rules. Proc. of Sixth Texas Conf. of Comput. Syst., Austin, Tex., Nov. 1977.
18. Palme, J. Protected program modules in SIMULA 67. FOAP Rep. No. C8372-M3(E5), Research Inst. of National Defense, Stockholm, 1973.
19. Parnas, D. L. Information distribution aspects of design methodology. *Information Processing 71*, 1 (1972), North-Holland Pub. Co., Amsterdam, 339–344.
20. Spitzen, J., and Wegbreit, B. The verification and synthesis of data structures. *Acta Informatica 4* (1975), 127–144.
21. Standish, T.A. Data structures: an axiomatic approach. BBN Rep. No. 2639, Bolt Beranek and Newmann, Cambridge, Mass., 1973.
22. Suzuki, N. Automatic verification of programs with complex data structures. Ph.D. Th., Comptr. Sci. Dept., Stanford, U., Rep. No. STAN-CS-76-552, Feb. 1976.
23. Wegbreit, B., and Spitzen, J. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), 389–396.
24. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2*, 4 (December 1976), 253–265.
25. Zilles, S. N. Abstract specifications for data types. IBM Res. Lab., San Jose, Calif., 1975.

Programming Languages
J.J. Horning Editor

An Example of Hierarchical Design and Proof

Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson
SRI International

Hierarchical programming is being increasingly recognized as helpful in the construction of large programs. Users of hierarchical techniques claim or predict substantial increases in productivity and in the reliability of the programs produced. In this paper we describe a formal method for hierarchical program specification, implementation, and proof. We apply this method to a significant list processing problem and also discuss a number of extensions to current programming languages that ease hierarchical program design and proof.

Key Words and Phrases: program verification, specification, data abstraction, software modules, hierarchical structures
CR Categories: 4.0, 4.6, 5.21, 5.24

1. Introduction

The use of structuring techniques in programming—for example, programming by successive refinement [5] (also called hierarchical programming)—has been recognized as increasingly helpful in the design and management of large system efforts. A number of such design techniques are now promoted for routine use in com-

Research supported by the Office of Naval Research (Contract N00014-75-C-0816), the National Science Foundation (Grant DCR74-18661), and the Air Force Office of Scientific Research (Contract F44620-73-C-0068).
Authors' present addresses: J.M. Spitzen, Advanced Systems Department, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304; K.W. Levitt and L. Robinson, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

mercial software development [33]. Some of these techniques are also alleged to permit the verification of large systems by reducing them to a collection of small programs, each easily verified.

Important questions about such hierarchical proofs are:

—Can systems be decomposed into subprograms that can be characterized by clear and natural assertions?
—Can proofs of the subprograms be combined to demonstrate the correctness of the system?
—Is it generally possible to formulate and prove significant implementation-independent properties of systems?

Several recent developments yield positive answers: the hierarchical design and module specification techniques of [24] and the data abstraction techniques of [9] and [26]. (The word "module" is very widely and imprecisely used and the reader should be wary of drawing inferences not based on our very specific use.) A module is the basic unit in a hierarchical decomposition—a collection of operations and data. The module permits the definition of complex abstract types. For example, a type "file" can be defined by a module with operations for creating a file, inserting a record into a file, reading a record, appending two files, etc., and data structures recording the file's contents.

To permit hierarchical proof, one must *formally specify* the modules of a hierarchical system. Styles of specification and their mathematical foundations differ (e.g. consult [8, 24, 27]; also, [15] and [13] are overviews), but the basic aim is to achieve abstract specification, i.e. specification that describes the input-output behavior of a module without recourse to an implementation of the module. This may be done in terms of the sequences of operations that have been performed on the module, or by abstracting from these sequences to a module *state.* In the first of these approaches, one may attempt to describe concisely the infinite class of possible histories by a small number of "algebraic specifications," as in [8]. In this paper, we will use the state approach. An important aspect of a good specification—in any method—is that, for a properly conceived module, it is a concise, intuitive, and precise characterization of the behavior of the module, successfully abstracting from the details of any implementation of the module. It is often possible to formulate and prove important properties of a module in terms of its specifications.

Similarly, both algebraic and state styles of specification lend themselves to two-stage implementations: First, the data structures of a module (other than the most primitive) are represented in terms of lower level data structures (as in [9] and [26]) and, second, *abstract programs* are written for each operation in terms of lower level operations. Each abstract program may then be proved to satisfy its specifications on the assumption that the more primitive modules it employs are correct, given the specific data representation used.

Should descriptions of hierarchical structure, formal specifications of modules, and implementations all be written in a single language? We have chosen to separate these functions, and will describe a powerful specification language, SPECIAL, and a very simple implementation language ILPL. An alternative is to provide abstraction directly in the implementation language—the approach of [4, 11, 14, 31, 32]. A second language issue is what characteristics a language (or, in our view, set of languages) should have to enable the correct implementation of hierarchies of abstractions. For example, what protection principles are needed to ensure the data integrity of a module?

The primary contributions of this paper are a description of the design and proof of a nontrivial, useful program, and a demonstration of a technique that has promise of making proof and formal description possible for large programs. Our example is a program to maintain unique lists with an efficient underlying implementation. We have attempted to address three classes of readers: those who wish to learn about formal specification should be able to do so by following our specifications and the associated prose; those who wish to learn about so-called "functional program verification" will be introduced to this style of proof; and those who are unfamiliar with list processing may obtain an introduction to some relevant techniques.

In the next section, we introduce our design and proof method. Then in Section 3 and Section 4 we present formal specifications of the two modules that comprise the top-level machine of an illustrative hierarchy. Based on these specifications, we are able in Section 5 to prove several properties of this machine. The implementation of this machine and a proof of the correctness of this implementation are presented in Section 6 and Section 7. Section 8 outlines how the hierarchy described up to that point might be refined into an executable program. Finally, Section 9 presents some concluding remarks.

2. Design and Proof Method

Suppose a programming problem P and a machine M are given and it is required to construct a program C that executes on M to solve P. M may be either a physical machine or a virtual machine provided, for example, by the compiler for a particular programming language. We believe that the program construction should proceed as follows.

First, we design an abstract computer AM on which it is easy to solve P. AM is designed by describing its states and executable instructions. We deliberately leave vague the meaning of an "easy solution" of P—for some purposes this will be a solution by a program AC that is only a page or two long; for other purposes it will be a solution by a program that can be mechanically proved correct using state-of-the-art verification systems. Having designed AM, we implement a solution of P on AM,

and, in practice, usually alter the design of AM in the process. If AM is the same as M, then we have satisfied the original requirement if it can be demonstrated that the solution is correct. Otherwise, we have reduced the original requirement to the new one of realizing AM in terms of M. To do this, we design a new machine AM′ on which it is easy to realize AM. We represent the states of the first machine AM in terms of the states of AM′ and we implement the executable instructions of AM by means of a set of programs AC′ on AM′. The choice of *representation* or *implementation* may prove awkward; if so, we resort to altering the design of AM′ or even that of AM. (Unfortunately, we usually cannot alter the original requirement P, though requirements sometimes are changed when a problem is better understood.)

