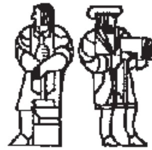
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

An Appraisal of Program Specifications

Computation Structures Group Memo 141-1
April 1977

Barbara H. Liskov
Valdis Berzins

This research was supported by the National Science Foundation under grant DC74-21892.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

1. Introduction

Every program performs some task correctly. What is of interest to computer scientists is whether a program performs its intended task. To determine this, a precise and independent description of the desired program behavior is needed. Such a description is called a *program specification.*

Several kinds of program specifications may be usefully distinguished. In constructing software, the first phase consists of an analysis of the requirements for the software. The result of this phase is a *requirements specification.* A requirements specification is an informal description of intended system behavior; it is often very large, running into hundreds of pages. The requirements specification details both the tasks that the system should perform, and some constraints on the speed and resource utilization with which these tasks should be accomplished.

After the requirements analysis phase, there are one or more design phases, which define a system structure meeting the requirements specification. Large programs are not constructed as single monolithic entities, but as a number of interconnected subprograms or modules. Most often these modules are organized in a hierarchical fashion, with modules higher in the hierarchy implemented in terms of (by means of calls on) modules lower in the hierarchy. The result of a design phase is the identification of modules, and a graph structure showing which modules are to be used in implementing which other modules. Later design phases are often used to elaborate the structure of modules that were considered only as indivisible "black boxes" in earlier design phases.

A common problem in system construction is that the kind of behavior expected by a user of a module may not be the same as what the module provides. This problem can be avoided, or at least greatly reduced, if the design phase provides specifications of the module. The module specification serves to document the intended module behavior and to communicate this behavior

to the implementor of the module and to programmers who will use that module in implementing other modules. Two kinds of module specifications are of interest: *functional specifications*, which describe the effect of the module on its external environment, and *performance specifications*, which describe constraints on the speed and resource utilization of the module.

In this paper, we survey existing techniques for providing formal functional specifications for program modules. We believe these techniques can form a basis for the functional part of requirements specification; this is discussed further in Section 4. The techniques we discuss are not applicable to performance specifications. Some work has been done in this area (see [51] for some recent work), but much remains to be done.

In the remainder of this section we discuss the advantages of formal specifications. Section 2 contains a discussion of module specification techniques for sequential programs. In Section 3, we review recent work on specifying parallel programs. Section 4 evaluates formal specification techniques and discusses expected future developments.

1.1 Advantages of Formal Specifications

Program specifications can have various degrees of formality. At the informal end of the spectrum, the specifications can be expressed in some convenient combination of English, diagrams, and a variety of standard mathematical notations. Sometimes specifications are required in a prescribed format, where the order of the sections and the information to be found in each section are given, but the contents of the sections can be in any language. These kinds of specifications are in common use today.

The ISDOS system [50] is a step up from this. The interface specifications for the modules of the proposed software system are expressed in a precisely defined formalism, which is parsed by machine, and checked for various completeness and consistency properties. The

specifications for the functional behavior of the modules, however, are informal, unconstrained, and unchecked.

A specification is formal if it is written entirely in a language with an explicitly and precisely defined syntax and semantics. Examples of suitable formal languages are first order predicate calculus and a programming language for which the semantics has been defined by one of the known techniques (for example, the part of PASCAL that has been axiomatized [27]). However, a program should not be its own specification, because this eliminates the redundancy needed to make verification meaningful. An independent description of desired behavior is always required.

There are advantages in using formal, rather than informal specifications. Formal specifications can be studied mathematically while informal specifications cannot. For example, a correct program can be proved to meet its specifications, or two alternative sets of specifications can be proved equivalent. Formal specifications can also be meaningfully processed by a computer. Certain forms of inconsistency or incompleteness in the specification can be detected automatically [18]. Since this processing can be done in advance of implementation, it should prove to be a valuable aid to program design. In addition, formal specifications can sometimes be realized automatically (for a recent survey see [3]), although the resulting implementation may not be as efficient as one designed by a programmer.

Even in cases where these mathematical tools will not be used, formal specifications are advantageous. When specifications are used as a communications medium among programmers during system design and implementation, it is essential that the programmers reading a specification all agree on what that specification means. This is more likely when the specification is formal, for two reasons. First, there is only one way to interpret a formal specification, because of the well defined and unambiguous semantics of the specification language. Second, the

formality of the language encourages greater rigor in the definitions. It is easy to hide incompletely designed program behavior under vague informal descriptions. Rigorous informal specifications are probably just as difficult to construct as formal ones; informal specifications appear easier to construct because they are usually incomplete.

Specifications are a useful component of program documentation. A well written specification will be easier to understand than a program because it is written in a language chosen for ease of expression, rather than for efficiency of implementation. In addition, specifications are helpful during program maintenance and modification: if the implementation of a module is changed, but the specification is still valid, then the modules using that module need not be changed. Again, formal specifications should be provided, so that the meaning of the specification is well defined.

Formal and informal specifications can complement one another nicely. Informal specifications have the virtue that the main points of the behavior can be communicated in an understandable and effective manner; unfortunately, often details of the behavior are not specified. Formal specifications contain all the details, but there may be insufficient emphasis of the main points. Therefore, we recommend that formal specifications always be accompanied by informal specifications as comments. In this way, the reader can get the idea of the specification quickly and easily, but also has sufficient information to understand fully what is meant.

2. Specifications for Sequential Programs

Specifications are closely related to modularization. If a module implements a clean abstraction, then it will have a simple specification. Conversely, if a module performs an arbitrarily chosen set of actions, sharing logical interdependencies with other modules, then the specification is likely to be at least as complicated as the implementation, and hence virtually

useless.

There are two kinds of abstractions that have proved to be particularly useful in program construction: procedural abstractions and data abstractions. A procedural abstraction performs a mapping from a set of input values to a set of output values. The domain and range of a procedural abstraction consist of data abstractions, and the behavior of the procedural abstraction is defined in terms of the behavior of these data abstractions. A data abstraction provides a set of data values and a set of operations to manipulate the values. Although each operation can be thought of as a procedural abstraction, it is more convenient for implementation and specification to treat the data abstraction as a unit.

