A Fortran Programming Methodology Based on Data Abstraction

John F. Isner
National Geodetic Survey

SUMMARY: Data abstraction has been an important consideration since the mid-1970s, with most research effort directed toward the development of experimental languages, formal specification techniques, and program verification schemes. The role of data abstraction in programming methodology, on the other hand, has received considerably less attention. In particular, the potential benefits of the application of data abstraction principles to conventional programming environments have been all but ignored. A programming methodology based on data abstraction and designed especially for the Fortran programming environment is presented here.

1. Introduction

An abstraction is a simplified description of a system that emphasizes the system's important characteristics and ignores those details immaterial to an understanding of the system at a given level. We refer to the abstract description of such systems as their *specification* and to the suppressed details as their *implementation* [12].

The principle of abstraction has played a major role in the evolution of high-level programming languages. Three kinds of abstraction mechanisms are generally recognized.

In *control abstraction*, the implementation of a control statement is suppressed and the specification of its effect presented abstractly—for example, by a flowchart. An example of control abstraction in Fortran is the IF... THEN... ELSE... ENDIF construct. (All references to Fortran assume the 1977 ANSI standard [1] unless otherwise noted.)

In *procedural abstraction*, procedural detail is suppressed by naming a group of statements, giving rise to procedures (Fortran subroutines and functions) and macros (Fortran statement functions). When a program requests SIN(X), it is confident of obtaining the sine of X (this is SIN's specification) without knowing or caring about the procedure used to compute it.

In *data abstraction*, both procedural and representational detail are suppressed so far as they relate to the behavior of a particular class of abstract objects. Formally, a data abstraction is defined as a collection of objects and a collection of operations such that the behavior of the objects can be specified completely in terms of the operations [8]. The definition precludes any mention of the underlying data representation of the objects or the implementation of the operations in characterizing object behavior.

The principle of data abstraction is illustrated by Fortran's built-in data type LOGICAL. LOGICAL values (the set of objects) have an associated set of operations (assignment plus the five operators .NOT., .AND., .OR., .EQV., and .NEQV.), and their behavior can be specified in terms of the operations by means of five truth tables. The truth tables do not reveal how truth values are represented (ones? zeros?) or how the operations are implemented. The behavior of LOGICAL values can be completely specified in this way only as long as language features which allow a programmer to defeat the built-in type system (such as EQUIVALENCEing) are avoided.

Built-in data types enable us to express our programs in the same high-level terms we use in reasoning about abstract objects like logicals, integers, and reals. They free us from a concern with machine-dependent

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques–*modules and interfaces, structured programming*; D.3.2 [**Language Classification**]: Fortran; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs–*specification techniques.*
General Terms: Design. Additional Key Words and Phrases: Programming methodology, data abstraction, state-machine specification technique, Parnas modules, information hiding.
Author's present address: J.F. Isner, National Geodetic Survey C16, National Ocean Survey, NOAA, 6001 Executive Boulevard, Rockville, MD 20852.

details which are immaterial to understanding the behavior of those objects. Herein lies the primary importance of data abstraction in programming methodology: It elevates our concerns from the level of implementation to the level of specifiable object behavior, thereby helping us to manage larger and intellectually more complex problems.

The role of data abstraction need not be limited to the built-in types. An example of a data abstraction that is not provided as a built-in type by any major high-level language is the *stack*. The stack abstraction consists of a set of objects (stacks) and a set of operations (empty stack creation, PUSH, POP, TOP, and EMPTY). The behavior of stacks is completely specifiable in terms of stack operations, as the following specification (following Guttag [3]) demonstrates for stacks of integers. Let S be a stack and I an integer. Then,

EMPTY(CREATE) = true
EMPTY(PUSH(S, I)) = false
TOP(CREATE) = undefined
TOP(PUSH(S, I)) = I
POP(PUSH(S, I)) = S
POP(CREATE) = CREATE

This specification makes no mention of an underlying stack representation (linked storage? contiguous storage?) or the implementation of the stack operations (linked list operations? array operations?).

The lack of a stack data type is not generally thought of as a disadvantage by most Fortran programmers. After all, the argument goes, Fortran provides a basic set of built-in types from which "data structures" representing abstract objects like stacks can easily be built. For obvious reasons, however, data structures and data abstractions are not the same. Failure to appreciate the distinction has led to the production of untold quantities of low-quality, high-cost software. This brings us to the second reason for the methodological importance of data abstraction, first recognized by D. L. Parnas [10]. Parnas argued that the cost of developing and maintaining a system depends on how well "design decisions" are localized, or hidden, from the rest of the system. In a system with *distributed* design decisions, knowledge of the important data structures is shared among the system's procedures. Such systems are characterized by global data structures and a lack of uniformity in the way those data structures are accessed. As an alternative, Parnas proposed *information hiding* as the criterion for decomposing systems into modules. Applied to data abstraction, this criterion would dicate that any data structures representing objects like stacks be packaged together with the operations which access them. It is interesting to note that design methodologies based on purely procedural criteria produce exactly the opposite result. By requiring each of the operations to be realized as a distinct procedure, they actually distribute design decisions and necessitate the use of global data structures.

This article proposes a programming methodology based on data abstraction designed especially for the Fortran programming environment. In the proposed methodology, a system is developed in three stages:

(1) The *design stage* produces the system decomposition. It follows the classic stepwise refinement model for the procedural aspect of the problem but provides for data abstractions. The product of the design stage is a set of informal specifications—one for each procedure or data abstraction identified in the decomposition.

(2) The *specification stage* is needed to refine our understanding of object behavior to a degree necessary for implementation of the data abstractions. Its product is a formal specification for each data abstraction identified in the design stage.

(3) In the *implementation stage*, the procedures and data abstractions are implemented. For the data abstractions, implementation may be regarded as a completely new and independent problem, calling for a reapplication of the entire methodology.

2. The Design Stage

In pure stepwise refinement [13], or the top-down approach, one begins by writing a short, high-level procedure which solves the given problem and then recursively elaborates each of its steps in terms of still lower level procedures. The resulting system decomposition is a collection of procedures.

As an alternative, we will begin by considering the given problem and asking "What data abstractions would be useful in solving this problem?" The kind of abstractions we would seek are not usually provided as built-in types; nor are they familiar abstractions like stacks. Typically, they are highly specialized abstractions that may be unlike anything we have previously encountered. Because they are so unfamiliar, our notion of object behavior is at first extremely ill-defined. The first step toward clarifying these concepts is to attach a name to each abstraction identified.