Next, the realization of AM on AM′ must be verified. This verification means that the (partial) solution of P obtained by executing AC on AM is equivalent to the (more complete) solution obtained by executing AC and AC′ together on AM′. In this latter solution, the execution of an instruction of AM by AC is viewed not as primitive but as a call on the subroutine in AC′ that realizes that instruction on AM′. Again, if AM′ is the same as M, we are done and otherwise we must continue to approach M by extending the sequence of machines AM, AM′, ... and programs AC, AC′, ... When we have extended this sequence to a program that executes on M, then the set of programs and subroutines AC, AC′, ... constitutes the required solution C on machine M.

This description of programming has been phrased in the top-down paradigm, but that is not what is important. To make the programming of large problems feasible, reliable, and controllable, they must be somehow divided into small parts. We have no special preference for top-down or bottom-up programming in arriving at this division, and suspect that a flexible mixture of both techniques is required in general. We do advocate the use of formal methods to describe the division, to validate the resulting design, and to prove the correctness of the final program.

The product of our endeavors will thus consist of:

—A hierarchy of abstract machines
—A formal specification of each machine
—A representation of the states of each machine (except the given machine) in terms of the states of the machine below it
—An implementation of the instructions of each machine (except the given machine) in terms of the instructions of the machine below it

We specify an abstract machine using a method originally proposed by Parnas [24] and subsequently extended by Robinson and Levitt [26]. (Our formal specification language, SPECIAL, is described in [28].) A machine has a state and an instruction set. We give the state by describing the initial values of a set of *V-functions*. We give the instructions, called *OV-functions*, by describing how each changes the state of the machine and what value it returns. (The return of a particular value may, for formal purposes, be thought of as part of the change of state.)

A V-function specification consists of a *header* that describes its arguments and result and an *initialization* that describes the values of the function in an initial machine state.

An OV-function specification consists of a *header* that describes its argument structure, an *assertion list* stating preconditions on the calls of the function, an *exception list* describing when its execution may have no effect other than signaling an error, and a set of *effects* that nonprocedurally describe the changed state due to an execution of the function by defining the resulting values of the V-functions of the machine. (These values are described in terms of the old values of the V-functions and the arguments to the OV-function.) The effects include, if appropriate, the designation of the value to be computed and may allow a nondeterministic choice of successor state.

It is usually possible to give additional structure to an abstract machine M by describing it as the "product" of modules M1, M2, ... , M*n*. When we do this, we will refer to the M*i* as submachines or modules of M; otherwise the terms machine and module are used as synonyms. To form such a product, we require that the functions of the M*i* be renamed to avoid conflicts. M has as its V-functions each of the V-functions of the M*i* with the same initial sections. M has as its OV-functions each of the OV-functions of the M*i*, with augmented effects sections. Specifically, if I is an OV-function of M*i* and V is a V-function of M*j*, where $i \sim = j$, we add to the effects of I the assertion that V is not changed by the execution of I.

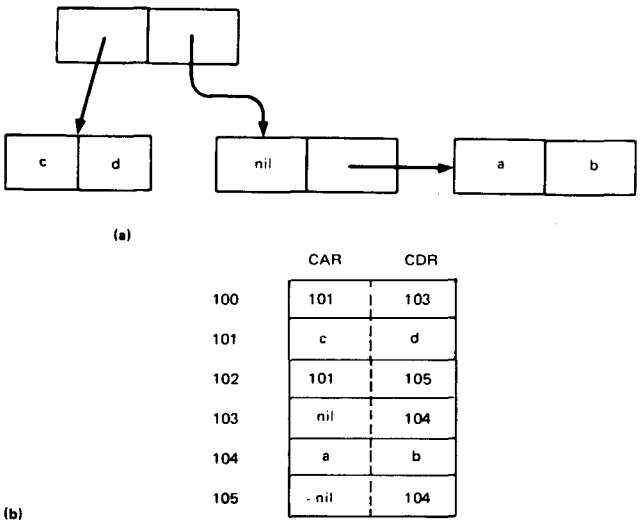
3. The ULIST Module

In this section and the next, we present an abstract machine consisting of two modules: one that maintains conventional list structures and one that maintains a class of *unique lists*—lists such that no two are structurally isomorphic. (Figure 1 illustrates the usual realization of conventional lists where distinct isomorphic lists are possible.) Thus the attempt to construct one of these unique lists yields an old list if there is already one with the right components.[1] Naturally, we want the check of existing cells to be efficient. We use a particularly effective method introduced by Deutsch in his verification system [6] to associate properties with arbitrary symbolic expressions.

We begin by presenting a formal specification of a machine providing unique lists, and explain our notation

[1] As an example of the utility of such a facility, suppose we save the property SIMPLIFIES-TO-ZERO in some table under—as key—the address of the list (SUBTRACT x x). If we subsequently independently create a conventional list of the same form, it will have a different address and the property will not be retrieved. But if both are unique then their addresses will be the same so that the property can be looked up successfully.

Fig. 1. Distinct isomorphic lists. (a)A picture of the list ((*c.d*) *nil a.b*). (b)Two distinct isomorphic versions of the list ((*c.d*) *nil a.b*) at 100 and 102.



by referring to this specification. The state of a ULIST machine is determined by what unique list cells exist. Hence we want a single V-function, UCELL(X1,X2) whose value is the cell with X1 and X2 as components—if there is any such cell—and the distinguished value "?" if there is no such cell. (All that will matter about "?" in this paper is that it does not satisfy the predicate ATOMP to be introduced in Section 3.) There are four instructions on the ULIST machine: UCONS to obtain a list with specified components, UCAR and UCDR to extract the components of a ULIST list, and ULISTP to test whether an arbitrary object is a ULIST list. The specification is given below.

```
module:   ULIST:
  forall:   Z1,Z2
  vfns:     UCELL(X1,X2)
              initial UCELL(X1,X2) = ?
  define:   ISUCELL(X) = (exists X1,X2:UCELL(X1,X2)=X~=?)
  ovfns:    UCONS(X1,X2) → X
              assert  ISUCELL(X1) or ATOMP(X1)
                      ISUCELL(X2) or ATOMP(X2)
              effects if UCELL(X1,X2) = ?
                        then UCELL(Z1,Z2)~=X and
                             'UCELL(X1,X2)=X and
                             X~=? and ~ATOMP(X) and
                             (X1~=Z1 or X2~=Z2
                                    ⇒ 'UCELL(Z1,Z2)
                             =UCELL(Z1,Z2))
                        else 'UCELL(Z1,Z2)=UCELL(Z1,Z2) and
                             X = UCELL(X1,X2)
            UCAR(X) → X1
              assert  ISUCELL(X)
              effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                      UCELL(Z1,Z2)=X ⇒ X1=Z1
            UCDR(X) → X2
              assert  ISUCELL(X)
              effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                      UCELL(Z1,Z2)=X ⇒ X2=Z2
            UCONSP(X) → B
              effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                      B = ISUCELL(X)
```

It will be common, in the specification of this module, to ask the question, "Is the object X a unique list cell?" Therefore we introduce the abbreviation ISUCELL, expressing this predicate in terms of the module's single V-function. Note also that in the specification we use the predicate ATOMP to distinguish the objects that may be the "leaves" of list structures from the objects that are list cells. Rather than implementing this predicate as we develop our hierarchy, we will simply assume that it is present in the most primitive machine of the hierarchy and is reflected upward, with the same meaning, in each nonprimitive machine. In Section 9, we will discuss the significance of this assumption.