In the following sections we present an overview of some existing formal techniques for specifying procedural abstractions and data abstractions, assuming that we are dealing with sequential programs only. Then we consider some of the problems with existing techniques.

2.1 Procedural Abstractions

A procedural abstraction can be viewed as a mapping from its inputs to its outputs. Procedural abstractions are implemented as procedures or subroutines; common examples are a square root routine, a sort package, and a compiler.

Some procedures exhibit nonfunctional behavior (compute different results for the same inputs on different occasions). This is because they have implicit inputs (for example, global variables they read) or outputs (global variables they change). If all of the logical inputs and outputs are considered, then any procedure can be described as a mapping from inputs to outputs.[1] For example, a random number generator may have a state variable called the seed, which

1. This approach works provided the procedure does not depend on some parallel activity (e.g., input from a terminal, the real time clock). Parallelism is discussed in Section 3.

it uses to compute the output value, and which it updates in preparation for producing the next value, so that the seed is both an implicit input and an implicit output. Such a random number generator can be described as a function of type (seed --> seed X value).

The specification of a procedural abstraction has two parts: the interface specification and the behavioral specification. The interface specification consists of the name of the module, and the types and sources or destinations of the inputs and outputs. (Typical sources for input and output are formal parameters of the procedure or global variables.) The interface information is syntactic in nature, and must be given in the declarations of most high level languages, at least for the explicit inputs and outputs.

There are two main techniques for specifying the behavior of procedural abstractions that meet the criterion of formality: input/output specifications and operational specifications. In either case, the module is treated as a black box, and the specification describes the relationship between the inputs and outputs of the module.

2.1.1 Input/Output Specifications

The input/output approach describes the relationship between the inputs and the outputs by giving a pair of constraints. Provided that the actual input satisfies the input constraints, the output is guaranteed to satisfy the output constraints.

Early work on formal I/O specifications was done by Naur [37], Floyd [13], and Hoare [23]. Their work was motivated largely by a desire to *prove* that programs have certain properties (e.g.,"correctness"). They attached assertions to various points in a program, and sought to prove that the assertions were true whenever control reached the associated points in the program. The assertions were expressed in the ordinary notation of mathematical logic, since this is a natural language in which to do proofs.

Each pair of assertions acts as a specification for the program fragment between them. Hoare introduced the notation P{program text}Q, for expressing I/O specifications, where the assertions P and Q are sentences of mathematical logic. Assertions are written in terms of program variables, logical variables, and the names of procedures. A program variable in an assertion stands for the value of the variable at the instant when control reaches the point in the program associated with the assertion. The logical variables have static values and are implicitly universally quantified. A procedure name in an assertion denotes the function computed by the associated procedure: the arguments of the function are the parameters of the procedure, and the result is the return value of the procedure. Assertions should refer only to procedures that are functional and have no side effects. In the case of nonfunctional procedures or procedures with side effects, the specification technique should establish standard notational methods for referring to the functions that compute the outputs and the side effects of a procedure (e.g., see [24]).

A small but nontrivial example of an I/O specification for a well known functional abstraction, the greatest common divisor, is given in Figure 1. The interface specification tells us that *gcd* has no side effects, since the return value is the only output. The input assertion states that both inputs must be positive. The output assertion states that both inputs must be evenly divisible by the output, and that the output must be the greatest such number, in the sense that it must be divisible by any common divisor of the inputs. Note the use of the abbreviation,

---

Figure 1. I/O Specifications for gcd.

Interface: gcd (integer, integer) returns integer

Behavior: x > 0 & y > 0 {gcd(x, y)} divides (gcd (x, y), x) & divides (gcd (x, y), y) & ∀i: integer [divides (i, x) & divides (i, y) ==> divides (i, gcd (x, y))]

Abbreviations: divides (x, y) ≡ ∃i: integer [y = x * i]

divides(x, y), to enhance the readability of the specification. An important part of constructing specifications is the identification of appropriate abbreviations; such abbreviations play a role in specification construction analogous to the role played by procedural and data abstractions in program construction.

Hoare's notation P{program text}Q specifies only partial correctness: it states how the program behaves, provided that it terminates. Note that such a specification is satisfied by an infinite loop. The definition was formulated in this way because it is often convenient to use different techniques for showing partial correctness and termination. Proof techniques based on I/O constraints that specify total correctness have also been developed [10, 11, 35]. In these formalisms, whenever the input constraints are met the program is guaranteed to terminate in a state satisfying the output constraints. The specification of an abstraction necessarily includes the termination requirements,[2] regardless of whether the proofs are done separately or together.

One of the main benefits obtainable from specifications of modules is that proofs of program properties can be decomposed using such specifications: to prove something about the result of a procedure call, one need only refer to the specifications, and not to the code implementing the function. Hoare [24] has studied some of the issues involved in proofs of programs containing procedure calls. His discussion is complicated by the possibility that data may be shared between different parameters of a procedure. If a function designed under the assumption that two input objects are independent is passed actual input objects that are identical or share subcomponents, then quite startling and unexpected behavior may result, especially if the function changes the input objects. These complications do not apply to parameters that are passed by value (copied).

---

2. If the program is not intended to terminate, then this must be explicitly stated.

2.1.2 Operational Specifications

The assertions of an input/output specification describe properties of program states; the computation that transforms a legal input state into a legal output state is not described explicitly. In an operational specification, the transformation is described explicitly by giving a program that computes the intended function. An operational specification differs from an implementation because, for the purposes of specification, simplicity is important and efficiency is not. The specification language should be chosen to make the specification as simple as possible. Note that it is not necessary to have an implementation for the specification language (it may be convenient for testing purposes), although the specification language must have a precisely defined meaning. In [96], McCarthy gives some early examples of operational definitions of functional abstractions. McCarthy's formal language is very simple, using only recursion and conditional expressions.

An operational specification for the greatest common divisor function is given in Figure 2. The specification uses the integer operation *min*, which returns the lesser of its two arguments and the integer operation *mod*, which returns the result of reducing its first argument modulo the second argument. The function *search-from* starts with the largest number that has a chance of being a common factor of x and y, $min(x,y)$, and tries all possibilities, biggest first, until it finds a common divisor. The algorithm will always terminate, since the inputs are positive, and since 1 is a

---

Figure 2. Operational Specification for gcd.