Having identified and named the primary abstractions, we next write a short procedure which solves the stated problem by operating on abstract objects. Operations are freely devised as needed and each new operation is named and added to the set of operations of the respective abstraction. As an abstraction acquires operations, we gain a clearer notion of object behavior in terms of the operations without giving any consideration to possible data representations for the objects or implementation of the operations. Stepwise refinement is applied to the procedure, yielding more and more procedural detail, but it is not applied to the abstract operations nor is refinement applied to the objects themselves. In the end, each type of object together with its collection of operations constitutes the preliminary design for a single data abstraction.

The remainder of this section is devoted to an example of the design process. The example is chosen from electrical engineering, a discipline in which Fortran is widely used. No prior knowledge of electrical circuit

theory is necessary to understand the example.

2.1 Example

The ABC company is a "low-technology" company, manufacturing devices based on pure resistant circuits. Our job is to provide ABC's engineering staff with software for computer-aided circuit design. The software must allow an arbitrary number of circuits to be described, modified, and analyzed concurrently. Finally, it must permit a user to work on the same circuit during several interactive sessions.

We begin by identifying the principal data abstractions. The following paragraphs attempt to describe and rationalize our choice of one particular abstraction, the *circuit diagram*.

ABC's engineers usually work with circuit diagrams such as the one shown in Figure 1. A circuit diagram consists of a set of *nodes* and a set of *circuit elements* joining the nodes. There are two types of circuit elements: the *battery* and the *resistor*. The *characteristics* of a circuit element are its *electromotive force* (its ability to impart energy to electrons, measured in *volts*), and its *resistance*, measured in *ohms*. A resistor has an electromotive force of zero. In the circuit represented by Figure 1, the battery's electromotive force imparts energy to electrons, causing a *current* of them to flow in the direction indicated by the arrow. The current divides among the circuit elements in a ratio determined by their resistance values (the higher the resistance, the more resistant an element is to a flow of current through it). When current I flows through a resistor with resistance R, the resistor consumes energy in an amount given by $I*R$, also measured in volts. In a real circuit, a *voltmeter* can be used to measure the energy (or voltage) difference between any two nodes of a circuit. It is precisely such voltage values that the ABC engineers need in order to perform their analyses.

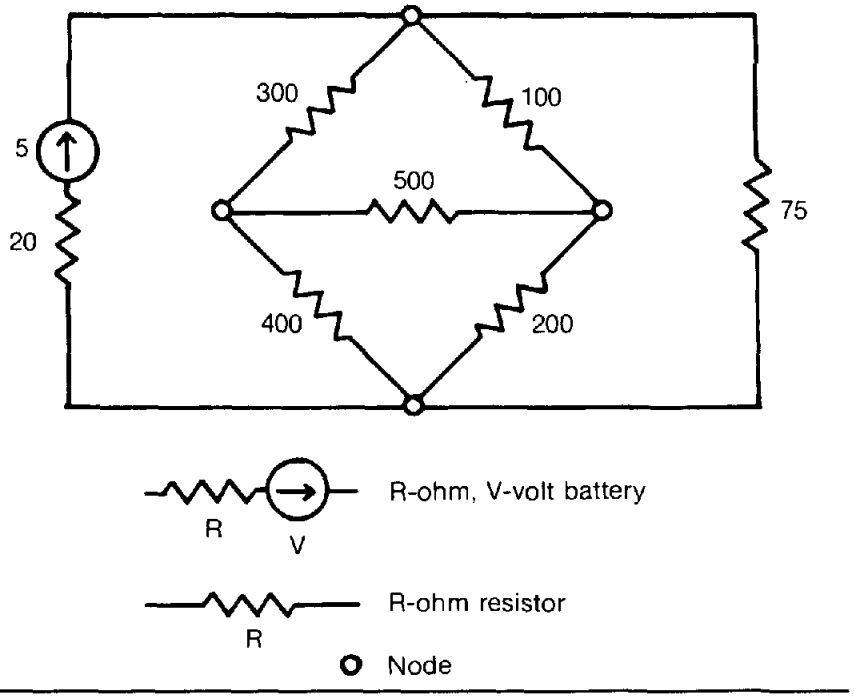

Fig. 1. A Circuit Diagram.

Circuit diagrams like the one in Figure 1 are useful in circuit analysis because they emphasize the important characteristics of a circuit—namely, its topology and the ideal characteristics of each of its elements—and ignore those details of real circuits which are irrelevant to understanding circuit behavior at a certain level. By definition, then, a circuit diagram is an abstraction. Our knowledge of circuit diagrams and their usefulness in circuit analysis provides a conceptual starting point for developing a specialized data abstraction for the ABC problem. Our first step toward making the abstraction more concrete will be to name it; we choose the name "Circuit Diagram data abstraction" (hereafter abbreviated as the "CD abstraction" or, simply, "CD").

Having identified and named the principal data abstraction, we proceed with the design by sketching a very high-level procedure to solve the stated problem in terms of operations on CD objects. As this procedure is refined, our understanding of CD object behavior will improve as new operations are invented and added to the CD operation set. Additional data abstractions will undoubtedly be discovered along the way.

Space does not permit a full development of the ABC design. Instead, we focus on the CD abstraction and summarize the thinking that led to a particular choice of CD operations.

(1) For describing circuits, operations for circuit construction and modification are needed. Because the CD abstraction is to be incorporated in an interactive process, these operations should be incremental in nature. That is, there should be an operation for creating an initial circuit, an operation for adding a single circuit element, and an operation for removing a single circuit element.

(2) A "readout" operation is needed to obtain voltage differences.

(3) Because voltage readings may be taken across any pair of nodes, the abstraction must incorporate a node numbering scheme.

(4) An element numbering scheme is also needed to distinguish

X := NEWCD(N)
Creates a new CD object *X* with *N* nodes. The nodes are initially unconnected and implicitly numbered 1 through *N*.

E := ADDELT(X, I, J, R, V)
Inserts an element with characteristics (*R*, *V*) between nodes *I* and *J* of CD object *X* (a positive *V* is a voltage rise from *I* toward *J*). Assigns element number *E* to the element.

REMELT(X, E)
Removes the element with element number *E* from CD object *X*.

V := VOLTS(X, I, J)
Measures the algebraic voltage difference between nodes *I* and *J* of CD object *X* (a net voltage rise from *I* toward *J* gives a positive reading).

