex subformula which has special meaning to the author of the definition. Consequently, the comment immediately following line 15 was introduced. Not only does it aid understanding, but also it indicates the significance of the formula in lines 13-15. Assuming that the comment is true, it is easy to see that the three properties together do specify the set of line numbers of lines with a match. The assumption that the comment is true can be checked in detail by examining the definitions of one_line_search and internal_form_of.

The second definition appears in lines 16-23. It defines a vector of integers v. Notice from line 17 that there are exactly the number of components in v as there are in the set matching lines. Line 18 implies that the elements of v are all from the set of matching line numbers, matching_lines. The formula in lines

19-23 is rather complex; however, the comment following it reveals its intent. From that comment, one would want to verify that the property implies two things about vector v: 1) that it is a sorted vector and 2) that there are no repetitions in the vector. Thus, the comment has made the specification more understandable and presented us with two properties to verify the correctness of the formula. (Note that "=>" signifies material implication.)

Given the definitions of matching_lines and v, the value of find is specified in lines 24 and 25. It is a vector which starts at 1 and whose length is just the cardinality of matching_lines. The ith component of the vector is given by the line numbered by the ith component of v. Using "the_line_numbered" as a mnemonic suggests that it refers to a line indicated by its argument, which is a line number. This assumption enables one to conclude that find does return all matching lines in the specified range, thus, completing analysis of the top-level definition. Of course, one must verify this assumption, along with all the others, by checking the subdefinitions, such as ill_formed, one_line_search, internal_form_of, and the_line_numbered. (Those definitions are not included in this paper.) One may proceed one definition at a time, providing a way of breaking the vast amount of detail into managable units.

4.3 Discussion of the Example

Upon comparing the English specification with the top-level definition, one might wonder whether they correspond. For instance, where is the effect of the at-sign, or the semantics of character classes, or the role of the "normal position" of the special characters? These concepts are all presented precisely in the subdefinitions of ill_formed, one_line_search, and internal_form_of. All that detail was factored out of the top-level definition of find. This enabled us to effectively understand the formal specification in a top-down way. (Furthermore, the top-level definition is independent of the syntax and semantics of the patterns.)

The principle of top-down analysis is thus applicable to making formal specifications understandable in an abstract, nonprocedural, axiomatic language, such as SPECIAL. The multi-word mnemonics of the example were essentially phrases suggesting the author's intent for the named quantities; assumptions one makes at the top-level must be noted for checking at lower levels in the definition. A third contribution to the understandability of find was the use of comments about particularly complex subformulas to indicate their purpose. Not only did these aid understandability, but also they provided a way to check the correctness of the subformulas in achieving their purpose in the specification.

In the example, it is obvious that both the English description (section 3) and the formal specification are

rather difficult to grasp. The cause is quite likely that the notion of find, being so full of intricacies, is itself hard to understand irrespective of the descriptive language used. Merely using English as a specification language does not guarantee understandability, no matter how well written and edited.

5. Suggestions for Writing Correct Formal Specifications

A program is correct if it fulfills its specification. One checks program correctness by comparing what the program does against what the specification says it should do. What then does one mean by correctness of a formal specification of a module? By what standard is it deemed correct? Liskov and Zilles (1975) views the specification process as translating from the concept in someone's mind of what a program should do to a formal specification. In general, it must conform to our intent for the module. In particular, it should be internally consistent, should leave no cases unspecified, and should prevent implementations from satisfying the specification without carrying out our intent. Guttag and Horning (1978) presents formal definitions of consistency and completeness. Though their definitions are given for languages based on algebraic axioms, it is easy to define similar notions for the other two types of specification languages. Gerhart and Yelowitz (1976) presents two examples of formal specifications which left out a crucial feature; therefore, programs could be

written that fulfilled those specifications but which did not satisfy the person's intent.

One could define correctness of specifications in terms of writing two different specifications and proving them equivalent. We feel however that this misses the point of the difficulty of writing specifications. Rather, we prefer an informal notion of correctness of a specification. To be correct, it must conform to the intent for the module, it must be internally consistent, it must leave no cases unspecified, and it should rule out implementations that do not conform to the intent for the module.

We recommend three techniques to aid the writing of correct specifications. Each has uncovered errors for us during the specification process.