Interface: gcd (**integer**, **integer**) **returns integer**

Behavior: gcd (x,y) = **if** $x \leq 0 \vee y \leq 0$ **then error** ("unexpected input")
else search-from (x, y, min (x, y))

Abbreviations: search-from (x, y, z) = **if** mod (x, z) = 0 & mod (y, z) = 0
then z
else search-from (x, y, z - 1)

divisor of any number. This function does indeed satisfy the I/O specifications given above, although it is not easy to prove that it does because the notions of "greatest" used by the two specifications are different.

A proof that a procedure correctly implements the abstraction defined by an operational specification is really a proof of the equivalence of two programs. Methods have been developed for doing such proofs when the two programs are recursive [36, 16].

There is a hazard of inadvertently giving incomplete specifications for both the I/O constraint and the operational specification techniques. For the I/O constraint method, it is possible to make the output constraints too strong, so that for some inputs there is no output value satisfying them. For the operational technique, it is possible to write incomplete conditionals or nonterminating recursions. Such specifications are misleading, and should not be written on purpose even if they specify the desired behavior. If incomplete specifications are desired, then the conditions under which the output is not defined, or in which an error occurs, should be explicitly identified.

2.2 Data Abstractions

A data abstraction consists of an abstract data type, or a family of related abstract data types. An abstract data type is a set of objects capable only of particular kinds of behavior, which correspond to a finite set of allowable operations on objects of the type [31]. All other operations on the data type must be realized using the ones in the finite set. The meaning of an abstract data type is completely characterized by the behavior of the operations; this is guaranteed by limiting access to the representation of an object to just the operations of the type. Note that the data representation may be changed freely, provided only that the behavior of the operations is preserved. All data types can be cast into this framework. For a discussion and a multitude of

examples, see [26].

Common examples of data abstractions are fixed point numbers, arrays, and databases. Data types may be provided by the programming language, or they may be defined by the programmer. Some programming languages (such as Alphard [53], CLU [33], and Simula67 [8]) provide support for user-defined data types, and user-defined types can be simulated in any programming language by appropriate coding conventions.

A data abstraction also has an interface specification and a behavioral specification. The interface specification consists of the name of the type, and the names and types of the associated operations. In the following, we discuss two methods of specifying the behavior of an abstract data type: axiomatically and via an abstract model.

2.2.1 Axiomatic Specifications

Axiomatic specifications define the behavior of an abstract data type by giving axioms relating the operations. There are a variety of axiomatic approaches. One of the best developed methods is based on data algebras; using the recently developed theory of heterogeneous algebras [4], this approach was developed by Zilles [32,54], Goguen et al. [14], and Guttag [18].

All objects of an abstract data type must have been produced by some sequence of the constructor operations of that type. Other operations of the type, called inquiry operations, yield results of different types and provide the only way to extract information from an object of the type. A complete set of axioms has to define the values of the inquiry operations for any object of the abstract data type.

Let us consider the array data abstraction as an example. This abstraction corresponds to a family of data types, parameterized by the type t of the array elements. Arrays are considered to be objects that can be created, changed, and interrogated. There are five array operators. The

alloc operation creates a new array; parameters to *alloc* specify the lower and upper bounds of this array. The *store* operation changes an array by replacing one of its elements, while the *fetch* operation retrieves an element. The *top* and *bottom* operations yield the upper and lower bounds of the array. Note that we consider an assignment to an array element to be equivalent to an operation on the array, which changes the state of the array as a whole. For uniformity, we use functional notation, introducing the operations *store* and *fetch*. In more conventional notation, *store*(*x*, *i*, *a*) would be written as *a*[*i*] := *x*, and *fetch*(*i*, *a*) as *a*[*i*].

Figure 3 shows a specification for arrays. The specification has two parts, specifying the

---

Figure 3. Axiomatic Specification for the Type Array [t].

Interface:

alloc (**integer**, **integer**)	**returns** array [t]
store (x: t, v: **integer**, a: array [t])	**changes** a
fetch (**integer**, array [t])	**returns** t
bottom (array [t])	**returns integer**
top (array [t])	**returns integer**

Axioms:

1. alloc (i1, i2) = **if** i1 > i2 **then error** ("bad array size")

2. store.a (x, i, a) = **if** i < bottom (a) ∨ i > top (a) **then error** ("index out of bounds")

3. fetch (i1, alloc (i2, i3)) = **if** i1 < i2 ∨ i1 > i3 **then error** ("index out of bounds")
   else UNDEFINED

4. fetch (i1, store.a (x, i2, a)) = **if** i1 = i2 **then** x **else** fetch (i1, a)

5. bottom (alloc (i1, i2)) = i1

6. bottom (store.a (x, i, a) = bottom (a)

7. top (alloc (i1, i2)) = i2

8. top (store.a (x, i, a)) = top (a)

interface and the behavior of the data abstraction. In addition to the types of the inputs and outputs of the operations, the interface specification tells us that the operations *alloc*, *fetch*, *top*, and *bottom* return values, but have no side effects, while the operation *store* returns no value, but changes the state of its third argument, labelled "a". The axioms in Figure 3 should be interpreted as describing the relationship between the operations and the state of an array. The transformation on the third argument of *store* is denoted by *store.a*. The axioms apply only if the arguments of the operations satisfy the type constraints given in the interface specification. The axioms are numbered for ease of reference.

According to the last four axioms, the arguments to *alloc* specify the bounds of a new array, and the *store* operation does not change the bounds. Thus arrays have fixed sizes, since there are no other operations that change arrays.

According to the first axiom, arrays must have at least one element, and an attempt to create an empty array will cause a runtime error. According to the second and third axioms, any attempt to examine or modify an element outside the bounds will also cause an error.

Axiom 4 says that the *store* operation updates the indicated element of the array and has no effect on the rest of the elements. Axiom 3 says that all of the elements of a new array are undefined; this means that the axioms do not constrain the behavior of the implementation in this case.