N := NODES(X)
Gives the number of nodes in CD object *X*.

(I, J, R, V) := ELT(X, E)
Gives the characteristics and location of element number *E* in CD object *X*.

N:= MAXNUM(X)
Gives the maximum element number assigned in CD object *X*.

Fig. 2. Informal Specification of CD Operations.

among several elements connected between the same two nodes, as are the battery and the 75-ohm resistor in Figure 1.

(5) Operations for extracting information about previously defined circuit elements and their arrangement are needed for the graphic display.

(6) Because the same program execution may operate concurrently on several CD objects, each operation must have an argument identifying the object being referenced.

(7) To enable the user to interrupt work on a circuit and resume work in a later session, an interface to the file system and a means of storing and retrieving CD objects must be provided.

An informal specification of the CD operations is given in Figure 2. Each entry indicates how the operation might be invoked (a non-Fortran syntax is used for value-returning operations to distinguish arguments which are read-only from function results). The figure omits the input/output operations. These will be discussed in a later section.

Actual circuits obey a *current law*, which requires that the algebraic sum of currents leaving each node be zero at equilibrium. When a circuit is modified (by the addition of a resistor, for example), currents change everywhere in the circuit to attain their new equilibrium values. This change is observable in the form of a new set of voltmeter readings. In the CD abstraction, the VOLTS operation may be requested at any time. Clearly, if it is "called" before an ADDELT operation and then again afterward, its value is likely to change. More generally, each of the circuit modification operations may have an effect on the result of the VOLTS operation. The nature of this cause-effect relationship is not captured by the informal specification of Figure 2.

Figure 2 gives a sense of "normal" CD behavior but fails to convey any feeling for how a CD object behaves under an exceptional sequence of operations. This is natural because the specification was developed with the normal case in mind. What happens when a circuit element with zero resistance is added to the circuit? What reading is obtained by VOLTS when a circuit contains no battery? What if a circuit consists of two or more unconnected parts? If the CD abstraction were implemented directly from Figure 2, the answers to these and similar questions would be found one by one, as the special cases were encountered. Often, the answers given by the implementation would clash with physical reality, requiring program "fixes" which degrade program quality.

In contrast, formal specifications of data abstractions clearly show the interactions among operations. Furthermore, the discipline involved in writing formal specifications refines our understanding of object behavior to a degree necessary for implementation by forcing us to consider both normal and exceptional cases. A formal specification may be used to answer any pertinent questions about object behavior *before* the abstraction is implemented, but should reveal or suggest nothing about implementation, leaving the implementor wide latitude in the choice of implementation data structures and algorithms. For these reasons, formal specifications must always be written. A specification technique is described in Section 4.

3. Implementation Issues

Both the language of implementation and the method of implementation within that language dictate the semantics of the abstract operations to some extent. Furthermore, the adequacy of the mechanisms used to protect objects from unauthorized access ultimately determines the validity of the formal specification. We cannot therefore formally specify the operations without first examining the implementation issues which affect their definition. In this section we consider how "objects" can be created, referenced, destroyed, and protected in a language with no built-in mechanisms for this purpose. We also consider the basic principles involved in input/output. Actual implementation techniques are given in Section 5.

3.1 Object Referencing

In Section 1, Fortran's built-in data type LOGICAL was briefly considered as an example of an abstract data type. From this viewpoint, two LOGICAL "objects" are created

by declaring two logical variables and assigning values to them, as follows:

```
LOGICAL L1, L2
:
L1 = .TRUE.          (1)
L2 = .FALSE.
```

The objects exist from the moment program execution begins (Fortran variables are static), and therefore require no explicit create operation. L1 and L2 may be thought of as *containing* their objects. The assignment L1 = L2 transfers a copy of L2's object into L1, destroying L1's object.

In contrast to (1), we propose the following Fortran statements to create two objects of the programmer-defined CD abstraction:

```
INTEGER CD1, CD2, NEWCD
:
CD1 = NEWCD(50)      (2)
CD2 = NEWCD(75)
```

In (2), CD1 and CD2 are INTEGER variables which receive their values from the INTEGER function NEWCD. Creation of the CD objects is performed explicitly at the time of the calls to NEWCD. Furthermore, CD1 and CD2 do not contain CD objects; they contain unspecified, yet unique, values which *refer* to CD objects. The assignment CD2=CD1 therefore causes CD2 to refer to the same object as CD1, and destroys the object reference previously contained by CD2 (but not the object itself). Subsequent operations on either CD1 or CD2 will therefore affect the same object. The programmer must be aware of the semantics of object reference and exercise caution in handling variables like CD1 and CD2. The differences between (1) and (2) are illustrated in Figure 3.

We will adopt the word "capability" to describe the kind of value returned by object-creating functions like NEWCD. In computer operating systems theory, the view of a system as a collection of objects may be facilitated through "capability-based addressing" [2]. In such systems, each object has a unique value associated with it, called a "capability." A capability is created by the system at the time an object is created. To reference an object, a program must be able to supply a copy of the object's capability. Capabilities must be protected from destruction or alteration (this is often done in hardware). By analogy to such systems, we will refer to the contents of variables like CD1 and CD2 as "capabilities" only when those values have been assigned by object-creating functions like NEWCD. The idea of using a capability-based referencing scheme for objects of an abstract data type was first proposed by Jones and Liskov [5].

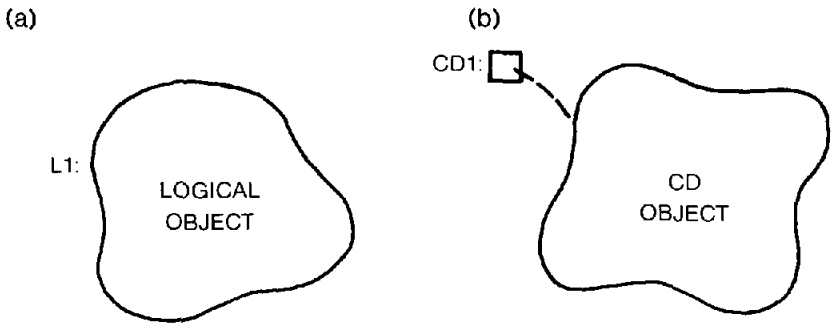

Fig. 3. Differences in the Semantics of Object Reference. (a) LOGICAL variable (b) CD object reference variable.