In the initial ULIST state, there are to be no list cells. We specify this by requiring that UCELL(Z1,Z2) be "?" initially. (We abbreviate slightly by listing at the head of each module symbols that should be read as universally quantified in all their uses in the module specification; for ULIST these are Z1 and Z2.)

Next, we specify the instruction UCONS(X1,X2) to obtain a cell with components X1 and X2. This instruction has no exceptions: it is required to achieve its effects for any arguments and state; in particular, this requires that any implementation have an unlimited set of cells. (Although this requirement is idealistic, it simplifies our presentation; SPECIAL does provide for the description of *resource errors*.) We assert that the arguments to UCONS are either outputs of UCONS [ISUCELL(X)] or atoms [ATOMP(X)]. Its effects are stated with an "if-then-else" assertion. We need to refer to two sets of values of UCELL—those associated with the state before the UCONS instruction is executed and those reflecting the changed state due to the execution. We will do this by writing UCELL(X1,X2) to refer to the old state and 'UCELL(X1,X2) to refer to the changed state. First, if the machine state is such that UCELL(X1,X2) is "?", then a new cell must be created. We do not choose to specify how cells are represented (e.g. by their integer addresses on some machine), but say only that the new cell is a value X that is not a cell before the execution of this instruction (i.e. UCELL(Z1,Z2)~=X for any X1, X2) and is the cell with the specified components afterwards ['UCELL(X1,X2)=X]. Besides constraining the value of the new cell, we must ensure that no other cells are affected by the instruction. Thus we say that if (Z1,Z2) is any pair of cell components other than the (X1,X2) given in the instruction call, then the new cell associated with (Z1,Z2) is the same as the old ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

If UCELL(X1,X2) is not "?", then the effects of the instruction are simpler. We constrain the result of UCONS to be the existing cell [Z=UCELL(X1,X2)] and require the new state to have exactly the same cells as the old ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

Next, we specify the UCAR and UCDR instructions. Our notion of the ULIST module is that it is not meaningful to ask for either of the components of an object other than a ULIST cell. Hence we assert that the

arguments to UCAR and UCDR are ULIST cells using the predicate ISUCELL which requires that its argument be in the image, under UCELL, of the set of pairs (Z1,Z2). The effects of UCAR and UCDR are similar. If Z1 and Z2 are such that UCELL(Z1,Z2) is the argument to UCAR or UCDR, then the UCAR component of UCELL(Z1,Z2) is Z1 and the UCDR component is Z2. (It is not obvious that such a specification is noncontradictory; this is a consequence of the theorem, proved below, that UCELL is single-valued: it maps distinct arguments to the same result only when that result is "?".) Besides giving the values of these functions, the specification asserts that they have no effect on UCELL ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

The last ULIST instruction is UCONSP. It is like UCAR and UCDR in that UCELL is unchanged. Its result B must be true or false, and true if and only if its argument is a ULIST list. But this is easily stated in terms of the UCELL V-function—it is equivalent to the existence of a pair (Z1,Z2) such that UCELL(Z1,Z2) is equal to the argument X [B = exists Z1,Z2: UCELL(Z1,Z2)=X].

4. The List Module

Our goal is to design an abstract machine that provides its user with both unique and conventional list processing. This machine is the product of ULIST and a module LIST that we will specify next. The formal specification of LIST is given below.

```
module:   LIST:
  forall:   Z1,Z2,Z
  vfns:     CELL(X1,X2,X)
              initial CELL(X1,X2,X)=false
  define:   ISCELL(X) = (exists X1,X2:CELL(X1,X2,X) and X~=?)
  ovfns:    CONS(X1,X2) → X
              assert  ATOMP(X1) or ISCELL(X1)
                      ATOMP(X2) or ISCELL(X2)
              effects 'CELL(X1,X2,X)
                      not CELL(Z1,Z2,X)
                      X~=?
                      not ATOMP(X)
                      Z~=X ⇒ 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
            CAR(X) → X1
              assert  ISCELL(X)
              effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                      exists Z2:CELL(X1,Z2,X)
            CDR(X) → X2
              assert  ISCELL(X)
              effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                      exists Z1:CELL(Z1,X2,X)
            CONSP(X) → B
              effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                      B = ISCELL(X)
```

The structure of this module is quite similar to ULIST: there is a single V-function CELL and four OV-functions CONS, CAR, CDR, and CONSP. However, there are important differences. First, whereas UCELL is a function from a UCAR/UCDR pair (X1,X2) to the unique list cell X—if any—with X1 and X2 as UCAR and UCDR, CELL is a predicate on the triple (X1,X2,X) that tests whether X is a conventional list cell with CAR X1 and CDR X2. This difference is necessary: because there may be more than one conventional list cell with a particular CAR and CDR, CELL cannot be a function. The second difference between the two modules is in the effects of UCONS and CONS. UCONS does not always change the ULIST state, but CONS always changes the LIST state. Even if there are already X1, X2, and X~=? such that CELL(X1,X2,X), an execution of CONS(X1,X2) will create a new cell with this CAR and CDR.

5. Properties of ULIST and LIST

Even though we have not yet implemented ULIST × LIST, we can prove properties of this machine just on the basis of its specifications. We illustrate this point by proving three results: that UCELL is one-to-one, that two structurally isomorphic unique lists are identical, and that if two conventional lists are structurally isomorphic, then certain corresponding unique lists are identical.

Consider the claim that the specification of ULIST is consistent, that is, implementable. For example, if UCELL is not one-to-one on that part of its domain that does not map to "?", then the specifications for UCAR and UCDR are not realizable. For suppose X=UCELL(Z1,Z2)=UCELL(Z3,Z4)~=?. If Z1 is not equal to Z3, then UCAR(X) is required to return both Z1 and Z3, an impossibility. Similarly, if Z2 is different from Z4, then UCDR(X) is required to return two different values.

This result is not, by itself, sufficient to show that ULIST can be implemented. On the other hand, a provably correct implementation of ULIST—given below—implies this result. However, the result is easy to state and has interesting consequences. Moreover, its proof illustrates a general proof technique applicable to abstract machines.

THEOREM 1. forall Z1,Z2,Z3,Z4:
UCELL(Z1,Z2)=
UCELL(Z3,Z4)~=?
⇒ Z1=Z3 and Z2=Z4.

PROOF. The theorem will be proved by induction on sequences of states of ULIST. This method of proving properties of abstract machines, which we call *generator induction*, is discussed in [3, 9, 30]. We must prove the theorem for any initial state of ULIST and for any state S′ such that the theorem holds in a state S and S′ is a state resulting from executing a ULIST instruction in S.