The interested reader may want to compare this axiomatization to the one given by Hoare [26], which presents a cleaner and more abstract view of arrays in which all array elements have defined values. We believe that the arrays described by Hoare are better than the ones we define. We chose to define the behavior as above because it corresponds more closely to that provided by common programming languages.

2.2.2 Abstract Model Approach

In the abstract model approach, the objects of the data type are represented in terms of other data abstractions with known properties established by formal (probably axiomatic) specifications given in advance. Then the operations of the type being defined can be specified in terms of the operations of the known abstractions selected as the representation. The operations are specified using the methods for specifying procedural abstractions. Often it is most convenient to give I/O specifications for some of the operations, and operational specifications for the rest.

This approach is analogous to the operational method for procedural abstractions, where a function is specified by giving a particular algorithm for computing it. It must be emphasized here again that clarity and simplicity are important for specifications, while efficiency is not. It is often convenient to choose representations that use objects of standard mathematical domains, such as sets and sequences, which are not supported by most programming languages. Careful choice of representation can greatly simplify the specifications of the operations.

As an example, we will give an abstract model representation for the array data abstraction described above. The representation will use tuples and sequences.

A tuple is a set of named elements, which may be of different types. Tuples are like mathematical Cartesian products, except that the components are referenced by named selectors rather than by numerical indices. Tuples are similar to records in the programming language PASCAL [52], except that they are static (cannot be updated). For example,

tuple[a: integer, b: real]

denotes a tuple type, with two components whose selectors are "a" and "b", and whose types are integer and real.

[a: 3, b: 4.1]

is an element of this type. If x denotes this element, then

x.a = 3 and x.b = 4.1.

Sequences contain zero or more elements, all of which must have the same type; the elements are numbered from 1 to n. Sequences cannot be updated. Sequence operations include:

<>	Denotes the empty sequence.
length(s)	Returns the number of elements in s.
s_i	If $1 \le i \le$ length(s), then the ith element of s.
addfirst(x, s)	Creates a new sequence with x as the first element, followed by the elements of *s* in order.
butfirst(s)	If length(s) > 0, returns a new sequence containing all but the first element of s in their original order.

An abstract model specification for arrays is shown in Figure 4. The "low" and "high" components of the representation are the bounds of the array, while the "elements" component contains information about the array elements with defined values; each such element is represented as a pair consisting of a value and its index in the array. If the bounds are legal, the *alloc* operation returns an array without any defined elements. The *top* and *bottom* operations return the appropriate components of the representation. The *store* operation simply adds the new index-value pair to the front of the elements sequence, without bothering to remove any old elements. The *fetch* operation searches the elements sequence from the front, finding the most recently stored value, if there is one. Note that the definition of *fetch* does not state what happens when an attempt is made to fetch an undefined element.

It is important not to read too much into abstract model specifications for data abstractions, or into operational specifications for procedural abstractions. The implementation must have the same behavior as the specifications, but it is not constrained to using the same representations and algorithms for realizing that behavior. For example, an implementation of

Figure 4. Abstract Model Specification for the Type Array [t]

Interface:

```
alloc (integer, integer)                  returns array [t]
store (x: t, v: integer, a: array [t])    changes a
fetch (integer, array [t])                returns t
bottom (array [t])                        returns integer
top (array [t])                           returns integer
```

Representation:

```
array [t] = tuple [ low: integer,
                    high: integer,
                    elements: sequence [tuple [index: integer, value: t]] ]
```

Operations:

```
alloc (i1, i2) = if i1 ≤ i2  then {low: i1, high: i2, elements: <>}
                             else error ("bad array size")

bottom (a) = a.low

top (a) = a.high

store.a (x, i, a) = if  a.low ≤ i ≤ a.high
                    then {  low: a.low,
                            high: a.high,
                            elements: addfirst ({index: i, value: x}, a.elements) }
                    else error ("index out of bounds")

fetch (i, a) = if a.low ≤ i ≤ a.high  then getval (a.elements, i)
                                      else error ("index out of bounds")

getval (elements, i) =  if length (elements) = 0 then UNDEFINED
                        else if elements_1.index = i then elements_1.value
                        else getval (butfirst (elements), i)
```

---

arrays is not constrained to retain the old array values. Indeed, the strategies used for the specification and the implementation often *should* be different, because speed and simplicity do not always go together.

2.2.3 Comparison of Axiomatic and Abstract Model Techniques

It is not clear which of the abstract model and axiomatic approaches is better. Abstract model specifications are probably easier for programmers to construct and understand, since they are more like programs; however, they tend to supply detailed information that is not really part of the abstraction.

Proofs of program properties can be given using either specification techniques. Proofs of implementations are probably equally difficult in the two approaches, although this is still a matter of research. A proof in the axiomatic approach must show that the axioms are satisfied by the operations of the implementation [19]; in the abstract model approach, a homomorphism is constructed from (an algebra derived from) the implementation to (the algebra described by) the abstract model [25, 53].

The axiomatic technique is well suited to proofs of programs using the data abstraction: the axioms, and theorems derived from them, can be used directly in the proof. For abstract model specifications, however, it is probably better to prove a set of theorems from the specifications and then use the theorems in proofs rather than use the specifications directly.

The complexity of a specification in either approach depends on the complexity of the abstraction. In general, the more potential error conditions an abstraction has, the more complicated is its behavior. For example, the array definitions in Figure 3 and Figure 4 are complicated by the fact that array elements can have undefined values. As was mentioned earlier, we believe that this is a deficiency of the array abstraction. In our experience of writing specifications we have found that slight changes in an abstraction often result in both a simpler specification and a better abstraction; this is another reason why specifications can be a useful aid in design.

The specification technique proposed by Parnas [39], in which a module is viewed as a state machine, can be formalized using either the axiomatic or the abstract model approach. Formalization using axioms is being investigated by Parnas [41]. Formalization via the abstract model approach is appropriate for the specification technique in use at SRI [47], in which "hidden" functions are added to the state machine when needed to define the behavior completely; this formalization is being studied at SRI and at MIT [44].