3.2 Encapsulation

By definition, the set of specified operations belonging to a data abstraction must be the only means of affecting or observing the behavior of objects of the abstraction. If other means are available, the specification can no longer be depended upon to predict object behavior correctly. It is therefore essential that objects of a data abstraction be protected from unauthorized access, intentional or otherwise.

One way of doing this is to incorporate data abstraction facilities in a programming language and rely on the compiler to provide the necessary protection through compile-time type checking. This has been done in recent programming languages like Clu [8] and Ada [4]. In a methodological (as opposed to language-design) approach, we must identify constructs within existing languages which provide the necessary protection. We seek language constructs that "encapsulate" (by means of scope rules) the variables making up the underlying object representation as well as the procedures implementing the operations that access the representation variables. The only Fortran construct providing absolute encapsulation is the procedure (external subroutine or function) with multiple entry points. We will call such procedures "modules," as suggested by D. L. Parnas [10]. A single module implements a single data abstraction. Each entry point of a module corresponds to a single operation of the abstraction. A module may also contain local declarations and local procedures (some versions of Fortran permit internal procedures, which are useful for this purpose). By Fortran's scope rules, these internal elements are inaccessible from outside the module.

Figure 4 schematically illustrates a module implementing the CD abstraction. Corresponding to each operation is an entry point of the same name, at which some sequence of instructions, denoted P_i, is executed. Each of the P_i is free to access the private data structure used as the underlying representation of CD objects (denoted "REP" in Figure 4) as

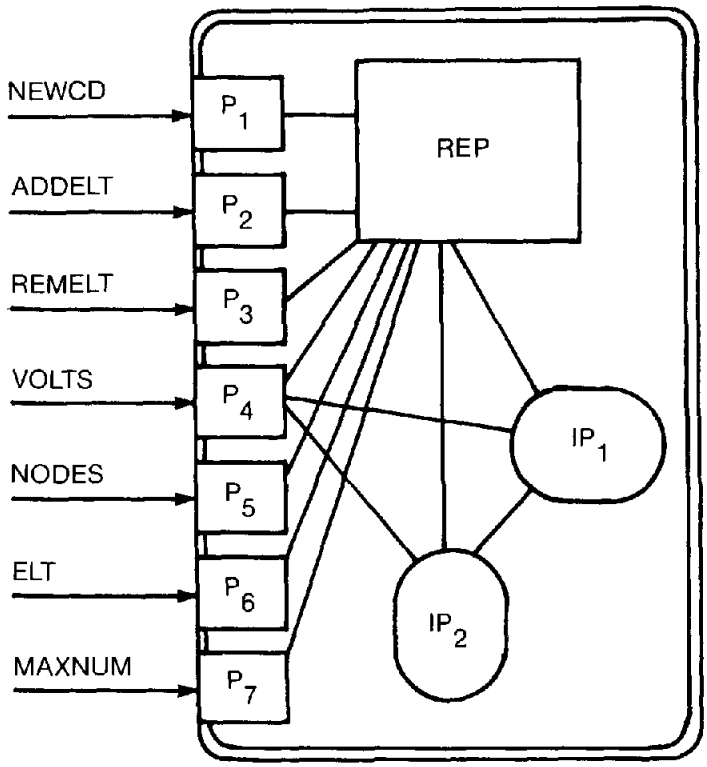

Fig. 4. Schematic View of CD Module.

well as the internal procedures, denoted IP_1 and IP_2. The IP_i may access the REP, and they may call one another. There is, however, no access from outside the module to the P_i, the IP_i, or the REP. Encapsulation is therefore complete.

Encapsulation provides only partial protection against unauthorized access. Suppose that a program requests an operation of the CD abstraction without first calling NEWCD. Using the terminology of Section 3.1, we would say that such a program does not possess a capability for a CD object. The result of such an operation is undefined since any initialization performed by NEWCD will not have been done. This kind of misuse can be prevented by having each operation perform a run-time check as its first action. To allow an application program to perform a similar check, we provide a new CD operation, CAPCD:

L:=CAPCD(X)
Returns.TRUE. if its argument is a capability for a CD object.

3.3 Input/Output

The typical large Fortran-based system is a collection of programs which communicate by means of information stored in files. In systems designed using our methodology, programs communicate by means of abstract object representations stored in files. The third major implementation issue concerns the *external representation* of objects in files and the nature of object movement between main memory and files.

If an object is to be allowed to exist outside of main memory, then protection must be extended to include the file system: No access to an object stored in a file must be possible except by means of operations of the module providing the objects. This rule implies that GET and PUT operations must be included in each data abstraction. If this is done, the external representation of objects is no longer an issue; it is only necessary that GET and PUT observe the same conventions with respect to the chosen representation. For example: if PUT encrypts, then GET must decrypt; if PUT writes the REP variables in a certain order, then GET must read them back in that order.

Given this requirement, we may add the following input/output operations to the informal specification of Figure 2:

PUTCD(X,U)
Puts an external representation of CD object X into the next consecutive locations of the file with Fortran unit number U and leaves the file positioned at the end of file.

X:=GETCD(U)
Gets an external representation of a CD object beginning at the current position in the file with Fortran unit number U, reconstructs the object, and returns a capability for it. Leaves the file positioned at the end of the external representation.

We will return to the subject of input/output in Section 5 in which a somewhat more versatile scheme is suggested.

4. Formal Specification

A number of techniques exist for specifying data abstractions. These differ in their degree of constructability, comprehensibility, range of applicability, extensibility, minimality, and formality (see [7] for a survey of techniques and a comparison based on these criteria).

A formal specification employs a mathematical formalism, in contrast to an informal specification, which is communicated in natural language. Extremely formal techniques are necessary if automatic program verification is the goal. This explains the current interest in such techniques. Unfortunately, for the practitioner, the more formal a technique, the less comprehensible and constructible it is. At the opposite extreme, informal specifications (such as Figure 2) may seem easy to construct and comprehend but present all the pitfalls of natural language communication.

The so-called "state-machine" technique developed by D. L. Parnas [9] and extended by others [11] offers a moderate degree of formality without sacrificing constructibility, comprehensibility, or minimality. It is therefore an ideal technique for the typical Fortran programming environment, where specifications serve mostly to communicate ideas among designers, programmers, and mathematicians, and where the parties involved have at least a minimum degree of mathematical maturity.

