An Introduction to the Construction and Verification of Alphard Programs

WILLIAM A. WULF, RALPH L. LONDON, AND MARY SHAW

Abstract—The programming language Alphard is designed to provide support for both the methodologies of "well-structured" programming and the techniques of formal program verification. Language constructs allow a programmer to isolate an *abstraction*, specifying its behavior publicly while localizing knowledge about its implementation. The verification of such an abstraction consists of showing that its implementation behaves in accordance with its public specifications; the abstraction can then be used with confidence in constructing other programs, and the verification of that use employs only the public specifications.

This paper introduces Alphard by developing and verifying a data structure definition and a program that uses it. It shows how each language construct contributes to the development of the abstraction and discusses the way the language design and the verification methodology were tailored to each other. It serves not only as an introduction to Alphard, but also as an example of the symbiosis between verification and methodology in language design. The strategy of program structuring, illustrated for Alphard, is also applicable to most of the "data abstraction" mechanisms now appearing.

Index Terms—Abstract data types, abstraction and representation, assertions, correctness, information hiding, levels of abstraction, modular decomposition, program specifications, program verification, programming languages, programming methodology, structured programming.

INTRODUCTION

OUR ultimate concern is with the cost and quality of *real* programs. Although problems that arise during maintenance of large programs are often ignored, it is nevertheless by now generally accepted that programming costs are too high, quality is too low, schedules are too often missed, and so on [5], [13], [28], [35].

The area called *structured programming* is concerned with those aspects of the software problem which result from our human limitations in dealing with complexity [1], [7], [9], [14], [29], [37], [38]. Recognizing that programs exist for long periods of time adds a new dimension, maintenance, to the problem, since it no longer suffices to develop the program in a well-structured manner. If a program is to be modifiable, the structure of the development must be retained in the ultimate program text. A major objective of the Alphard programming language design, currently under way at Carnegie-Mellon University, is precisely the retention of this structure. Alphard deals with complexity by restricting both the form of the programs (by eliminating the **goto**, for example [10]) and the process of creating them (as is the case with *stepwise refinement* [37]).

Research on program verification has addressed the software problem differently, by proving that the programs we write are in fact consistent with their specifications [12], [17], [19], [25]. Recently, attention has turned to verification of collections of related functions as a means of segmenting the verification task along the same lines as the decomposition of the program itself. For example, proof techniques described by Hoare [19] and Spitzen and Wegbreit [33], [34] can show that a data representation and its associated operations possess the expected properties, provided that the representation is directly manipulated *only* by the associated operations and not by other parts of a program. This decomposition and factorization permit parts of the verification to be performed for each operator definition instead of for each use. Ultimately, the techniques rely on induction on the number of data operations performed. Related proofs may be found in [15], [16], [41].

Well-structured, understandable, easily modified, and demonstrably consistent programs can in principle be written in any programming language. In practice, however, we know that the presence or absence of certain features in a language can materially affect all these desirable properties. We also know, from both natural and artificial languages, that the language we use to express our ideas can shape the ideas themselves [36]. Thus, by choosing language features and structure properly we can hope to exert a positive influence on the programs written in the language.

Instead of starting with an existing language and focusing on methodology or verification individually, we therefore chose to treat these issues jointly in a new language design. Alphard's abstraction mechanism, the **form**, encapsulates a set of related function definitions and associated data descriptions [27]. As a result, the user can attend independently to defining an abstract behavior and to using this abstract behavior in other programs. The strategy for verifying a **form** consists of showing that 1) the data structures used in the implementation constitute a valid representation of the abstract concept, 2) the initialization performed when an instance of the **form** is created produces a legitimate representation of an abstract object, 3) the implementation of each function behaves as its implementation description promises, and 4) the abstract description of the behavior is represented by these implementation descriptions.

Manuscript received April 17, 1976; revised July 29, 1976. The research described here was supported in part by the National Science Foundation under Grant DCR74-04187 and in part by the Defense Advanced Research Projects Agency under Contract F44620-73-C-0074, monitored by the U.S. Air Force Office of Scientific Research, and Contract DAHC-15-72-C-0308. The views expressed are those of the authors.

W. A. Wulf and M. Shaw are with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

R. L. London is with the University of Southern California Information Sciences Institute, Marina del Rey, CA 90291.

Preview of the Alphard Language

A key concept in structured programming is *abstraction*: the retention of only the essential properties of an object and the corollary neglect of inessential details. Several abstraction techniques have appeared in the literature on structured programming [11], [29], [30], [37].

In Alphard programs, information about the implementation of an abstraction is isolated and textually localized within a **form**. This has several advantages over more traditional organizations.

1) The places where modifications must be made are more likely to be close together.