2.2.4 Problems for Further Research

There are a number of issues concerning procedural and data abstractions that are not sufficiently well understood. One such issue is how to specify the behavior of a program when an error is detected during execution. Models for error handling are needed both for programming languages (see [15]) and for specification techniques. In the absence of such a model, the behavior of a program in the presence of errors is often left unspecified, and winds up being determined by what is easiest to implement at each installation. At best, as in the examples above, the errors are named, and the conditions under which they occur are specified. Parnas has described one model for program behavior in the presence of errors [39, 40], but his model is informal and incomplete. Another model that appears to be promising is described in [48].

Side effects are also not well understood. A side effect is said to have occurred when the value of some variable, x, changes, but no explicit assignment to x has been made. Since all changes to values must occur through variables, there must be some other variable, y, which refers to the value of x. Two common ways in which variables come to share the same values are through a call by reference, or through pointers.

Hoare [24] discusses some of the issues introduced by sharing via the call by reference mechanism. It is not hard to describe the effects of procedures that change their arguments. The

difficulty lies in describing the effects of sharing that may occur at some times and not at others. For instance, it is possible to cause two parameters passed by reference to share the same variable by invoking a procedure with that variable for both arguments. The behavior in such a case may be different than if the inputs were distinct. Specifications and proofs must treat such cases separately. As the number of variables that may be sharing a value increases, the number of ways the sharing may happen increases drastically, making the enumeration impossibly tedious. Furthermore, these special cases are often uninteresting. We believe that linguistic methods for limiting sharing are desirable (e.g., [30]) but also that more convenient notations for describing shared data are needed.

An additional construct that is useful in sequential programs is the control abstraction, a program unit that produces a sequence of values, one at a time, for use in a generalized **for** statement (see [33] and [49] for discussions of control abstractions). Some work has been done on specification and verification of control abstractions [49], but more is needed. Control abstractions introduce a kind of quasi-parallelism between the program producing the values and the body of the **for** loop; it may be that specification techniques for parallel programs, discussed below, are appropriate here.

3. Specifications of Parallel Programs

A sequential computation has a single site of execution activity, where the instructions of a program are executed one after another by a *process*. Sequential programs are executed by only one process at a time. In a parallel computation there may be many sites of activity, with a separate process executing instructions at each site. A parallel program is executed by an arbitrary (and possibly variable) number of processes, where more than one process may be actively executing instructions at the same time.

If a process can run without communicating with other processes, then the program executed by that process can be viewed as a sequential program, and the techniques used in the preceding section can be used to specify its behavior. This is often the case in a time sharing system, where the jobs belonging to different users do not interact. However, for many useful and interesting applications, processes must cooperate to perform some joint task. In this section, we discuss specification techniques for describing the behavior of parallel programs in which concurrent processes interact.

Concurrent processes can interact either by explicitly passing each other messages via a queue or I/O stream, or by changing the state of a shared data object. We will assume that processes interact by changing shared data. However, our discussion applies also to situations where processes interact by passing messages, since the I/O stream can be treated as a shared data object.

Interacting processes generally have to be *synchronized* to keep them from interfering with each other. For example, a process performing a computation sequence that changes the state of a shared object can interfere with another process in the middle of a computation sequence that depends on the state of the same object. Computation sequences that change or depend on the state of a shared object are known as *critical sections* with respect to that object. Whenever a process is in a critical section, all other processes have to be prevented from performing some subset of the possible operations on the associated shared object in order to prevent destructive interference.

New techniques are necessary for specifying the behavior of cooperating parallel programs because it is necessary to express new kinds of information. In a sequential program, the values passed in as parameters (or free variables) are the same as the values that get operated on, so that the behavior of a procedure can be viewed as a function from the inputs to the outputs. When

there is concurrent activity, some other process may change the states of some of the input objects between the time of the call and the time the procedure actually gets executed, as well as several times during the execution. Sometimes this kind of behavior is desired, and sometimes it is not.

For example, consider the specification of an operation, *dequeue*, that dequeues an element from a FIFO queue. If the input queue is not empty, *dequeue* should remove and return the oldest element on the queue. This behavior is desired for both sequential and parallel programs. The desired behaviors for empty queues are different, however. In a sequential program, some sort of error condition is desired, because there is no oldest element, and never will be. In a parallel program, the desired behavior is that *dequeue* should wait until the queue is no longer empty (presumably some other process will eventually place an element in the queue).

Note that the value returned by a *dequeue* operation is not determined solely by the input values it receives. The value depends also on the inputs to subsequent *enqueue* operations, and on the history of previous *enqueue* and *dequeue* operations (including the number of previous *dequeue* operations also waiting for values).

Thus we see that a procedure cannot always be viewed as an indivisible operation in a parallel programming environment. In the queue example, we have to consider two suboperations, the request to retrieve an object from the queue, and the actual retrieval. Another example is the readers/writers problem [7], where three suboperations are considered for readers: attempting to read, actually reading, and end of read.

Just as for sequential programs, careful modularization is needed to make the specifications manageable. The hard problem in parallel programming is to synchronize the processes correctly, which must be done whenever processes share data. If data abstractions are used properly, the operations of the abstract data type can be made to coincide with the critical sections associated with objects of that type, and hence processes need to be synchronized only when

they perform these operations. Thus the specification of a data abstraction can include a local and complete specification of the synchronization requirements for all processes using the data abstraction. Much current research is concerned with defining linguistic mechanisms to support the use of data abstractions in parallel programming (e.g., monitors [5, 28, 29], serializers [22]). Languages where interprocess interactions are limited solely to messages passed via shared message buffers are also being investigated (eg. Gypsy [1]).

Specification techniques for parallel programs are not as well understood as are techniques for sequential programs. However, some techniques are emerging that appear to be promising. These techniques differ in the way information about the activities of concurrent processes is represented. Two approaches are discussed below.