Parnas' idea was to separate the operations of a data abstraction into two groups and to regard the objects as *machines* capable of existing in a (not necessarily finite) number of states. Each operation in the "O" group causes a machine to undergo a state change. Each operation in the "V" group causes no state change but allows some aspect of the current state to be observed. The state of an object (or at least the externally visible component of its state) is simply the collective results of all V-operations. The specification, then, need only indicate the effect of each O-operation on the result of each V-operation. Such effects are stated as predicates in the first-order predicate calculus.

In the CD abstraction, the V-operations are VOLTS, NODES, ELT, MAXNUM, and CAPCD. Unlike the other V-operations, CAPCD can be regarded as an inquiry directed at the "manager" of CD objects rather than as an aspect of the state of some particular object.

The O-operations are NEWCD, ADDELT, and REMELT. The set of O-operations has been expanded by adding ZAPCD, an operation for disposing of CD objects which are no longer needed.

A complete state-machine specification of the CD data abstraction is given in the Appendix. We begin here with a description of the technique and conclude with a discussion of the specification itself.

A specification consists of three sections. The OPERATIONS section is the heart of the specification and contains an entry for each operation. The DECLARATIONS section gives the Fortran data type of each of the variables appearing as a parameter or function result in the OPERATIONS section. The DEFINITIONS are macros whose sole purpose is to shorten and improve the readability of the OPERATIONS section. Identifiers for variables and operations are composed of capital letters, as in Fortran. Macro names are written in italics.

The entry for an individual operation begins with a heading which tells what kind of operation it is (O or V) and gives the semantics of its procedure call and return.

A PURPOSE is given for readability only and corresponds to the informal specifications of Figure 2.

EFFECTS are given only for the O-operations. The EFFECTS of an operation consist of a single predicate in the first-order predicate calculus. To improve readability, a predicate may be split into several lines, which are then understood to be "anded" together. Such predicates express the effect of an O-operation on the results of the V-operations and are therefore the heart of a state-machine specification.

The EXCEPTIONS of an operation consist of a numbered sequence of predicates which are evaluated in the order listed. The first predicate evaluating to "true" causes the operation to terminate without returning a value (in the case of *V*-operations) and without causing a state change (in the case of O-operations). If all predicates in the sequence are false, the operation terminates normally.

The meaning of most of the symbols used in writing EFFECTS and EXCEPTIONS can be found in standard logic textbooks such as [6]. A nonstandard symbol is the apostrophe. When the name of a V-operation is prefixed by an apostrophe within a predicate in the EFFECTS or EXCEPTIONS section of an O-operation, its value refers to the old state—that is, the state existing prior to the current set of effects.

INITIAL VALUES are required only for the CAPCD operation to insure that prior to any calls to NEWCD, CAPCD is false for all arguments (since capabilities for CD objects are created only by NEWCD).

Let us now examine the CD specification in more detail. NEWCD will fail only if it is asked to create a CD object with a nonpositive number of nodes. Otherwise, it returns a capability for a CD object in an initial state characterized by the predicate

$$\begin{aligned}&\text{CAPCD}(X)\\&\text{NODES}(X) = N\\&(i)[\text{ELT}(X, i) = (0, 0, 0, 0)]\\&\text{MAXNUM}(X) = 0\\&\textit{equilibrium}(X)\end{aligned}$$

That is to say: CAPCD is true for argument X; the NODES operation has the value N: the MAXNUM operation has the value zero; the ELT operation has the value (0, 0, 0, 0) for all values of its second argument; and "*equilibrium*(X)." The identifier *equilibrium* is recognized as a macro name. Examining the DEFINITIONS section, the fully expanded *equilibrium* macro is seen to be a predicate expressing a complex relationship among the results of the ELT and VOLTS operations. It is precisely this relationship which captures, in purely symbolic terms, the additive properties of voltages and the implications of the current law. In the initial state brought about by NEWCD, it is easy to verify that the *equilibrium* predicate gives

$$\text{VOLTS}(X, i, j) = \begin{cases} 0 & i = j \\ -\text{VOLTS}(X, j, i) & i \neq j \end{cases}$$

Any of the V operations may be called in the initial state. VOLTS, however, is restricted (by its fourth exception) to measuring the voltage difference between any point and itself. This is consistent with the *equilibrium* predicate evaluated in the initial state.

To leave the initial state, an O-operation is needed. REMELT cannot be requested because the CD

object has no elements. This is stated by REMELT's second exception, which can be read "the value of ELT(X, E) in the old state is (0, 0, 0, 0)." Since the initial state is characterized by NEWCD's effect, (i) [ELT(X, i) = (0, 0, 0, 0)], this exception would be raised if REMELT were called.

Instead, we leave the initial state by calling ADDELT. From the EFFECTS of ADDELT, it is apparent that the state of CD object X after the operation is characterized by a value of MAXNUM that is one greater than its value in the old state, a value of (I, J, R, V) for the operation ELT(X, MAXNUM(X)), and *equilibrium*(X). The reader may verify that ADDELT(X, 1, 2, 20, 5) (which adds the battery of Figure 1) causes the fourth term of the *equilibrium* predicate to yield the new equality VOLTS(X, 1, 2) = 5, precisely the effect desired. It would be instructive for the reader to instantiate *equilibrium* for the circuit diagram of Figure 1, assuming some particular sequence of ADDELT operations.

To be perfectly correct, a specification should state that V-operation values not explicitly mentioned in an EFFECTS section do not change. The EFFECTS of ADDELT should therefore be written as:

MAXNUM(X) = 'MAXNUM(X) + 1
ELT(X, MAXNUM(X)) = (I, J, R, V)
(i)[i≠MAXNUM(X) ⊃ ELT(X, i)
= 'ELT(X, i)]
CAPCD(X) = 'CAPCD(X)
NODES(X) = 'NODES(X)
equilibrium(X)

We will continue to use the shorter form for the sake of readability with the implicit understanding that V-operations whose results are not characterized by an EFFECTS predicate remain constant.

A reader planning to use the state-machine specification technique should be warned that the Appendix does not illustrate several of its fine points. In particular, *hidden* and *derived* V-operations are not illustrated. Examples using both kinds of operation can be found in [11].

5. Implementation Techniques

5.1 Single, Multiple, and Dynamic Object Implementations

The Fortran implementor must decide whether a module will provide more than one object. In the ABC application, multiple objects are clearly required. Had the problem been to "perform a series of circuit analyses," a single-object implementation would have sufficed; the single object could be effectively reused by calling ZAPCD and then NEWCD.