The basis of the induction is immediate, since UCELL is always "?" in an initial machine state. Thus it suffices to assume the theorem holds in some state S and to deduce its validity in a successor state S′ that results from the operation UCONS(X1,X2) (since UCONS is the only operation that changes UCELL's re-

sults). Suppose that this execution of UCONS returns X and that, in the resulting state, there are Z1, Z2, Z3, and Z4 such that 'UCELL(Z1,Z2)='UCELL(Z3,Z4)~=?. If there has been no state change—the "else clause" of the effects—or if 'UCELL(Z1,Z2)=UCELL(Z1,Z2) and 'UCELL(Z3,Z4)=UCELL(Z3,Z4), then the inductive assumption gives the desired result. Suppose that there has been a state change—the "then clause" and that 'UCELL(Z1,Z2)~=UCELL(Z1,Z2). Thus Z1=X1 and Z2=X2. If Z3=X1 and Z4=X2, we are done. If Z3~=X1 or Z4~=X2, then the second equation of the else clause implies that 'UCELL(Z3,Z4)=UCELL(Z3,Z4) which is not equal to X by the first equation of the else clause, a contradiction. This completes the proof.

Next, we extend this result to show that structural equality implies identity for ULIST lists. Let us write UCAR*, UCDR*, and UCONSP* to refer to the values returned by these instructions in some state. (We introduce this notation to emphasize the careful distinction between these values, useful in stating static mathematical properties of a specification, and the instructions that might be executed to obtain them in some implementation.) Theorem 1 implies that, in any state, UCAR* and UCDR* are functions mapping the set of X such that UCONSP*(X) to a range not containing "?". It is a simple matter, using Theorem 1, to show that the conclusion X=Y follows from the hypotheses UCONSP*(X), UCONSP*(Y), UCAR*(X)= UCAR*(Y), and UCDR*(X)=UCDR*(Y); we leave the details to the reader. Using this corollary, we can prove that structural isomorphism implies identity for the lists of the ULIST machine. We define structural isomorphism of unique lists recursively by:

```
UISO(X,Y) ⇐ if UCONSP*(X) and UCONSP*(Y)
               then UISO(UCAR*(X), UCAR*(Y))
                 and UISO(UCDR*(X),UCDR*(Y))
               else X=Y,
```

and can then prove:

THEOREM 2. forall X,Y: UISO(X,Y) ⇒ X=Y.

Discussion. We wish to prove this theorem by structural induction on unique lists. (Structural induction is described by Burstall in [3]; the theorem can also be proved, less easily, by generator induction.) We will prove the theorem for the atoms that form the leaves of a unique list and we will prove that if the theorem holds for the proper sublists of a unique list, then it holds for the entire list. For such an induction to be sound, it is essential that there be no circular lists. Fortunately, the ULIST machine instructions provide no way to create circular lists. Since there are no lists in an initial ULIST state, since each instruction creates at most one new list, and since we are only interested in machine states achievable by the execution of finitely many instructions, this induction is well-founded. (This same argument demonstrates that UISO is total.)

PROOF. The basis of the induction is the case that ~UCONSP*(X) or ~UCONSP*(Y), and it is an immediate consequence of the definition of UISO. We make the inductive assumptions UCONSP*(X), UCONSP*(Y), UISO(UCAR*(X),UCAR*(Y)) ⇒ UCAR*(X)=UCAR*(Y), and UISO(UCDR*(X), UCDR*(Y)) ⇒ UCDR*(X)=UCDR*(Y), and must prove that UISO(X,Y) ⇒ X=Y. Expanding the definition of UISO(X,Y), we conclude that UISO (UCAR*(X),UCAR*(Y)) and UISO(UCDR*(X), UCDR*(Y)); hence by the inductive assumptions, UCAR*(X)=UCAR*(Y) and UCDR*(X)=UCDR*(Y). The corollary of Theorem 1 now yields the desired result.

Next we prove a theorem about a program that might be run by a top-level user of the ULIST × LIST machine to translate conventional lists to unique lists. (Because of the overhead associated with the maintenance of unique lists, it is common to do some computations with the corresponding conventional lists, and convert only the final result to a unique list.) We claim that this may be done by the program UCOPY defined as follows:

```
UCOPY(X) ⇐ if CONSP(X)
              then UCONS(UCOPY(CAR(X)),
                         UCOPY(CDR(X)))
              else X.
```

The major result about UCOPY is that if two conventional lists are isomorphic, then their U-copies are identical. Isomorphism of conventional lists is defined by:

```
ISO(X,Y) ⇐ if CONSP*(X) and CONSP*(Y)
              then ISO(CAR*(X),CAR*(Y))
                and ISO(CDR*(X),CDR*(Y))
              else X=Y.
```

Let UCP(XA,XB) be an abbreviation for ISO(XA,XB) ⇒ UCOPY(XA)=UCOPY(XB). We will then prove:

THEOREM 3. forall XA,XB: UCP(XA,XB)

Discussion. The meaning of this formula is subtle, since the effects and result of UCOPY are contingent upon the machine state in which it is executed. A more precise statement would be as follows. Suppose ISO(XA,XB). Suppose that UCOPY is applied to XA, beginning in some state S1. This application terminates (since our lists are acyclic) yielding a value XA′ and a state S2. Suppose, moreover, that S3 is any successor of S2, resulting from a series of state transitions, starting at S2. Finally, suppose that UCOPY, applied to XB from state S3, yields value XB′ and state S4. Then XA′=XB′.

We are going to prove this result by induction. An inductive assumption of this rather lengthy form would be very cumbersome. Fortunately, the machine specifications imply that if UCONSP*(X) holds in some state S, then it holds in every successor state. Also, if UCAR*(X) and UCDR*(X) are defined in a state S, then they remain defined and retain the same value in all successor states. In view of these facts, sometimes called *frame axioms*, we can safely omit further refer-

ences to changing states and use the simple statement of the theorem. (It is very interesting to consider whether this kind of problem reduction might be done mechanically.)

PROOF. First, suppose ~CONSP*(XA). Then ISO(XA,XB) implies that XA=XB. Also, UCOPY(XA)=XA and UCOPY(XB)=XB so that the desired result is immediate. Next suppose CONSP*(XA). Then ISO(XA,XB) implies that CONSP*(XB). We proceed by simultaneous structural induction on XA and XB. That is, we assume

I1. forall X:UCP(s(XA),X)
I2. forall X:UCP(X,s(XB))

where s is either CAR* or CDR*. (Clearly, UCP is symmetric in its two arguments.) From ISO(XA,XB) it follows that

I3. ISO(Car*(XA),CAR*(XB)), and
I4. ISO(CDR*(XA),CDR*(XB)).

Combining these results with I1 and I2 we obtain

I6. UCOPY(CDR*(XA))=UCOPY(CDR*(XB)).