3.1 The State Variable Approach

One way to specify the behavior of a parallel program is to describe the states of the machine before and after the program is executed. Owicki has followed this approach. In [38] she extended Hoare's I/O constraint technique to parallel programs, by adopting his notation and giving proof rules for a language with primitives for parallel execution and synchronization. Owicki's work, like Hoare's, is geared toward proofs of program properties. The work reported in [38] deals with arbitrary program fragments, and hence can also be used to specify the operations of a data abstraction. Owicki is currently working on applying her techniques to the problem of specifying and verifying the behavior of data abstractions shared by concurrent processes.

Owicki found that her axiomatization is incomplete without a rule of inference which says that a program has a property if it can be shown that the program, when augmented by adding auxiliary variables, has that property. The auxiliary variables are used to encode information about the interactions between processes in the state of the program. For example, an auxiliary

variable may be used to record the number of processes that are currently performing a given operation on a given shared data object. Auxiliary variables may be introduced into a program only as the targets of assignment statements. Their values may not be used by the program, although they may be used in assertions about the program. Consequently, the auxiliary variables can be removed from a program without affecting the outcome of the computation, and any property of a program that can be proved by introducing auxiliary variables must also hold for the original program.

An example of a specification using Owicki's technique is shown in Figure 5. The program fragments labeled $S_1, \ldots, S_n$ (separated by the delimiters "**cobegin**", "//", and "**coend**") are all identical except for the auxiliary variables. Each program fragment is executed by a separate process, and each contains a critical section. We wish to specify that the critical sections are mutually exclusive: no two distinct processes may be executing a critical section at the same time. In order to express this property formally, we introduce an array of auxiliary variables called *InCS*, which is indexed by the subscripts of the process labels S_i. All of the components of *InCS*

Figure 5. State Variable Specification of Mutual Exclusion

```
Incs := 0;
cobegin  S1: ...
         // ... //
         Si:  while true do
                <non critical section>
                InCS(i) := 1;
                <critical section>
                InCS(i) := 0;
                <non critical section>
              end;
         // ... //
         Sn: ...
coend;
```

Invariant: $\forall\, i,j\ [\ (1 \le i \le n\ \&\ 1 \le j \le n\ \&\ \neg(i=j)) \Longrightarrow \neg(InCS(i)=1\ \&\ InCS(j)=1)\ ]$

are initially zero. Just before process S_i enters its critical section, *InCS* (*i*) is set to one, and just after process S_i leaves its critical section, *InCS* (*i*) is set to zero. These are the only places in the program where *InCS* is mentioned. Therefore we know that *InCS* (*i*) = 1 whenever process S_i is in its critical section. The invariant assertion says that no two distinct components of *InCS* have the value one, which implies that no two processes are in their critical sections at the same time, as we require. The invariant is asserted to hold at all points in the program between the **cobegin** and the **coend** statements. This example is adapted from [38], where Owicki proves that the appropriate invariant holds for a program with a similar structure and a particular synchronization scheme, thus establishing that the mutual exclusion requirement is met.

Robinson and Holt [46] have considered the problem of specifying the synchronization constraints associated with shared data as a part of the specification of the data abstraction. For each operation, they introduce an abstract program consisting of a sequence of suboperations. Some of the suboperations update "state functions", which are generally used to count the number of processes currently active in a given section of code (cf. Owicki's auxiliary variables). The synchronization constraints are expressed by invariant assertions involving the state functions before and after certain suboperations of the abstract program. Just those program states that do not violate the invariant are legal. Suboperations that would result in an illegal program state have to be suspended until it is safe for them to proceed.

Robinson and Holt point out that a proof of correctness of an implementation must include a proof that the implementation preserves the invariants. They also discuss a way of verifying the correctness of the *specification*. This involves constructing the state machine that has those state transitions allowed by the invariant. This state machine can be checked automatically for various properties, such as freedom from deadlock; it may also be presented to the user for review. Methods for determining properties of specifications are even more important for parallel

than for sequential programs, because it is especially difficult to determine that the specified behavior does indeed correspond to the user's expectations when there are interacting concurrent processes involved.

3.2 The Event Approach

The earliest event oriented specifications for concurrent programs are path expressions [6, 20, 12]. Path expressions are tied to data abstractions; the execution of a suboperation of an operation of the data type is taken to be an event. Path expressions are constraints on the order in which events can happen. Habermann gave informal descriptions of several versions of path expressions, which he intended as synchronization primitives for a high level programming language; for one version [6] an algorithm for implementing path expressions using Dijkstra's P and V operations [9] was presented. Lauer and Campbell [34] gave a formal definition of a slightly different version, using the Petri net formalism [21, 42]; this kind of path expression is a regular expression (as in automata theory), which generates a set of strings corresponding to all of the legal sequences of events. Finally, Flon and Habermann [12] discussed proofs of correctness of parallel programs specified by path expressions, connecting a new kind of path expression with invariant assertions.

A more formal and abstract view is taken by Greif [17]. This work is based on the assumption that it is not possible to define a global notion of time, or a total ordering on events, that all observers can agree upon. Rather, a local view is taken; two events are ordered only if some communication has taken place that allows the ordering between the events to be deduced, much as in the relativity theory of physics. Time is described as a partial ordering, so that there may be pairs of events that are not ordered with respect to one another, and hence can be "simultaneous" or "concurrent". An event is supposed to be instantaneous and indivisible. Events

are usually defined as the beginning or the end of some step in the computation.

Greif describes a computation as a partially ordered set of events: some events are known to come before certain other events. The partial order can be viewed as a history of the computation. Each event is associated with exactly one process, so that the set of events in a computation can be partitioned into disjoint subsets (one for each process).

Synchronization constraints are expressed as axioms constraining the legal partial orders, admitting some computations but not others. For example, Figure 6 shows the axiom for mutual exclusion. S_1 and S_2 denote the sets of events corresponding to two distinct invocations of the critical section (possibly by different processes).[3] The axiom says that the computations S_1 and S_2 may not overlap in time: either every event of S_1 must happen before any event of S_2 does, or vice versa. The notation "x < y" means that the event x occurs strictly before the event y. Note that the axiom is independent of the number of processes involved, so that it can be used even in situations where the number of processes is variable.