1) Use software tools to perform as many checks as possible. Development of such tools is an active area of research, and several examples exist. The specification handler for SPECIAL (Roubine and Robinson, 1976) verifies syntactic correctness and performs type checking on all expressions (every expression has a type in this strongly typed language). Guttag, et.al. (1978) describes a tool for AFFIRM, which is one of the languages based on algebraic axioms; the tool enables one to automatically simulate the module specified and to prove properties of the specification. Of course, one of the advantages of the third class of languages, the operational ones, is that one can execute the

specification itself.

2) Walkthroughs uncover many errors, as well as offering valuable comments for improving understandability. If one creates documentation according to our guidelines in section 3, the documentation will provide much detailed insight regarding the intent of the specification, and therefore will provide a basis for judging correctness.

3) If one is not using an operational specification language, and if the software tools available cannot simulate the module specified, then a quick implementation in a very high level language will provide a concrete system for testing the functional capabilities of the module in question. In general, we wrote implementations in INTERLISP (Teitelman, 1975) for our SPECIAL specifications. This often uncovered specification errors not detected by walkthroughs or the SPECIAL specification handler (Roubine and Robinson, 1976). The implementations, since they were in such a very high level language, took remarkably little time, for using the very high level language enabled us to trade performance characteristics of the test module for programmer time. For instance, the final module checked in this way took only two to three days for one programmer to implement, even though it was a kernel providing the functional capabilities of a character-oriented text editor. Furthermore, we found significant regularity in the implementation of most types of expressions in

SPECIAL, suggesting that much of each implementation could be done automatically by a software tool.

6. Prospects for a Library of Formal Specifications

The kind of library considered here would be a reference collection of module specifications for designers to browse through. For any given need, such as a file management module, the library would contain a number of alternative specifications ranging through the spectrum of functional capabilities a file management module might offer. Naturally, reusing specifications rather than creating them could vastly reduce the cost of formal specifications.

There are four issues to consider in the feasibility of such a library. They are discussed in sections 6.1-6.4. Our conclusions regarding a library appear in section 6.5.

6.1 Modifiability of a Specification

Given that the most appropriate specification is found, it may not be a perfect match to the designer's needs. In that case, the more easily the specification can be modified to suit those needs exactly, the better. There are two simple techniques which will make specifications rather modifiable.

1) The first is to structure the definitions of each function at the interface of a module so that each particular detail that might need modification is localized in only one subdefinition. Then only one

subdefinition need be changed rather than the changes being spread throughout the specification. (This is just the notion of abstraction.) The specification of the Find operation in section 4 illustrates this. Both the syntax and the semantics of the pattern language for the search could easily be modified for different user environments because both were provided in subdefinitions.

2) A second idea is that the author of the specification should list the decisions that are arbitrary and depend on the environment in which the module will be used. For example, in a sequential programming environment, an attempt to remove an item from an empty queue is an error. However, in a multiprocessing environment, such a condition is merely a signal that the process requesting information should be suspended until some other process adds something to the queue.

These two simple principles will make the prototypes in the library rather modifiable.

6.2 Number of Prototypes

For any particular type of module needed, the ideal would be that a handful of prototypes would cover the major possibilities for a given need, so that the designer can quickly ascertain which, if any, fits best. If there are many prototypes necessary for each application, then the time spent analyzing each one will make using the library prohibitive.

The evidence we have thus far is very encouraging. In our study of data structures and text editors, a handful of prototypes seem adequate. Even if the instances below are off by a factor of two in the number of variations needed, the fact that only a handful suffice means that the number of prototypes to be examined is not prohibitive. However, we have insufficient evidence to extrapolate beyond the domain of data structures and medium-sized software tools such as text editors. As the size of the module grows, it is not clear that only a handful will be sufficient.

For a stack, two versions seem necessary: one from which one can read only the last item stored, and one permitting any value designated by a movable pointer to be read, but not modified. For a queue, we suggest four variations: one where reading occurs at only the front, a priority queue, and two character streams. In a priority queue, the first entered of the largest values (highest priority) is read or removed from the sequence before any others; no other values can be read. Character streams enable the details of synchronizing input/output operations to be hidden in the module rather than forcing all programs to be aware of the means of synchronization. One of the character streams is character oriented; the other is oriented to blocks of characters or lines.

For trees, we suggest two variations: the binary tree and general tree with arbitrarily many branches.