2) A smaller portion of the program will be likely to require reverification when a change is made.

3) The user of the abstraction may ignore the details of the implementation.

4) It becomes possible to make *absolute* statements about certain things (e.g., data structures) which are independent of even perverse programmers.

5) The implementation of the abstraction may (sometimes) ignore the complexity of the environment in which the abstraction will be used.

The specific language mechanism used to capture this style of decomposition, the Alphard **form**, is derived from Simula *classes* [7]; similar adaptations have also appeared in CLU [23] and Concurrent Pascal [3], [4], and related features are beginning to appear in other languages (see, for example, [8]). At this point we shall only introduce the general nature of the construct and the Alphard notation; more details will follow an explanation of the verification issues.

The Alphard **form** permits the programmer to introduce a new abstraction into the program. In most ways the newly introduced abstraction will resemble a new *type* as that term is used in other programming languages.[1]

Thus, an Alphard program might contain a definition such as:

```
form complex=
    beginform
    . . .
    endform
```

This definition introduces a new abstract notion, "complex variable." The **form** contains all the information relevant to the implementation of the abstract notion. In this case, for example, we would find in the **form** both the definition of the data structure to be used in representing a complex variable (e.g., two real variables), and the definition of a set of operations on them (addition, multiplication, assignment, etc.). The **form** also gives a formal specification of the abstract properties of these complex variables as described in the next section.

Once such a definition is written, a programmer can write an *abstract* program using the newly defined notion. Variables of the new type may be declared, the defined operations may be performed, and so on. We may, for example, write:

```
local x,y,z:complex;
. . .
x←x+y*z;
. . .
```

because certain features of the language allow new functions to be associated with the infix operators.

All of this is, of course, very similar to the notions found in *extensible languages* [31]. However, the emphasis is considerably different: we are not interested in general syntactic extension. Rather, we are concerned with *encapsulation*, separating the concrete realization (implementation) of an abstraction from its use in an abstract program. Thus, for example, all of the representational information in a **form** is inaccessible to the abstract program; only those properties defined in the formal specification are accessible.

With this overview of the language, we turn to a technique for verifying the properties of a **form**. Since so much of the syntax and semantics of Alphard are tuned to this verification technique, we shall explain the technique first, then present the language via an extended example. For now, the important property of the language is its ability to separate the use of an abstraction from its concrete representation. The verification technique exploits this separation and permits the implementation (the **form**) to be verified independently of the abstract program in which it is used.

In order to show as clearly as possible the relation between language and verification we have omitted a number of issues from this discussion of Alphard. These include data representation, iteration mechanisms, reference variables, storage allocation, statement and expression syntax, exception handling, and input-output. At least for the programs given here, the reader's intuition and good sense should be sufficient.

Verification of Forms

Our overall strategy for verifying Alphard programs parallels the program decomposition implicit in the notion of a **form**. We shall presume a relatively small main program expressed in terms of operations on abstract objects natural to the problem. This main program is verified by traditional methods (e.g., inductive assertions), treating the specifications of the abstract objects and operations as if they were primitive. Then, to justify the use of the abstract objects we verify that the concrete implementation of each abstraction is consistent with its specifications. In general the implementation of an abstraction will be given in terms of further, *lower level*, abstract objects and operations on them. Thus, the verification of the algorithms used to implement an abstraction will be similar to the verification of the most abstract (top level) program. An obvious requirement of this approach is that each of the implementations be *correct*, or verified, if the ultimate program is to be verified. Roughly speaking, the verification will show that the specified relations exist between all abstractions and their implementations so that each implementation "behaves like," or models, its abstraction.

The key to the utility of this approach is separating the proof of each program that uses an abstraction from the proof

[1] In general, the abstraction introduced by a **form** need not be a type in the traditional sense. We use the word "type" informally in this paper, however, and the reader will not be badly misled by thinking in those terms.

of the implementation of that abstraction. Several advantages accrue from this separation.

1) Individual proofs are kept manageably small.

2) Program modifications generally imply reverification of only the affected program portion, usually a single **form** (exceptions occur when the modification affects the specification of the abstraction implemented by the **form**).

3) Although the entire program can be considered correct only when all portions have been verified, it is feasible for certain portions to be unverified during program development. Alternatively, some verified **forms** may be available from a library while others may have been developed and verified by a subgroup independently; these **forms** can be used confidently during the development of further programs or **forms**.

The remainder of this section explicates a proof methodology which permits this separation. It is based on ideas from Hoare's notable paper on correctness of data representations [19].