Greif's formalism is powerful enough to handle real concurrency, with multiple processors that may be in widely scattered locations. The formalism can be used to state requirements concerning priorities and various other fine points of scheduling, such as the absence of deadlock and starvation. It can not describe performance properties, such as throughput, because only the order of events is discussed, and not the actual elapsed time between two ordered events.

Figure 6. Event Axiom Specification for Mutual Exclusion

$$(\forall e_1 \in S_1 \, \forall e_2 \in S_2 \, [e_1 < e_2]) \lor (\forall e_1 \in S_1 \, \forall e_2 \in S_2 \, [e_2 < e_1])$$

3. Although we have not done so here, the sets S_1 and S_2 can be defined formally using Greif's techniques.

4. Future Research Directions

We believe that the use of formal specifications can enhance the reliability and decrease the cost of software, and that research in this area should be supported and encouraged. In the remainder of this section, we discuss future research directions that we believe have promise.

In Sections 2 and 3, we discussed current work in specification techniques. Considerable gains have been made in understanding and defining specifications, but much remains to be done. In the area of sequential programs, useful specification methods have been invented for both functional and data abstractions. The main work that needs to be done is to establish proof techniques associated with the various methods, to extend the methods to those aspects of program behavior not yet covered (e.g., errors, side effects and control abstractions), and to develop specifications languages that make the techniques easy to use.

In the area of parallel programs, specification methods need considerable development. It is not even clear what the important aspects of the behavior of a parallel program are. The concepts need to be developed before they can be precisely defined. For example, the input/output behavior of a module is usually treated separately from other aspects, such as the priority and the degree of concurrency. However, these aspects are not completely independent, and it is not yet known how to describe the relationship between them in a useful way. Once standard concepts dealing with orderings and synchronization are developed, standard notations for these concepts should be introduced, with the result that specifications can be given at a higher level.

As specification languages come into existence, they should be used to specify programs so that the techniques can be evaluated. We have only limited experience with specification of programs of reasonable size [43, 45]. The study of specifications should focus on well-modularized programs consisting of modules supporting data and functional abstractions. The utility of

specifications in program construction should also be evaluated.

The specification methods discussed in this paper are suitable for describing a program design, or at least those parts resulting in modules supporting data or functional abstractions. To the extent that an entire system can be considered to be an abstraction, or a group of abstractions, the techniques ought to be useful for describing system behavior, although the difficulty of describing very high level abstractions is formidable (consider the complexity of a specification of a programming language processor). Attempts to specify real applications will be valuable in determining the applicability of the techniques to describing entire system behavior.

The use of specifications during program design will be enhanced if tools exist for increasing the level of confidence that a set of specifications does indeed capture the intended concepts and behavior. For instance, properties and implications of a specification can be derived, and reviewed by the user [18]. A catalog of properties that are useful gauges of behavior and a set of methods for extracting these properties from a specification are needed as a preliminary step towards building an automated facility to assist the user in evaluating a proposed specification or system design.

For specifications to contribute most effectively to the programming process, computer support of specifications is desirable. We expect that ultimately there will be systems that support program development in an integrated fashion. Such systems will be organized around a data base containing information about abstractions (see [33] for a description of such a system). For each abstraction, the data base will include specifications, source code and object code of implementations, and the known properties of the abstraction (e.g., that a particular implementation has been verified). Many programs will run on the data base, including specification and programming language processors, and verifiers.

Very little work has been done in the area of formal requirements specifications. In one

study that has been done [2], properties of the desired system behavior were specified using a model. The specifications resulting from the system design must be checked to see if they satisfy these properties, which we envision as a set of interdependent constraints the user expects the system to satisfy. We speculate that the techniques used in formally specifying a system design can be carried over to this phase, with the benefit that conflicts and inconsistencies among the requirements can be caught early.

References

1. A. L. Ambler, et al. "GYPSY: A Language for Specification and Implementation of Verifiable Programs". *Proc. of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12*, No. 3 (March 1977).

2. D. E. Bell and L. J. LaPadula. "Secure Computer Systems". Report ESD-TR-73-278, Mitre Corp., Bedford, Mass., November 1973.

3. A. Bierman. "Approaches to Automatic Programming". *Advances in Computers, Vol. 15* (1976), pp. 1-63.

4. G. Birkhoff and J. D. Lipson. "Heterogeneous Algebras". *J. of Combinatorial Theory, Vol. 8* (1970), pp. 115-133.

5. P. Brinch Hansen. *Operating System Principles*. Prentice Hall, Englewood Cliffs, N. J., 1973.

6. R. H. Campbell and A. N. Habermann. "The Specification of Process Synchonization by Path Expressions". *Lecture Notes on Computer Science, Vol. 16*, Springer Verlag, Heidelberg-Berlin-New York, 1974.

7. P. J. Courtois, F. Heymans, and D. L. Parnas. "Concurrent Control with 'Readers' and 'Writers'". *Comm. of the ACM, Vol. 14*, No. 10 (Oct. 1971), pp. 667-668.

8. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. "The Simula 67 Common Base Language". Publication No. S-22, Norwegian Computing Center, Oslo, 1970.

9. E. Dijkstra. "Co-operating Sequential Processes". *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 1968.

10. E. Dijkstra. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs". *Comm. of the ACM, Vol. 18*, No. 8 (Aug. 1975), pp. 453-457.

11. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation, Prentice Hall, Englewood Cliffs, N. J., 1976.

12. L. Flon and A. N. Habermann. "Towards the Construction of Verifiable Software Systems". *Proc. Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8*, No. 2 (1976), pp. 141-148.

13. R. W. Floyd. "Assigning Meanings to Programs". *Proc. Amer. Math. Soc. Symposium in Applied Mathematics Vol. 19* (1967) pp. 19-31.

14. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. "Abstract Data Types as Initial Algebras and the Coorrectness of Data Representations". *Proc. of Conference on Computer Graphics, Pattern Recognition and Data Structures*, 1975, pp.

89-93.

15. J. B. Goodenough. "Exception Handling: Issues and a Proposed Notation". *Comm. of the ACM, Vol. 18*, No. 12 (Dec. 1975).

16. I. Greif. "Induction in Proofs about Programs". Master's Thesis, Technical Report MAC TR-93, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1972.