We do not consider a threaded tree a third logical variation, since it is an implementation of a fast means of performing tree traversal, an operation specified at the interface of the module. Therefore, though implementations using thread links affect performance, they do not change the functional capability.

For text editors, we found three logical alternatives. One has operations oriented to adding, deleting, or moving characters. Another has operations oriented to lines and line numbers. The third example is an editor whose operations are oriented to moving a cursor on a CRT screen and editing via changing the screen. The specification for each text editor ranged from 9 to 28 pages including copious comments based on the suggestions of section 3.

Three principles which emerged from specifications we wrote should cut down on the number of alternatives necessary for any given application, if the principles are followed in writing a new entry for the library.

1) Select a consistent set of decisions for those details that depend on the module's use in practice and make those decisions easy to modify using the two suggestions in section 6.1.

2) Avoid issues that are not fundamental to the logical, functional capabilities of the module. Those differences would multiply the number of entries for a given application in the library without adding any new abilities. For instance, in specifying text editors, we

did not define a user command language, for there are many legitimate syntactic variations, each of which will be of varying value to different user communities.

3) Specify many fundamental, primitive operations to give each prototype a maximal number of basic features. A designer, after selecting a specification from the library, can delete operations from the interface not needed in his/her environment, assuming that each operation was defined using information-hiding as stated in principle (1). For instance, our specification of a line-oriented editor was patterned after the functional capabilities of SOS (National Institute of Health, 1977). Operations corresponding to the alter mode, where the user can modify a range of lines using character-oriented operations relative to a pointer, can easily be removed.

In conclusion, our experience is too limited to speculate whether for all applications, only a handful of prototype specifications will cover almost all variations in modules for the particular application. However, the three principles made it possible for only a handful of prototypes to cover classes of data structures and to cover text editors. That fact is quite encouraging.

6.3 Contents of a Library Entry

First, we consider the type of documentation for any given formal specification in the library; then we consider whether implementations can be stored as well.

The documentation with the entry is critical, not only for understandability, but also for the designer to be able to quickly eliminate entries that are not close to his/her need. Then, the designer can focus attention on two or three that are most promising. Otherwise, the library would bog designers down on issues that should be resolved quickly.

In addition to comments within the specifications, which we discussed in sections 3 and 4, five types of information seem valuable for quickly deciding on the relevance of a module specification.

1) One is a description of the purpose of the module and the kinds of needs it fills.

2) A second is a summary description of each class of functional capabilities the system has. Notice that this is the kind of information not included in the formal specification, as pointed out in section 2.

3) Third, the kinds of decisions not made by the module should be described; these are implementation decisions which would have to be made after selecting a module specification, when the coding phase begins. This will reinforce the fact that those issues cannot be assumed. For instance, in specifying a text editor, we would write in this section, that a decision to use array storage, linked lists, or other alternatives to store the file being edited, would have to be made when programming of this module begins.

4) Specific references, such as texts and journals,

if available, should be given describing various implementations, algorithms, and analyses of them for use when programming begins.

5) As discussed in section 6.1, the author of the specification must include all ways foreseeable that the specification might require modification to tailor it to specific needs.

In addition to the documentation identified above, could implementations be stored as well? Even for a module which has a very simple specification, there may be many different implementations. For instance, Horowitz and Sahni (1976) presents a specification of a symbol table; yet, that text spends the majority of two chapters describing and analyzing alternatives for implementation, such as linear search, binary search, fibonacci search, various hashing techniques, tree indexing, etc. Even a single operation can have many competing algorithms; consider sorting with quicksort, heapsort, bubblesort, and insertion sort as alternative implementations.

As one considers larger and larger modules, the number of possibilities could grow dramatically. For the kernel of an operating system, one would have various hierarchical decompositions of the kernel (as a module) into many smaller modules. For each of the smaller modules of each of the decompositions, there would be alternatives in implementation.

Some reprogramming may be necessary, since the

module specification may oftentimes need slight modification due to varying environments in which the module is to be used. This will require each program stored to be very well structured and very well documented so that it may be easily modified.

It is technically feasible to include with the functional specification of the module's interface various hierarchical decompositions and corresponding programs implementing the module. Cooprider (1979) presents a technique for defining and maintaining a family of software systems. His Software Construction Facility (SCF) provides a module interconnection language which defines module interconnections of entire systems, the shared aspects (and differences) among versions of those systems, and the sequence of operations for assembling versions of modules into a complete system. Parts of the interpreter for SCF have been implemented. This technique might be adapted to enable a family of implementations to be stored in the library even for large systems.