We must prove that UCOPY(XA)=UCOPY(XB). This is done as follows:

```
UCOPY(XA) = UCONS*(UCOPY(CAR*(XA)),
                   UCOPY(CDR*(XA)))
              {by the definition of UCOPY}
          = UCONS*(UCOPY(CAR*(XB)),
                   UCOPY(CDR*(XB)))
              {by I5, I6}
          = UCOPY(XB)
              {by the definition of UCOPY}.
```

6. Implementation of ULIST × LIST

We now wish to implement the machine specified above in terms of more primitive facilities. Specifically, we will consider a machine, LIST × SEARCH, that has conventional list processing capabilities and an associative search capability. Since we retain the LIST facilities in this second level machine, the main problem is to describe ULIST in terms of associative search and conventional list processing.

Our SEARCH machine is formally specified below.

```
module:  SEARCH:
  forall:  K,T
  vfns:    PRIMARYTABLE( )
             initial PRIMARYTABLE( )~=?
           GET(KEY,TABLE)
             initial GET(KEY,TABLE)=?
           TABLEP(TABLE)
             initial TABLEP(TABLE) =
               (TABLE=PRIMARY-TABLE( ))
  ovfns:   NEWTABLE( ) → TABLE
             effects 'PRIMARYTABLE( )=PRIMARYTABLE( )
                     'GET(K,T) = GET(K,T)
                     'TABLEP(T) = (TABLEP(T) or T=TABLE)
                     not TABLEP(TABLE)
                     TABLE~=?
           SAVE(VALUE,KEY,TABLE)
             assert  TABLEP(TABLE) and VALUE~=? and
                       KEY~=?
             effects 'PRIMARYTABLE( )=PRIMARYTABLE( )
                     'GET(K,T) =  if K=KEY and T=TABLE
                                    then VALUE
                                    else GET(K,T)
                     'TABLEP(T)=TABLEP(T)
           GETOP(KEY,TABLE) → VALUE
             assert  TABLEP(TABLE) and KEY~=?
             except  NOTTHERE: GET(KEY,TABLE)=?
             effects 'PRIMARYTABLE( )=PRIMARYTABLE( )
                     'GET(K,T)=GET(K,T)
                     'TABLEP(T)=TABLEP(T)
                     VALUE=GET(KEY,TABLE)
           PRIMARYTABLEOP( ) → TABLE
             effects TABLE='PRIMARYTABLE( )=
                       PRIMARY-TABLE( )
                     'GET(K,T)=GET(K,T)
                     'TABLEP(T)=TABLEP(T)
```

The basic idea is that there are a number of "search tables"—a special table called PRIMARYTABLE that exists initially, and as many secondary tables as the user wishes to create using the NEWTABLE instruction. In each table one can save a value under a key, writing over any previously saved value, or look up the value saved under a key. (We could simplify the argument structure and specifications of the instructions of this module by using only a single table and a GETOP instruction whose single argument combined the information in the two arguments of the present GETOP. But we believe that the version given yields a clearer implementation and it is also more suggestive of the implementation used by Deutsch [6].)

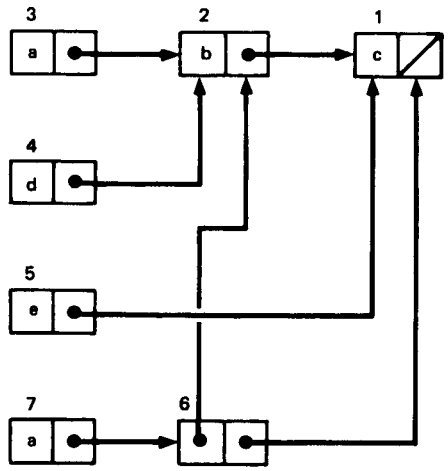
Note that the specification of GETOP uses an exception if there is no entry in the table under the key that is sought. It is worthwhile to contrast the use of exceptions and assertions in this specification. We assert that TABLEP(TABLE), indicating that a compilation or verification assert that TABLEP(TABLE), indicating that a compilation or verification may assume this fact in processing the implementation of GETOP but must verify it for uses of GETOP. The assertion describes a condition that must be guaranteed to hold at calls of the function. The exception, on the other hand, describes a condition—the presence of a particular entry in the table—that may or may not hold. Implementation programs are simplified by the possibility of structuring them to consider the "normal" and "exceptional" cases separately.

Any scheme for implementing unique lists requires determining whether a given pair of arguments to UCONS are the UCAR and UCDR of a previously UCONSed cell. In the specification of ULIST, the V-function UCELL serves to map from arguments pairs to cells; however, because of a quirk of Interlisp (the possibility of basing a hash probe on a single pointer but not on a pair of pointers), UCELL is not directly implementable in Interlisp. Instead, Deutsch employed—and we formally describe—a scheme based on two levels of search. This scheme is illustrated in Figure 2(c). The lists

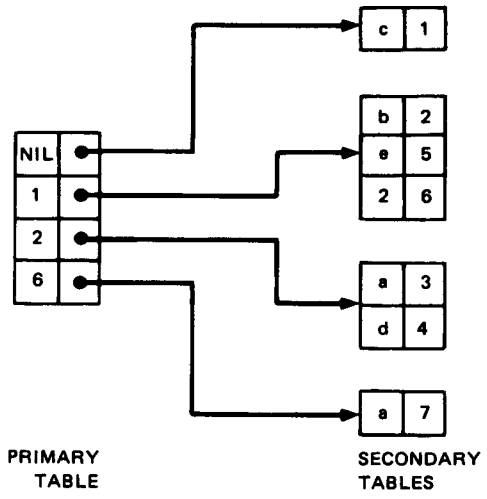
Fig. 2. Implementation of ULIST.

1. (c)
2. (b c)
3. (a b c)
4. (d b c)
5. (e c)
6. ((b c) c)
7. (a (b c) c)

(a) A SET OF SEVEN LIST STRUCTURES



(b) THE SAME LIST STRUCTURES IMPLEMENTED WITH SHARING OF COMMON CELLS



(c) THE SEARCH TABLES FOR THESE LISTS

in Figure 2(a) are shown as they might be uniquely represented in Figure 2(b). Each list cell is shown with a numeric cell identifier corresponding to the list number. The primary table of Figure 2(c) represents the association between UCDRs of cells and secondary tables. A secondary table, corresponding to a particular UCDR X2, then associates UCARs of cells with the cell, if any, having that UCAR and UCDR X2.

Implementation of ULIST × LIST by LIST × SEARCH is now possible. The implementation has three parts: a representation of states, an initialization program, and programs realizing each of the OV-functions of the upper level. First, the state representation is described by two formulas:

```
UCELL(X1,X2) ⇐ GET(X1,GET(X2,PRIMARYTABLE( )))
CELL(X1,X2,X) ⇐ CELL(X1,X2,X)
                  and GET(X1,GET(X2,
                  PRIMARY-TABLE( )))~=X
```

In these formulas, the left-hand sides refer to V-functions of the upper machine and the right-hand sides to V-functions of the lower machine. The first of these formulas describes the state correspondence needed to implement the two level search procedure: it provides that an upper state where UCELL(X1,X2) is not "?" will be represented by a lower state where the double search with X1 and X2 yields a result other than "?". Also, it requires that if UCELL(X1,X2) is "?", then the double search in the lower level state must yield "?" too.