When only a single object is needed, a simple solution is to declare as ordinary program variables the data structure to be used as the underlying object representation (the REP of Figure 4). Figure 5 presents the skeleton of a Fortran function implementing a single-object version of the CD abstraction. For illustra-

```
      INTEGER FUNCTION NEWCD(N)
      INTEGER N, X, I, J, E, U, TCODE, NODES, MAXNUM, ADDELT, MAXOBJ,
     * NUMOBJ/0/
      REAL R, V, VOLTS
      LOGICAL CAPCD
      PARAMETER (TCODE=12345,MAXOBJ=1)
      · other declarations
      .
      .
C  THE REP
      INTEGER A, B, TYPE
      REAL C(100)
          IF(NUMOBJ.EQ.MAXOBJ)STOP 1
          IF(N.LE.0.OR.N.GT.10)STOP 2
          NUMOBJ = NUMOBJ + 1
          A = N
          B = 0
          TYPE = TCODE
          NEWCD = TCODE
          RETURN
      ENTRY CAPCD(X)
          CAPCD = X.EQ.TYPE
          RETURN
      ENTRY ZAPCD(X)
          IF(X.NE.TYPE)STOP 3
          NUMOBJ = NUMOBJ - 1
          X = 0
          RETURN
      ENTRY NODES(X)
          IF(X.NE.TYPE)STOP 4
          NODES = A
          RETURN
      ENTRY MAXNUM(X)
          IF(X.NE.TYPE)STOP 5
          MAXNUM = B
          RETURN
      ENTRY VOLTS(X, I, J)
          .
          .
      ENTRY ELT(X, I, J, R, V, E)
          .
          .
      ENTRY ADDELT(X, I, J, R, V)
          .
          .
      ENTRY REMELT(X, E)
          .
          .
      ENTRY PUTCD(X, U)
          .
          .
      ENTRY GETCD(U)
          .
          .
      END
```

Fig. 5. CD Module (Single Object Implementation).

```
      INTEGER FUNCTON NEWCD(N)
      INTEGER N, X, I, J, E, U, TCODE, NODES, MAXNUM, ADDELT, MAXOBJ,
     * NUMOBJ/O/,GETCD, GETSE
      REAL R, V, VOLTS
      LOGICAL CAPCD
      PARAMETER (TCODE=12345, MAXOBJ=1)
      .
      . other declarations
      .
C THE REP
      INTEGER A, B, SE, TYPE
      REAL C(100)
      .
      . other entry points
      .
      ENTRY PUTCD(X, U)
          IF(X.NE.TYPE)STOP 6
          .
          . transfer the variables A, B, C, and TYPE to unit U
          .
          CALL PUTSE(SE,U)
          RETURN
      ENTRY GETCD(U)
          IF(NUMOBJ.EQ.MAXOBJ)STOP 7
          .
          . transfer the variables A, B, C, and TYPE from unit U
          .
          IF(TYPE.NE.TCODE)STOP 8
          SE = GETSE(U)
          GETCD =TCODE
          NUMOBJ = NUMOBJ + 1
          RETURN
      .
      .
      END
```

Fig. 6. CD Module (Multiple Dynamic Object Implementation).

tive purposes only, the CD REP is assumed to consist of the variables *A*, *B*, and *C*.

Although a single-object implementation is simple, it has several disadvantages. For one, the REP variable dimensions are fixed by the storage requirements of the largest problem anticipated. Suppose that array *C* requires n^2 reals, where *n* is the number of nodes in a CD object. Then, the CD implementation of Figure 5 can create CD objects with ten nodes or less. This limitation must be documented as an exception in the specification of NEWCD, adding complexity to the specification. If occasional CD objects with 100 nodes must be created, the dimension of *C* will have to be increased to 10,000 even though most of this space will go unused most of the time. For the sake of the specification and lower system overhead, we would prefer an implementation which provided *dynamic* objects; that is, objects whose REP takes up no more storage space than actually required for a given problem.

Modules offering both multiple objects and dynamic objects can be implemented in any of the extended versions of Fortran which provide dynamic storage allocation together with an address function. Dynamic storage allocation of precisely the amount needed can be requested each time a new object is created by NEWCD. References to the acquired storage must be made by means of base-offset addressing (using the address function), but the clarity of the code can be preserved by declaring statement functions that map individual components of the REP onto the acquired storage. This technique is illustrated for the CD module in Figure 6. In the figure, *A*, *B*, and *C* have become statement functions which can be used to reference locations within a dynamically acquired storage block. ALLOC and ADDR are the two nonstandard functions referred to above; ALLOC(M) returns the address of a storage block containing M words, and ADDR returns the address of its argument. Figure 6 assumes word addressability, full-word integers, and full-word reals. (Note: The use of a statement function as the target of an assignment is also nonstandard.)

5.2 Capabilities and Object Protection

Figures 5 and 6 suggest how the CAPCD exception might be enforced and also offer two interpretations of capabilities. In Figure 5, a capability is the "secret" type code 12345. The CAPCD exception is enforced by comparing the value in *X* with this value at the beginning of each operation. In Figure 6, a capability is a displacement between a variable (DUMI or DUMR) and the beginning of a dynamically acquired storage block. Since DUMI and DUMR are local to the CD module, the capability cannot be used by the calling program to gain access to the REP. Enforcement of the CAPCD exception has been accomplished by having the NEWCD operation store a type code in a fixed location in REP storage and each subsequent operation compare the contents of the location referenced by the capability to this code. This technique is illustrated using code 12345 in Figure 6.

5.3 Subsidiary Abstractions

As stated in Section 3, the CD abstraction is implemented by regarding the formal CD specification as the precise statement of a totally new and independent problem. This time, the problem is to provide the specified operations. Again, the problem is attacked in three stages:

(1) Propose a set of "subsidiary" data abstractions deemed useful in implementing the operations. Develop a collection of

```
      INTEGER FUNCTION NEWCD(N)
      INTEGER N, X, I, J, E, TCODE, NODES, MAXNUM, ADDELT,
     * DUMI(1), ALLOC, ADDR
      REAL R, V, VOLTS,
     * DUMR(1)
      LOGICAL CAPCD
      PARAMETER (TCODE=12345)
      EQUIVALENCE (DUMI, DUMR)
      .
      . other declarations
      .
C THE REP
      INTEGER A, B, TYPE
      REAL C
      TYPE(X) = DUMI(X+1)
      A(X) = DUMI(X+2)
      B(X) = DUMI(X+3)
      C(X, I) = DUMR(X+3+I)
          IF(N.LE.0)STOP 1
          NEWCD = ALLOC(N**2+3) - ADDR(DUMI)
          A(NEWCD) = N
          B(NEWCD) = 0
          TYPE(NEWCD) = TCODE
          RETURN
      ENTRY CAPCD(X)
          CAPCD = TYPE(X).EQ.TCODE
          RETURN
      ENTRY NODES(X)
          IF(TYPE(X).NE.TCODE)STOP 2
          NODES = A(X)
          RETURN
      ENTRY MAXNUM(X)
          IF(TYPE(X).NE.TCODE)STOP 3
          MAXNUM = B(X)
          RETURN
      .
      .
      END
```

Fig. 7. Implementation of GETCD and PUTCD Operations.

procedures which implement the operations by manipulating the objects.