However, since the number of implementations that may need to be stored for any module could be relatively large, developing a reasonably complete library including implementations as well could take many years. However, the library would be of great value to designers just with the module interface specifications as the alternative programs for each module are added slowly.

6.4 Retrieval

Clearly, facilitating retrieval from the library is a crucial issue for the library to succeed. It is not clear what assumptions one should make regarding the designer. Should one assume the need will be clearcut, such as needing a specification for a symbol table that is suitable for a block-structured language? Or is it more likely the case that the designer will have only a vague notion of the need and will to browse among many wide-ranging classes of modules? When it is clear what assumptions can be made about the designer, one can determine whether existing retrieval techniques can be used or modified.

6.5 Preliminary Conclusion

A library of formal specifications of modules appears quite promising from the analysis presented here. However, we have pointed out some open questions that should be resolved. The most notable of these is the issue of retrieval. The second is the need for an appropriate module interconnection language and its interpreter to facilitate storing alternative implementations for large systems.

Five causes of difficulty in understanding formal specifications were listed in section 2. Of these, a library based on our practical suggestions address three. The comments we have advocated within the specification and the descriptions with a library entry will provide the summary information normally missing from formal specifications and will state explicitly

facts normally implicit in the formal specification. The library itself will provide a large body of defined concepts, as in natural language. On the issue of the semantics of the specification language being quite different than programmers are used to, the concept of a library is neutral, since no assumption is made regarding the semantics of the language. The last of the five causes it treats is unavoidable; in this view the vast amount of detail is considered a major advantage of formal specification.

A library of formal specifications deals with the high cost of creating formal specifications by drawing on a very large pool of them rather than writing new ones.

7. Possibilities for Semi-formal Specification Techniques

In the previous section we described a library of formal specifications as an approach to reducing the effort and high cost of creating formal specifications for a large software system. In this section we examine the prospects of another approach, which we dub semi-formal specifications. The goal is to combine as much of the advantages (rigor, precision, detail, etc.) of formal specifications with the ease of use of English specifications.

One possibility is to change the semantics of the specification language and/or to specify only to a certain level of detail. We consider this possibility in

section 7.1.

Another possibility is to let a person specify a module in a language closer to English, particularly in its semantics, and to use a software tool to help fill in the detail and precision of a formal language. This is considered in section 7.2. Clearly, the two possibilities are not mutually exclusive.

7.1 Almost Formal Specifications

To deal with the problem of the semantics of present formal specification languages being quite different than programmers are used to one could try to create a new language. Balzer and Goldman (1979) suggests one.

To deal with the problem of the amount of detail in formal specifications (compared to English ones), one could allow informal portions in a formal specification, in order to deal with detail which it is unprofitable to formally specify. This certainly opens Pandora's box unless one uses informality only for features whose formal specification would not warrant its cost. For instance, consider current mail systems on the ARPA net. Given the way they treat acknowledgement of receipt of a message sent by the user, it would probably not pay to specify formally the exact conditions under which an acknowledgement is received by the user for a message he/she just sent. It is sufficient to have an input that is changed, depending on whether acknowledgement is received; the conditions under which the input is

changed could be explained informally. (On the other hand, for future mail systems, one might wish to formally specify those conditions to define aspects of the legal significance of acknowledging delivery (Crocker, 1980).)

To some degree, allowing some level of detail unspecified is probably a necessity in truly large, complex modules. For instance, in KSOS (Ford Aerospace, 1978) there are several examples of this; two of them follow.

1) The lowest level is a module mac with a comment as its heading which states "CONTENTS: Machine". Its only callable function MACclock takes no arguments and returns an integer value called "time". The initial value is the special value UNDEFINED. There is no specification of how the value is changed, what it becomes, etc. Obviously, a formal specification of the machine time was not important to the purposes of the designer's.

2) Certain functions are independent of the device, and are defined in a module dif. One is DIFcreateDevice which takes one argument, devInit. The effects of calling this function are given in two formal statements and a comment, "other effects as dictated by contents of devInit". It is clear that the comment does indicate that there are other effects which are not formally specified, since the two formal statements of effects do not depend on the argument devInit at all.