The second formula describes how the upper level conventional lists are represented. The answer is, they are represented by lower level lists. We state this by using CELL on both sides of the definition, to be read as describing the upper level CELL in terms of the lower level CELL. However, there is a subtlety: some of the lower level lists will be used in the implementation to represent upper level unique lists and these, so far as the upper level is concerned, do not exist as lists. Thus, we add the second conjunct to this formula to exclude these lists from the set of upper level cells.

The initialization program must start from an initial state of LIST × SEARCH and arrive at a state of that machine that represents an initial state of ULIST × LIST. However, all that is required of an initial state of ULIST × LIST is that UCELL(X1,X2) is "?" and CELL(X1,X2,X) is "false" and, in view of the representation just given, this is represented by an initial state of LIST × SEARCH. Hence, the empty program suffices for initialization.

Finally, we must realize the upper level OV-functions UCONS, UCAR, UCDR, UCONSP, CONS, CAR, CDR, and CONSP by lower level programs. These are given below.

```
UCONS(X1,X2) : begin locals TABLE, C;
  execute TABLE ← GETOP(X2,PRIMARYTABLEOP( )) then
    on normal: execute C ← GETOP(X1, TABLE) then
      on normal: return(C);
      on NOTTHERE: C ← CONS(X1,X2);
                   SAVE(C,X1,TABLE);
                   return(C) end;
    on NOTTHERE: TABLE ← NEWTABLE( );
      C ← CONS(X1,X2);
      SAVE(C,X1,TABLE);
      SAVE(TABLE,X2,PRIMARYTABLEOP( )) end end;
UCAR(X)      : CAR(X);
UCDR(X)      : CDR(X);
UCONSP(X)    : begin locals TABLE, C;
  if CONSP(X)
    then execute
           TABLE ← GETOP(CDR(X),PRIMARYTABLEOP( ))
             then
         on normal:
           execute C ← GETOP(CAR(X),TABLE);
           then
```

```
            on normal:      return(C=X);
            on NOTTHERE: return(false) end;
        on NOTTHERE: return(false) end
    else return(false) end;
CONS(X1,X2)   : CONS(X1,X2);
CAR(X)        : CAR(X);
CDR(X)        : CDR(X);
CONSP(X)      : CONSP(X) and ~UCONSP(X);
```

First, note that occurrences of CONS, CAR, CDR, and CONSP in the defining programs denote these instructions on the lower level machine. Next, note that the defining programs for the top level functions CONS, CAR, and CDR are trivial because exactly the right instruction exists at the lower level. The defining programs for UCAR and UCDR are also single instructions; this is a consequence of the decision we made to represent upper level unique lists by lower level conventional lists. The nontrivial implementations are those for UCONS, UCONSP, and CONSP.

The implementation for UCONSP is a block that introduces two local variables: TABLE and C. If the argument X does not satisfy the lower level CONSP predicate, it cannot—in view of the representation—satisfy UCONSP. However, if it is a list cell, the UCONSP program uses the "execute" statement, a special feature of our implementation language. We use this statement to call an OV-function that may have exceptions and then deal with the normal exit and the exceptional exits in turn. Thus the UCONSP program first searches in the primary table with CDR(X) as key. If there is no exception, then the TABLE that results is searched with CAR(X) as key. A normal exit from this second search with result C indicates that C is a unique list with the same components as X and therefore is the only such unique list. Hence X is a unique list if and only if it is C. If either search has an exceptional exit, this means that there is no unique list with CAR(X) and CDR(X) as components. Thus UCONSP returns "false."

The implementation of UCONS has a similar structure. If both searches have normal exits and result C, then UCONS just returns C. If the first search encounters the NOTTHERE exception, this means that there are no existing unique lists with UCDR X2. Hence we create a new search table to record such unique lists, enter it in the primary table under key X2, create the new (representation of a) unique list CONS(X1,X2), and enter it in the new secondary table under key X1. The new list is then the answer returned by UCONS.

If the first search has a normal exit, but the second search has a NOTTHERE exception, this indicates that there is already a secondary search table TABLE for unique lists with UCDR X2, but that there is no entry in TABLE with X1 as UCAR. Hence we again create a new unique list representation CONS(X1,X2), enter it in TABLE under Key X1, and return it as the answer of UCONS.

Finally, the implementation of CONSP introduces some difficulty. Although there is a CONSP instruction at the lower level, it does not suffice: the lower level CONSP is satisfied by the lower level cells that represent upper level unique lists but these are not conventional lists in the abstraction provided by the upper level machine. We have given an implementation that makes an additional test [~UCONSP(X)] to avoid this problem, a correct but unpleasantly inefficient implementation of what ought to be a low-overhead type checking operation. For present purposes, the correct but inefficient implementation suffices; Section 9 discusses some alternatives.

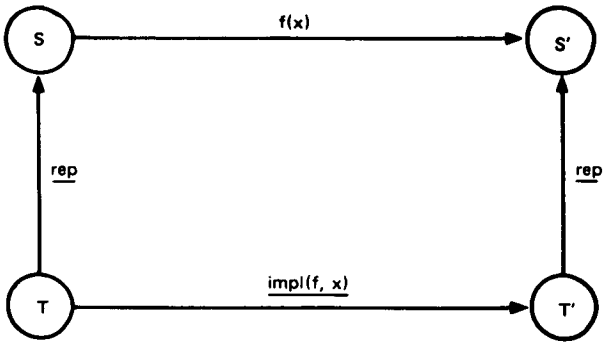
7. Correctness of the Implementation

The proof that an implementation is correct with respect to a pair of machine specifications and a state representation has two parts. First, we must prove that the initialization program for the lower level—in this case the empty program—can be executed from any initial state of the lower level machine to yield a lower level state that represents an initial state of the upper level machine. Second, we must prove that this representation is preserved by the execution of the implementations of OV-functions in the lower machine. That is, suppose S and S′ are states of the upper machine and T and T′ are states of the lower machine. Suppose that S′ is a state that results from the execution of an OV-function call "F(X)" according to the specification of the upper machine. Also, suppose that T′ is a state that results from the execution of the implementation of "F(X)" in the lower machine. Then, we must prove that T′ is a representation of S′. (This may be thought as a proof that the diagram of Figure 3 commutes.)

In doing these proofs, it is important to note that the execution of the implementations of the upper machine instructions does not fully exercise the facilities of the lower machine. For example, in our LIST × SEARCH machine there are states such that, for some X, the result of "GETOP(X,PRIMARYTABLEOP( ))" is neither an exception nor a secondary table but, instead, a list cell. Since we never store anything other than secondary tables in the primary table, we know that this can never occur and would like to use this knowledge to help the proof. We can do this by formulating an invariant predicate I(T) on the states T of the lower machine. We then prove that I holds for states resulting from the initialization of the lower machine. We also prove that if I(T) holds, and P is the implementation of an upper machine OV-function, then I(T′) holds where T′ results from T when P is executed in the lower machine. Having proved such an invariant property, we may assume that I holds for all states that arise in the proofs described in the previous paragraph.