17. I. Greif. "Semantics of Communicating Parallel Processes". Ph. D. Thesis, Technical Report MAC TR-154, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1975.

18. J. V. Guttag. "The Specification and Application to Programming of Abstract Data Types". Ph. D. Thesis, University of Toronto, Report CSRG-59, 1975.

19. J. V. Guttag, E. Horowitz and D. R. Musser. "Abstract Data Types and Software Validation". Report ISI/RR-76-48, Information Sciences Institute, University of Southern California, August 1976.

20. A. N. Habermann. "Path Expressions". Dept. of Computer Science, Carnegie-Mellon University, 1975.

21. M. H. T. Hack. "Analysis of Production Schemata by Petri Nets". Master's Thesis, Technical Report MAC TR-94, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1972.

22. C. Hewitt and R. Atkinson. "Parallelism and Synchronization in Actor Systems". Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1976.

23. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Comm. of the ACM, Vol. 12*, No. 10 (Oct. 1969), pp. 576-583.

24. C. A. R. Hoare. "Procedure and Parameters - an Axiomatic Approach". *Symposium on the Semantics of Algorithmic Languages*, E. Engeler, Ed., Springer Verlag, Berlin-Heidelberg-New York, 1971.

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

26. C. A. R. Hoare. "Notes on Data Structuring". *Structured Programming*, A.P.I.C. Studies in Data Processing, No. 8, Academic Press, London-New York, 1972.

27. C. A. R. Hoare and N. Wirth. "An Axiomatic Definition of the Programming Language Pascal". *Acta Informatica, Vol. 2*, No. 4 (1973), pp. 335-355.

28. C. A. R. Hoare. "Monitors: an Operating System Structuring Concept". *Comm. of*

the ACM, Vol. 17, No. 10 (Oct. 1974), pp. 549-557.

29. J. H. Howard. "Proving Monitors". *Comm. of the ACM, Vol. 1*, No. 5 (May 1976).

30. B. W. Lampson, J. J. Horning, R. L. London and G. L. Popek. "Report on the Programming Language Euclid". *ACM SIGPLAN Notices, Vol. 17*, No. 2 (February 1977).

31. B. H. Liskov and S. Zilles. "Programming with Abstract Data Types". *Proc. of the ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, Vol. 9*, No. 4 (April 1974), pp. 50-59.

32. B. H. Liskov and S. Zilles. "Specification Techniques for Data Abstractions". *IEEE Trans. on Software Engineering, Vol. SE-1*, (1975), pp. 7-19.

33. B. H. Liskov, A. Snyder, R. Atkinson and C. Schaffert. "Abstraction Mechanisms in CLU". To appear in *Comm. of the ACM*.

34. P. E. Lauer and R. H. Campbell. "A Description of Path Expressions by Petri Nets". *Conference Record of the Second Symposium on Principles of Programming Languages* (Jan. 1975).

35. Z. Manna. "The Correctness of Programs". *J. of Computer and System Sciences, Vol. 3*, No. 2 (May 1969), pp. 119-127.

36. J. McCarthy. "A Basis for a Mathematical Theory of Computation". *Computer Programming and Formal Systems*, Braffort and Hirschberg, Eds., North Holland Publishing Co., Amsterdam-London, 1963.

37. P. Naur. "Proof of Algorithms by General Snapshots". *Bit Vol. 6*, No. 4 (1966), pp. 310-316.

38. S. S. Owicki. "Axiomatic Proof Techniques for Parallel Programs". Ph. D. Thesis, Report TR-75-251, Cornell University, July 1975.

39. D. L. Parnas. "A Technique for the Specification of Software Modules, with Examples". *Comm. of the ACM, Vol. 15*, No. 5 (May 1972), pp. 330-336.

40. D. L. Parnas and H. Wurges. "Response to Undesired Events in Software Systems". *Proc. of the 2nd International Conference on Software Engineering* October 1976.

41. D. L. Parnas and G. Handzel. "More on Specification Techniques for Software Modules". Research Group on Operating Systems I, T. H. Darmstadt, Fed. Rep. of Germany.

42. C. A. Petri. "Kommunikation mit Automaten" (Communication with Automata). Ph.D Thesis, Technische Hochschule, Darmstadt, 1962.

43. W. R. Price. "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems". Ph. D. Thesis, Carnegie-Mellon University, June 1973.

44. R. N. Principato. "A Formalization of the Parnas Module Specification Technique". Master's Thesis, Dept. of Electrical Engineering and Computer Science, Mass. Inst. of Technology, Cambridge, Mass., forthcoming.

45. L. C. Ragland. "A Verified Program Verifier". Ph. D. Thesis, Report TR-18, University of Texas at Austin, 1973.

46. L. Robinson and R. C. Holt. "Formal Specifications for Solutions to Synchronization Problems". Computer Science Group, Stanford Research Institute, Menlo Park, Calif.

47. L. Robinson and K. N. Levitt. "Proof Techniques for Hierarchically Structured Programs". *Comm. of the ACM, Vol. 20*, No. 4 (April 1977), pp 271-283.

48. J. C. Schaffert. "Specifying Meaning in Object-Oriented Languages". Master's Thesis, Dept. of Electrical Engineering and Computer Science, Mass. Inst. of Technology, Cambridge, Mass., forthcoming.

49. M. Shaw, W. A. Wulf and R. L. London. "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators." To appear in *Comm. of the ACM*.

50. D. Teichroew and M. J. Bastarche. "PSL User's Manual". ISDOS Working Paper No. 98, University of Michigan, 1975.

51. B. Wegbreit. "Verifying Program Performance". *J. of the ACM, Vol. 23*, No. 4 (October 1976), pp. 691-699.

52. N. Wirth. "The Programming Language, PASCAL". *Acta Informatica, Vol. 1*, No. 1 (1971), pp. 35-63.

53. W. A. Wulf, R. L. London and M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Trans. on Software Engineering, Vol. SE-2*, No. 4 (December 1976), pp. 253-265.

54. S. Zilles. "Algebraic Specification of Data Types". *Progress Report XI*, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., pp. 52-56; also Computation Structures Group Memo 119, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1974.