Another example of formal specifications being selectively imprecise is in dealing with arithmetic. Formal specifications often include arithmetic on the real numbers, without concern for whether it will be implemented with finite, machine arithmetic. In fact, correctness proofs often do not concern themselves with the limits of a particular machine arithmetic.

Consequently, even in formal specifications, some informality has been used. We suggest that judicious use of informality could significantly help in writing understandable formal specifications.

Heninger (1980) overviews a system specification using descriptive techniques which may prove very natural to programmers. Though not a module specification, their requirements specification of the flight software of the A-7 aircraft offers techniques that might be adapted to module specification. One of their principles has been, "Be as formal as possible" (Heninger, 1980, p. 4). It makes significant use of tables as a precise definition technique; it also uses informality judiciously.

We feel a "semi-formal" approach is immediately practicable and that it will dovetail well with application needs for which there may be no suitable entry in a library of formal specifications. It addresses two problems: the amount of detail necessary in a formal specification and the fact that the semantics of present specification languages is quite different from what

programmers are used to. It does not directly address the other three problems raised in section 2. The impact on difficulty in creating specifications is that detail that is not worth formally specifying could be described informally; the amount of impact this would have is probably far less than having a library of formal specifications.

7.2 Sophisticated Tools to Aid in Creating Formal Specifications

Since formal specifications are so difficult to write, software tools for this purpose would be quite valuable. Limited aids for this purpose would be (1) a good file maintenance system which would keep track of various versions, updates, etc., clearly marking all changes with their date for each item modified and (2) an automatic index listing pages where all variables and functions are used, locations of their definitions, etc. Both are within the state of the art; for instance, INTERLISP (Teitelman, 1975) has some basic facilities along these lines.

More sophisticated tools, not within the state of the art, would aid in the following tasks: (1) detecting ambiguity and vagueness in English specifications or (2) filling in much of the detail of a formal specification from a more general description (perhaps in English, perhaps not). We have recently been conducting preliminary studies of whether such sophisticated tools will be technologically feasible in the foreseeable

future. Our method has been to study portions of the English specification and formal one for KSOS. One purpose is to compare and contrast the informal and formal. The second has been to determine the difficulties involved in semi-automatically generating the formal specification from the English one; Balzer, et al. (1979) also studies this. While we do not believe that it will be possible for a machine under human direction to generate a formal specification from an English description so rich, complex, and long as that for KSOS (62 pages), we expect that our study will suggest what sophisticated tools will be possible.

Two developments support such a study. First is the remarkable progress in the last decade in natural language processing within artificial intelligence; English interfaces to data bases have even become commercially available, e.g. the ROBOT system of Artificial Intelligence Corporation (Harris, 1977). Second is the expectation that by the end of this decade VLSI technology will make the hardware for extremely sophisticated personal computing available very inexpensively.

The critical issues regarding tools for processing English specifications in some way are threefold.

1) The nature of software specification may offer simplifying heuristics for understanding English. For instance, the success of research in English query of data bases is largely due to the fact that the semantics of meaningful input is constrained by the data base

schema. Consequently, simplifying heuristics for processing English are successful in the data base domain, though they would not work in more open-ended domains such as technical reports and novels. A person's language seems to be naturally constrained by the interface. Therefore, the possibility of sophisticated tools with English input depends on the amount of semantic constraint that the domain of software specification affords.

2) There may be many tedious details that a sophisticated software tool can fill in easily, freeing the designer for decisions which people are more adept at than juggling a plethora of detail. Of course, this has been the trend in computer languages since the inception of computer software. One way is to automatically fill in detail for a higher level concept a designer chooses, whether the designer's choice is given in a formal language or a natural one.

3) A sophisticated tool could suggest places in an English specification that may be ambiguous or vague and the ways that they are ambiguous or vague. A weakness of natural language understanding systems is in fact finding ambiguity even where there is none. An English understanding system would therefore detect the genuine cases of ambiguity, but would flood the designer with messages regarding items that simply are not ambiguous. The issue is whether heuristics can be developed for the limited domain of English specification which would

limit the false cases of ambiguity to a reasonable size for the designer to handle. In addition to our study, Mander and Presland (1979) examines this.