We will illustrate the proof of correctness of implementations in this methodology (Figure 3) by proving the correctness of the implementation given in the preceding section. The necessary invariant assertion has two parts. First, if a fetch from the primary table yields a

Fig. 3. A necessary condition for implementation correctness.



result, not "?", then that result is a (secondary) table. Second, if a fetch from the secondary table yields a result, not "?", then that result is a list cell whose components are the keys of the two fetches. Stating this formally, we have

```
I(T) = GET(Z2,PRIMARYTABLE( ))=TABLE~=?
  implies (TABLEP(TABLE) and
    (GET(Z1,TABLE)=Z~=?
    implies (CONSP(Z) and CAR(Z)=Z1
    and CDR(Z)=Z2))).
```

(Note that the notation is such that the state T does not appear explicitly in the right-hand side of this definition; note also the implicit universal quantification of Z1, Z2, Z, and TABLE.) It is easy to show that I is an invariant of the lower machine states that arise in the implementation. It is true of the initial state because its antecedent is always false in this initial state. If it is true of a state T, then the execution of the implementations of UCAR, UCDR, UCONSP, CONS, CAR, CDR, and CONSP involve no calls of SAVE and therefore no changes in GET. The remaining case is the implementation of UCONS(X1,X2). This implementation can affect the truth of I because it does call SAVE. However, it calls SAVE only with C, which has the proper CAR and CDR and is stored under the appropriate keys, and with TABLE which does satisfy TABLEP (in view of the effects of NEWTABLE) and is also saved in the primary table under the proper key. Thus I is indeed invariant.

We have already shown that an initial state of the lower machine, followed by an empty initialization, represents an initial state of the upper machine. We must now prove that the representation of the upper machine state by the lower machine state is preserved by the execution of the implementations. It should be clear, in each case, that the result returned satisfies the corresponding specification. Except for UCONS and CONS, the instructions of the upper machine are implemented by programs whose only effect is the return of a result; thus these implementations all preserve the representation of the specified upper state by the resulting lower state.

The upper machine's CONS instruction is implemented by the CONS of the lower machine. Since the execution of the lower machine CONS affects only the V-function CELL, and only in a way consistent with the representation, this implementation also preserves the representation. The remaining upper machine construction is UCONS; consider its implementation. If both execute statements are "normal," there is no state change; that the result returned is correct is immediate from the invariant. If the outer execute statement has a normal exit and the inner a "NOTTHERE" exception, then the implementation creates exactly one new cell, and saves it in the proper table, thus preserving the representation of UCELL. Moreover, since the first conjunct of the representation of CELL becomes true exactly where the second conjunct becomes false, the specification that the representation of the upper level CELL be unaffected by execution of UCONS is satisfied.

Finally, if both execute statements have "NOTTHERE" exceptions, then a new secondary table is created and saved under the proper key in the primary table. This does not affect the representation, and the remainder of code in this case preserves the representation by the argument just made for the case of a single exception.

This completes the proof of the ULIST implementation.

8. Further Implementations

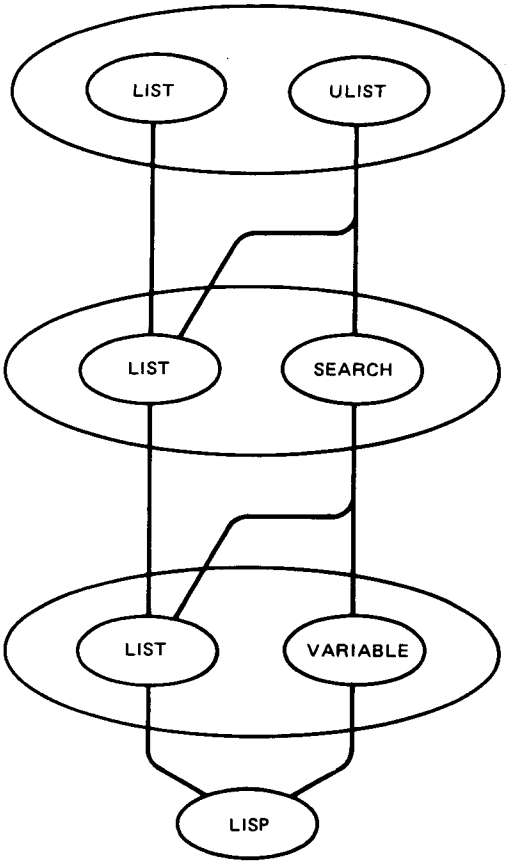
The preceding sections have described how properties of an unimplemented machine can be proved from its formal specifications, how such a machine can be realized in terms of a more primitive machine, and how such a realization can be proved with respect to the two machines. To save space, we will in this section sketch rather than fully presenting the further refinement of the LIST × SEARCH machine.

If Interlisp is an acceptable primitive machine, then the programs described so far solve the original problem, since it provides the LIST and SEARCH facilities to which we have reduced the problem. This would raise an interesting problem for the proof of the LIST × SEARCH specifications. The most complete extant specification of Interlisp, [19], is not written in SPECIAL; this proof would thus require a different theory from that discussed here.

A more primitive Lisp than Interlisp can also be used as the basis of our hierarchy. For example, one can easily implement the facilities of SEARCH, except for the PRIMARYTABLEOP instruction, in terms of Lisp lists; the implementation is just the usual Lisp "association list." The implementation of PRIMARYTABLEOP can be accomplished by using a single variable to remember which association list represents the primary search table. That is, LIST × SEARCH can be implemented in terms of VARIABLE × LIST. (VARIABLE is a very simple module: its state is the value saved in the variable and it has two instructions, one to read the value and one to save a new value.) The machines of this hierarchy, and their component modules, are shown in Figure 4.

Alternatively, one can distinguish two kinds of search

Fig. 4. An implementing hierarchy for unique lists.



operations in the implementation of ULIST—those that start from the primary table and those that start from one of the secondary tables. Actual use of ULIST suggests that it is reasonable to use a hashtable for the primary search table and association lists for the secondary tables. Since Interlisp provides named hashtables, this means that LIST × SEARCH could be realized by HASHTABLE × LIST and, in turn, HASHTABLE × LIST could be realized by Interlisp. (We will not provide an implementation of HASHTABLE here. The interested reader should consult [30]; in that paper, HASHTABLE is implemented in terms of arrays and a "hash probe" function and it is proved that the implementation is correct.)

9. Concluding Remarks

In Section 6, we implemented some specifications in an Algol-like language called ILPL, which is described in Appendix C of [22]. However, the use of this language, while convenient, is not essential to the use of our methodology. On the contrary, we believe that enough structure is given to even a large system by its decomposition and precise specification in SPECIAL to permit implementation in many languages. The critical points in the design and implementation of systems tend to be global issues such as a decision on how to decompose a system into modules or how to describe the implementation of a module by a hierarchy of abstractions—exactly the areas in which SPECIAL is expressive. By contrast, the details of any particular programming language usually address very local issues in programming, e.g. whether to use a case or conditional statement to describe a choice, whether to use a "while-do" or a "repeat-until" statement to describe a particular iteration. While such local decisions certainly have an impact on the clarity of the programs that can be written, we believe that this impact is negligible by comparison with the impact of well or poorly done overall design and specification. If the latter is precise, so that a large system is implementable by a large number of loosely coupled small parts, then many different languages may be equally good for implementing the parts.