(2) Formally specify the subsidiary abstractions identified in (1).

(3) Implement the procedures; implement the subsidiary abstractions not already provided by Fortran as built-in types.

The procedures written in stage 3 must, of course, be made internal to the module being implemented, but the subsidiary abstractions must be implemented as separate, external modules. The function of the new modules is to provide objects for the REP of the CD abstraction. Encapsulation is not violated by making the subsidiary abstraction modules external since encapsulation must protect objects, not types.

Consider the CD abstraction. Surely, the most difficult part of implementation will be guaranteeing the *equilibrium* predicate. The predicate does not suggest an implementation (indeed, it should not), but merely gives a property that any implementation must satisfy. Many different methods of solving circuit problems yield solutions satisfying *equilibrium*; the choice of a method is up to the implementor. In the *nodal* method, a circuit problem is parameterized in terms of *node potential* unknowns, giving rise to a system of simultaneous linear equations. If the solution to these equations is the set of node potentials e_1, e_2, ... e_n (where n is the number of nodes), then the value returned by VOLTS(X, I, J) is simply $e_J - e_I$. If a procedure employing the nodal method were used in implementing the CD module, it might occur to the implementor to propose a "systems of linear equations" data abstraction with an operation for creating new systems, an operation for defining equation coefficients, and an operation for solving. Each CD object would then have a subsidiary "systems of equations" object. In Fortran terms, the CD REP would have an additional integer variable SE containing a capability for a "systems of equations" object.

5.4 Input/Output

An implementation of the input/output operations GETCD and PUTCD described in Section 3.3 is illustrated in Figure 7. The figure assumes that each CD object contains a single "systems of equations" object. It also assumes a single-object implementation.

The scheme employed in Figure 7 works correctly for any number of levels of subsidiary abstraction and any number of calls to GETCD and PUTCD as long as the application program requesting these operations observes strict sequentiality. That is, if a program requests a sequence of PUT operations for objects of the type T_1, T_2, ..., T_n, respectively, then the only way to retrieve the nth object is by first requesting $n - 1$ GET operations of type T_1, T_2, ..., T_{n-1}, respectively. This scheme is not well-suited to the ABC application in which access to any one of the previously saved CD objects may be required at any time.

One way of removing this difficulty is to introduce a new abstraction called the *rep library*. A rep library is associated with a file, but abstracts from the file by suppressing details associated with keeping track of external REP locations and file positions. In a rep library, each external REP has an associated number by which it is known to both the library and the library user.

5.5 Exception Handling

In Section 4, no method of handling exceptions was suggested. We now give Fortran implementations for two alternate methods: preconditions and signalling.

A *precondition* is a condition which must be satisfied at the time an operation is requested. Failure to satisfy a precondition results in program termination, preferably pre-

ceded by an error message. All exceptions are handled as preconditions in Figures 5–7, making use of the STOP *i* statement for program termination.

A more sophisticated scheme allows the calling program to be notified or signalled when the exception is raised. *Signalling* can be implemented by using label constants in the manner of Figure 8, where REMELT's second exception has been implemented by signalling.

Once a choice of methods for handling each exception has been made, the formal specification should be revised in the manner of Figure 9.

```
In the calling program:
  DO   10 E = 1, MAXNUM(X)
       CALL REMELT(X, E, *10)
       .
       .
10   CONTINUE

In the CD Module:
  ENTRY REMELT(X, E, *)
       .
       .
       IF(element E does not exist)RETURN 1
       .
       .
```

Fig. 8. Implementation of the Second REMELT Exception by Signalling.

```
O-OPERATION REMELT(X, E, *)
    PURPOSE: Removes the element with element number E
             from CD object X.
    EFFECTS:
        ELT(X, E) = (0,0,0,0)
        equilibrium (X)
    PRECONDITIONS:
        1. ~'CAPCD(X)
    SIGNALS:
        1. 'ELT(X,E) = (0,0,0,0)
```

Fig. 9. Modified REMELT Specification.

6. Conclusion

In the experience of the author, large Fortran systems developed using our methodology tend to be more reliable than systems developed using smaller conventional decomposition techniques. They also tend to be smaller

By eliminating COMMON storage and localizing design decisions within modules, a low degree of module coupling is achieved. This means that program errors and program modifications have a small radius of effect. The result is high system reliability and ease of maintenance.

Smaller system size results from the phenomenon of *sharing*: Each operation is coded once rather than at the point of each reference. Smaller program library size results from sharing *across* applications, a phenomenon rarely observed nowadays in spite of the proliferation of common procedure libraries. In practice, sharing of subsidiary abstractions like hash tables, rep libraries, and systems of equations will occur most frequently.

Drawbacks of the methodology include decreased program efficiency (e.g., run-time type checks), the necessity of using a relatively dangerous construct to achieve encapsulation (the Fortran multiple entry point procedure), and the inevitable reliance on nonstandard language features to implement nontrivial abstractions.

It is our strong belief that in an era of rising software costs, the advantages of a programming methodology based on data abstraction far outweigh its disadvantages. We further believe that the methodology proposed here is not only a viable alternative to traditional methodologies in a conventional environment, but is one which brings us more closely into line with the direction of current developments in programming languages and systems.

References

1. American National Standards Institute. Programming Language Fortran, ANSI X3.9-1978.

2. Fabry, R.S. Capability-based addressing. *Comm. ACM 17*, 7 (July 1974), 403–412. Describes the use of capabilities for addressing objects in an object-oriented operating system.

3. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. *Comm. ACM 21*, 12 (Dec. 1978), 1048–1064. Presents an algebraic technique for specifying abstract data types and shows how to use such specifications in proofs of correctness.

4. Ichbiah, J.D., et al. Preliminary Ada reference manual. *ACM SIGPLAN Notices 14*, 6 (June 1979).

5. Jones, A.K., and Liskov, B. An access control facility for programming languages. Carnegie-Mellon Univ., Pittsburgh, Pa., 120, 1976. Proposes a capability-based system for defining types and objects which facilitates compile-time checking of access rights in languages which support user-defined abstract data types.

6. Korfhage, R.R. *Logic and Algorithms.* John Wiley and Sons, New York, 1966. The logical notation used in this paper has been adopted from this standard undergraduate logic textbook.

7. Liskov, B., and Zilles, S. Specification techniques for data abstractions. *IEEE Trans. Software Eng. SE-1* 1 (March 1975), 7–19. A classic paper describing, illustrating, and comparing five categories of techniques for specifying data abstractions.

8. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564–576. Clu was one of the first languages to provide linguistic support for user-defined abstractions. This paper illustrates Clu's abstraction mechanisms by means of an extended programming example.

9. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM. 15*, 5 (May 1972), 330–336. Introduces the specification technique now widely referred to as the "state machine technique." Gives several examples of varying complexity.

10. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM. 5*, 12 (Dec. 1972), 1053–1058. Compares two designs for a KWIC index system and argues that the design based on information hiding is more maintainable.

11. Robinson, L., Levitt, K.N., Neumann, P.G., and Saxena, A.R. A formal methodology for the design of operating system software. In *Current Trends in Programming Methodology, Vol. I: Software Specification and Design,*

Prentice Hall, Englewood Cliffs, N.J., 1977. Presents a specification language derived from Parnas' technique and uses it to describe a collection of modules in a secure operating system.

12. Shaw, M. The impact of abstraction concerns on modern programming languages. *Proc. IEEE 68*, 9 (Sept. 1980), 1119–1130. Traces the ways in which programming languages have evolved in response to new ideas about how to cope with complexity. Shows how the abstraction principle has played a key role in more recent developments.

13. Wirth, N. Program development by stepwise refinement. *Comm. ACM. 14*, 4 (April 1971), 221–227. Proposes a method for developing a program in a series of procedural refinement steps.

Appendix. Formal Specification of CD Abstraction

DECLARATIONS
INTEGER X,N,E,I,J
REAL R, V
LOGICAL L

OPERATIONS

O-OPERATION X:=NEWCD(N)
PURPOSE: Creates a new CD object X with N nodes. The nodes are initially unconnected and implicitly numbered 1 through N.
EFFECTS:
CAPCD(X)
NODES(X)=N
(i)[ELT(x,i)=(0,0,0,0)]
MAXNUM(X)=0
equilibrium(X)
EXCEPTIONS:
1. N ≤ 0

V-OPERATION L:=CAPCD(X)
PURPOSE: Indicates whether its argument is a capability for a CD object.
INITIAL VALUES
(i)[~CAPCD(i)]

V-OPERATION V:=VOLTS(X,I,J)
PURPOSE: Reads the algebraic voltage difference between nodes I and J of CD object X (a net voltage rise from I to J gives a positive reading).
EXCEPTIONS:
1. ~CAPCD(X)
2. ~*node*(I)
3. ~*node*(J)
4. ~*path*(I,J)

V-OPERATION N:=NODES(X)
PURPOSE: Gives the number of nodes in CD object X.
EXCEPTIONS:
1. ~CAPCD(X)

V-OPERATION (I,J,R,V):=ELT(X, E)
PURPOSE: Gives the characteristics and location of element number E in CD object X.
EXCEPTIONS:
1. ~CAPCD(X)

O-OPERATION E := ADDELT(X, I, J, R, V)
PURPOSE: Inserts an element with characteristics (R, V) between nodes I and J of CD object X (a positive V is a voltage rise from I to J). Assigns element number E to the element.
EFFECTS:
MAXNUM(X)='MAXNUM(X)+1
ELT(X,MAXNUM(X))=(I,J,R,V)
equilibrium(X)
RETURNS:
MAXNUM(X)
EXCEPTIONS:
1. ~'CAPCD(X)
2. ~*node*(I)
3. ~*node*(J)
4. $R \leq 0$

V-OPERATION N:=MAXNUM(X)
PURPOSE: Gives the maximum element number assigned in CD object X.
EXCEPTIONS:
1. ~CAPCD(X)

O-OPERATION REMELT(X, E)
PURPOSE: Removes the element with element number E from CD object X.
EFFECTS:
ELT(X,E)=(0,0,0,0)
equilibrium(X)
EXCEPTIONS:
1. ~'CAPCD(X)
2. 'ELT(X,E)=(0,0,0,0)

O-OPERATION ZAPCD(X)
PURPOSE: Disposes of CD object X.
EFFECTS:
~CAPCD(X)
EXCEPTIONS:
1. ~'CAPCD(X)

DEFINITIONS
node(i) ↔ i ≥ 1 ∧ i ≤ NODES(X)
connection(i, j) ↔ ∃e[(ELT(X, e)=(i, j, r, v) ∨ ELT(X, e)=(j, i, r, v)) ∧ r ≠ 0]
path(i, j) ↔ i=j ∨ *connection*(i, j) ∨ ∃k[*connection*(i,k) *path*(k,j)]
equilibrium(X) ↔
(i)[VOLTS(X,i,i)=0] ∧
(i)(j)[VOLTS(X,i,j)= −VOLTS(X,j,i)] ∧
(i)(j)(k)[(*connection*(i,k) ∧ *path*(k,j)) ⊃
VOLTS(X,i,j)=VOLTS(X,i,k)+
VOLTS(X,k,j)] ∧

$$(i)\left[\; 0 = \sum \begin{array}{l} \text{IF} \\ \quad (\text{ELT}(X,e)=(i,j,r,-v) \wedge r\neq 0) \vee \\ \quad (\text{ELT}(X,e)=(j,i,r,+v) \wedge r\neq 0) \\ \text{THEN} \\ \quad (\text{VOLTS}(X,i,j)+v)/r \\ \text{ELSE} \\ \quad 0 \end{array} \;\right]$$

The attention of Computing Practices readers is called to this month's ACM Forum letter "On Programmer Involvement for Quality Assurance" by Martin Gorfinkel, which refers to the paper by Gustafson and Kerr in the January issue.