Such sophisticated tools would reduce the difficulty of creating formal specifications in two of the areas cited in section 2. First, they might relieve the designer of creating much of the detail in formal specification by the machine supplying it. Second, they might enable the designer to express his/her expressions in English, thereby using the many predefined concepts familiar to the designer, but which are not yet defined in formal specifications. The other three problems mentioned in section 2 are not directly addressed in the goals outlined for these tools.

Clearly, such tools are a long-term research goal.

8. Conclusions

We have illustrated that many of the ideas of structured programming carry over to writing understandable module specifications. This may seem quite surprising, since two of the three classes of specification languages are nonprocedural and have virtually no sense of flow of control or sequencing and since structured programming involves disciplined use of flow of control for clear programs. This demonstrates that the notions that are applicable to nonprocedural languages deal more with managing details of formal definitions and reasoning about them than with coding flow of control.

Our preliminary analysis shows that a library of formal specifications of modules appears quite feasible and extremely promising. Small-scale experiments in collecting and using such a library are called for while formal specification languages continue to be enhanced. Probably the most appropriate first step is publishing formal specifications of software modules and collecting them in a way analogous to the Collected Algorithms of the ACM. (I am indebted to Leon S. Levy for this suggestion.) This would facilitate research on formal specifications and would make various styles of writing them more widely known.

We feel that the most promising approach to reducing the massive effort in creating a formal specification for a complex system is such a library. A complementary approach for items not in the library is writing a semi-formal or almost formal specification. Here one would be as formal as possible, only filling in detail informally when formal specification of such detail would not warrant the effort. A mixture of current specification languages, tables, and a little bit of English could prove quite effective in precisely defining module interfaces with less effort. Clearly, it is a difficult trade-off between rigor and precision on the one hand and ease of specification on the other.

The third approach we recommend is a research goal: sophisticated software tools to aid the designer in writing formal specifications. The payoff here would be

highest, though the scientific problems in creating such tools are large.

Acknowledgements

David Crocker, Peter Freeman, Leon S. Levy, Jim Neighbors, and Linda Salsburg made many valuable suggestions in wading through early drafts of this. Linda Salsburg wrote many of the specifications and quick implementations that influenced the suggestions here.

Bibliography

Ambler, Allen L., Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, 12, No. 3, March, 1977.

Balzer, Robert, "AP2 Reference Manual", Information Sciences Institute, Marina del Rey, CA, (in preparation).

Balzer, Robert and Neil Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," Conference on Specifications of Reliable Software, Cambridge, MA, 1979.

Cooprider, Lee W., "The Representation of Families of Software Systems", Ph.D. Dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.

Crocker, David, personal communication, 1980.

Ford Aerospace, "Secure Minicomputer Operating System (KSOS): Computer Program Development Specifications (Type B-5)," Technical Report No. WDL-TR7932, Ford Aerospace & Communications Corporation, Palo Alto, CA, 1978.

Gerhart, S. L. and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, 1976, 195-207.

Guttag, J., "Notes on Type Abstraction (Version 2)", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980, 13-23.

Guttag, John V., Ellis Horowitz, and David R. Musser, "Abstract Data Types and Software Validation," CACM, 21, No. 12, 1048-1063, 1978.

Guttag, J. V. and J. J. Horning, "The Algebraic Specification of Abstract Data Types", Acta Informatica, Vol. 10, 1978, 27-52.

Harris, L. R., "User Oriented Data Base Query with the ROBOT Natural Language Query System", International Journal of Man-Machine Studies, Vol. 9, 1977, 697-713.

Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980, 2-13.

Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", CACM, Vol. 12, 1969, 576-580.

Horowitz, Ellis and Sartaj Sahni, Fundamentals of Data Structures, Woodland Hills, CA: Computer Science Press, Inc., 1976.

Kernighan, B. W. and P. J. Plauger, Software Tools, Reading, MA: Addison-Wesley Publishing Co., 1976.

Levitt, Karl N., Lawrence Robinson, and Brad A. Silverberg, "A Basis for Simulating Modules Written in SPECIAL", Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1979.

Liskov, Barbara H. and Valdis Berzins, "An Appraisal of Program Specifications," Proceedings of the Conference on Research Directions in Software Technology, Peter Wegner, Jack Dennis, Michael Hammer, and Daniel Teichroew (eds.), 1977.

Liskov, B. H. and S. N. Zilles, "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, 1975, 7-18.