This is not to deny that care should be taken in the choice of an implementation language. Certainly one ought to use a language with lucid syntax and a flexible set of control structures. Since we advocate the decomposition of a program into many parts, it follows that we recommend choosing a language that can be compiled into a form in which linkage between the parts is economical. Since we seek to implement systems, we are interested in the ultimate efficiency of implementations and therefore require a language in which machine-level representations can be described for the use of the most primitive levels of a hierarchy.

A related issue is the provision of data structures by the base language. For example, we assumed above that our base machine provided a set of objects satisfying the predicate ATOMP and disjoint sets of objects to represent abstractions such as cells and tables. If an adequate facility for defining concrete data types is present in the base, then it need not be provided by the hierarchy and—if the base language is carefully implemented—the cost of soundly manipulating objects of different types will be kept to a minimum. (Such a base should permit an efficient implementation of the upper level CONSP instruction in Section 6; by contrast, [6] is not intended as a hierarchical solution to the unique list problem, does not distinguish the list cells of the different levels of abstraction, and uses the same selectors CAR and CDR for all of them.) If the base does not have a sufficient facility, for example because it is a bare machine, then a type system must be synthesized as part of the hierarchy of machines. This can be quite hard, but it is possible [22].

Some base languages will provide not only concrete but also abstract data structures; these include CLU [15], a modification of Simula [23], Modula [31], and Alphard [32]. Some of these facilities are clearly redundant if our methodology is used with a tool that statically confirms that implementation programs are compatible with specifications, e.g. in what functions they call or what objects they refer to. On the other hand, the use of such a base language can ease the proof that implemen-

tation programs have the protection semantics implicit in the methodology.

Boyer and Moore have developed a formal semantics and a verification condition generator for our methodology [1], using the underlying theory of their Lisp Theorem Prover [2]. This makes it possible to produce precise machineable versions of the theorems given in Section 7 and preliminary experiments encourage us in the hope that these theorems may be mechanically proved. This will be a major theme in our future work.

There are certainly many other ways to specify programs formally. We think that the method of algebraic specifications [8] is very promising. It is similar to our method in its precision and compatibility with formal proof. It appears, in some published examples, to produce specifications that are quite concise but may require of readers greater mathematical sophistication than do ours; we are not aware of its use on examples as large as [22]. It would be premature to draw firm conclusions about the relative merits of the two methods and we look forward to the further development of both.

Acknowledgments. We thank B. Elspas, R. Boyer, and the referees for their helpful suggestions; R. Boyer and J Moore for their work on formalizing SPECIAL; and S. German for the idea of trying to prove Theorem 3.

Received January 1976; revised August 1977

References

1. Boyer, R.S., and Moore, JS. Private communication, June 1977.
2. Boyer, R.S., and Moore, JS. A lemma driven automatic theorem prover for recursive function theory. Proc. Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., Aug. 1977.
3. Burstall, R.M. Proving properties of programs by structural induction. *Comptr. J. 12*, 1 (Jan. 1969), 41–48.
4. Dahl, O.J., Myhrhaug, B., and Nygaard, K. Common base language, S-22. Norwegian Comptng. Ctr., Oslo, Norway, Oct. 1970.
5. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming.* Academic Press, New York, 1972.
6. Deutsch, L.P. An interactive program verifier. Ph.D. Th., Dept. of Comptr. Sci., U. of California, Berkeley, 1973.
7. Good, D.I. Provable programming. Proc. Int. Conf. Reliable Software, SIGPLAN Notices (ACM) *10*, 6 (June 1975), 411–419.
8. Guttag, J. Abstract data types and the development of data structures. *Comm. ACM 20*, 6 (June 1977), 396–404.
9. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1*, 4 (1972), 271–281.
10. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2*, 4 (1973), 335–355.
11. Ichbiah, J.D., et al. The system implementation language LIS. Tech. Rep. 4549, E/EN, Compagnie Internationale pour l'Informatique, Louveciennes, France, Dec. 1974.
12. Igarashi, S., London, R.L., and Luckham, D.C. Automatic program verification I: A logical basis and its implementation. *Acta Informatica 1*, 4 (1975), 145–182.
13. An appraisal of program specifications. Computation Structures. Group Memo 141-1, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., April 1977.
14. Liskov, B., and Zilles, S. Programming with abstract data types. Proc. ACM SIGPLAN Conf. Very High Level Languages, SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50–59.
15. Liskov, B., and Zilles, S. Specification techniques for data abstraction. *IEEE Trans. Software Eng. SE-1*, 1 (March 1975), 7–19.
16. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, Braffort and Hirschberg, Eds., North-Holland, Amsterdam, 1963, pp. 33–70.
17. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* M.I.T. Press, Cambridge, Mass., 1962.
18. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM 16*, 8 (Aug. 1973), 491–502.
19. Moore, JS. The Interlisp virtual machine specification. Rep. CSL 76-5, Xerox Palo Alto Res. Ctr., Palo Alto, Calif., Sept. 1976.
20. Morris, J. Protection in programming languages. *Comm. ACM 16*, 1 (Jan. 1973), 15–21.
21. Morris, J.M. Types are not sets. Proc. ACM Symposium on Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 120–124.
22. Neumann, P.G., et al. A provably secure operating system: The system, its applications, and proofs. Final Rep., SRI Proj. 4332, SRI Int., Menlo Park, Calif., Feb. 1977.
23. Palme, J. Protected program modules in Simula 67. Res. Inst. of Nat. Defense, Stockholm, Sweden, July 1973.
24. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330–336.
25. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15*, 12 (Dec. 1972), 1053–1058.
26. Robinson, L., and Levitt, K.N. Proof techniques for hierarchically structured programs. *Comm. ACM 20*, 4 (April 1977), 271–283.
27. Robinson, L., et al. On attaining reliable software for a secure operating system. Proc. Int. Conf. Reliable Software, SIGPLAN Notices (ACM) *10*, 6 (June 1975), 267–284.
28. Roubine, O., and Robinson, L. SPECIAL Reference Manual. Tech. Rep. CSL-45, SRI Project 4828, SRI Int., Menlo Park, Calif., 3rd ed., Jan. 1977.
29. Wegbreit, B. The treatment of data types in EL1. *Comm. ACM 17*, 5 (May 1974), 251–264.
30. Wegbreit, B., and Spitzen, J. M. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), 389–396.
31. Wirth, N. Modula: A language for modular multiprogramming. *Software—Practice and Experience 7* (1977), 3–35.
32. Wulf, W.A. ALPHARD: Toward a language to support structured programs. Comptr. Sci. Dept., Carnegie-Mellon U., Pittsburgh, Pa., April 1974.
33. Yourdon, E., and Constantine, L.L. *Structured Design.* Yourdon Press, New York, 1975.