London, R. L., J. V. Guttag, J. J. Horning, B.W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof Rules for the Programming Language EUCLID", Acta Informatica, Vol. 10, 1978, 1-26.

Mander, K. C. and S. G. Presland, "An Introduction to Specification Anaysis - SPAN", Technical Report CSS/79/12, Dept. of Computational and Statistical Science, The University of Liverpool, 1979.

Musser, David R., "Abstract Data Type Specification in the Affirm System," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980, 24-31.

National Institutes of Health, "SOS (An Advanced Line-Oriented Text Editor) User's Guide," Comp. Center Branch, Div. of Computer Research & Technology, National Institutes of Health, Bethesda, MD, 1977.

Neumann, Peter G., Robert S. Boyer, Richard T. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System: The System, Its Applications, and Proofs," SRI Project 4332, Final Report, Stanford Research Institute, Menlo Park, CA, February, 1977.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, Vol. 15, No. 12, December, 1972, 1053-1058.

Parnas, D. L., "On the Design and Development of Program Families," IEEE Transactions on Software Engineering,

Vol. SE-2, No. 1, March, 1976, 1-8.

Parnas, David L., "The Use of Precise Specifications in the Development of Software," Information Processing 77, B. Gilchrist, (ed.), North-Holland Publishing Company, New York, 1977.

Robinson, Lawrence and Karl N. Levitt, "Proof Techniques for Hierarchically Structured Programs", CACM, Vol. 20, No. 4, 1977, 271-283.

Robinson, Lawrence, Karl N. Levitt, Peter G. Neumann, and Ashok R. Saxena, "A Formal Methodology for the Design of Operating System Software," Current Trends in Programming Methodology, Volume 1, Software Specification and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

Roubine, Olivier and Lawrence Robinson, SPECIAL Reference Manual, Technical Report CSG-45, Stanford Research Institute, Menlo Park, CA, August, 1976.

Teitelman, Warren, "Interlisp Reference Manual," Xerox Palo Alto Research Center, Palo Alto, CA, 1975.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-80-1026	2. GOVT ACCESSION NO.: AD-A090 551
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): PRACTICAL SUGGESTIONS FOR WRITING UNDERSTANDABLE, CORRECT FORMAL SPECIFICATIONS	5. TYPE OF REPORT & PERIOD COVERED: Interim
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Ralph M. Weischedel	8. CONTRACT OR GRANT NUMBER(s): F49620-79-C-0131
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Delaware, Department of Computer and Information Sciences, Newark, DE 19711	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: August, 1980 13. NUMBER OF PAGES: 58
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

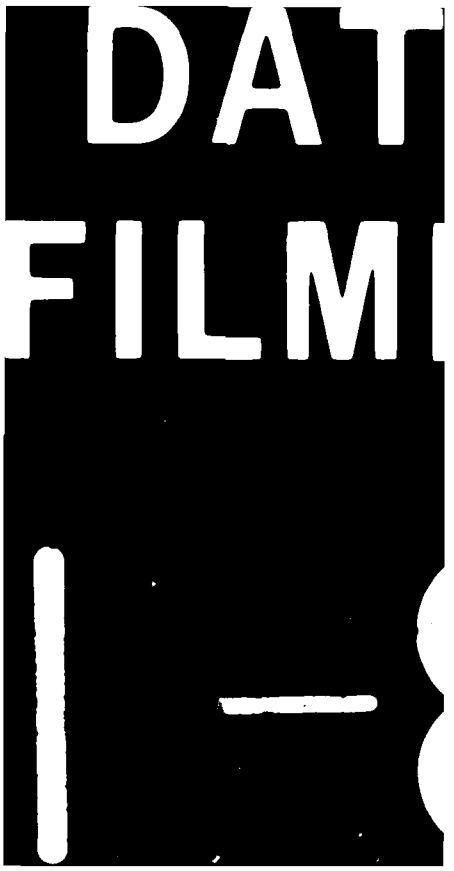
formal specifications, SPECIAL, KSOS, module specification, software design, library of specifications, English specifications

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The first half of this report is tutorial. It describes the three major classes of formal specification languages and argues that understandability is critical for formal specifications. Reasons why they are difficult to understand are identified, and practical suggestions for making them more understandable follow from that. The suggestions are illustrated by the specification of a pattern-matching facility. Three practical suggestions for checking the correctness of formal specifications follow.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED


DAT
FILM