Suppose that we have an abstract type T, that "y" is an arbitrary object of type T, and that $A_1,\cdots,A_n$ are abstract operations defined on objects of type T. Out first concern will be to define the objects of this type and the operations on them in a manner which permits a *higher level* program to use these objects and be verified easily. This definition consists of three parts: the **specifications**, which constitute the user's sole source of information about the **form**, the **representation**, which describes the representation and related properties of an object of this type, and the **implementation**, which contains the definitions of the functions that can be applied to an object.

In the **specifications**, we first define the class of objects belonging to this type by a predicate which, for reasons which become clear later, is called the *abstract invariant* I_a. Second, since the abstract type T may be defined only under certain assumptions about the environment in which it is created, we capture these assumptions by a predicate β_{req}. Third, we give another predicate β_{init}, which characterizes the initial value given to an abstract object when it is created. Fourth, we define the abstract operations by their *input-output relations*, using pairs of predicates which characterize their effect. We call these β_{pre} and β_{post} and write in Hoare's notation [17]

$$\beta_{pre}(y)\ \{A_i\}\ \beta_{post}(y).$$

A_i is assumed to read or change only y.

Our next concern will be to characterize a concrete implementation of these abstract objects and operations. Suppose that "x" is the concrete representation of an object of type T, and hence, in general, "x" will be a collection or *record* of concrete variables. Further, suppose that $C_1,\cdots,C_n$ are the concrete operations which purport to be the implementations of the abstract operations $A_1,\cdots,A_n$. The set of concrete objects is also defined by a predicate, which we shall call the *concrete invariant* I_c. The relation between a concrete object x and the abstract object that x represents may be expressed by a *representation function*, **rep**:

$$\mathbf{rep}(x)=y$$

Note that the **rep** function may be many-one; that is, more than one concrete object may represent the same abstract object. **Rep** must, however, be defined for all x satisfying I_c.

The concrete operations C_i must also be characterized in terms of their input-output relations. To avoid confusion in the sequel we shall refer to these predicates as the input and output conditions, β_{in} and β_{out}, rather than as pre and post conditions. Thus,

$$\beta_{in}(x)\ \{C_i\}\ \beta_{out}(x).$$

We assume that each C_i alters or accesses variables only in x.

Finally, we shall presume a distinguished concrete operation C_{init} which is invoked whenever an object is created; this operation is responsible for initializing the concrete representation.

Now, at an intuitive level, we wish to show that the concrete representation and the implementation of the concrete operations are "correct." More specifically, we wish to show that it is safe for the programmer working at the abstract level to prove the correctness of his program using *only* the abstract specifications of the types he uses: I_a, β_{req}, β_{init}, and (for each abstract operation) β_{pre} and β_{post}. In the sequel, we often discuss an arbitrary function whose corresponding abstract and concrete operations are denoted by the symbols A and C, respectively; our remarks are therefore implicitly quantified over the set of such operations.

We have chosen to break the proof of the correctness of the concrete realization into four steps. The first step establishes the validity of the concrete representation. The second establishes that the concrete initialization operation is sufficient to ensure that β_{init} and I_c hold initially, provided β_{req} is satisfied. The third establishes that the code of the concrete operations is in fact characterized by the input-output assertions, β_{in} and β_{out}, and furthermore that I_c is preserved. The last step establishes the relation between the concrete input-output assertions and the abstract pre and post conditions. After describing the proof steps we discuss the relationship between this methodology and Hoare's.

For the form

Step 1: Validity of the representation[2]

$$I_c(x) \supset I_a(\mathrm{rep}(x))$$

Step 2: Initialization of an object

$$\beta_{req}\ \{C_{init}\}\ \beta_{init}(\mathrm{rep}(x)) \wedge I_c(x)$$

For each function

Step 3: Verification of concrete operations

$$\beta_{in}(x) \wedge I_c(x)\ \{C\}\ \beta_{out}(x) \wedge I_c(x)$$

Step 4: Relation between concrete and abstract specifications

[2]This condition is actually slightly stronger than necessary since we only need to ensure that those representations *reachable* by a finite sequence of applications of the concrete operations actually represent abstract objects; in practice, however, the stated theorem is not restrictive since I_c can be made stronger if necessary. Note, by the way, that we need not prove the dual ($I_a(y)$ implies the existence of an x such that $y=\mathrm{rep}(x) \wedge I_c(x)$) since this is guaranteed for reachable abstract objects by Steps 1–4.

a) $I_c(x) \wedge \beta_{pre}(rep(x)) \supset \beta_{in}(x)$
b) $I_c(x) \wedge \beta_{pre}(rep(x')) \wedge \beta_{out}(x) \supset \beta_{post}(rep(x))$

where the primed variable in Step 4b) represents the value of that variable prior to the execution of the operation.

Note that Steps 1 and 4 are theorems to be proved while 2 and 3 are standard verification formulas. Only the last step, 4, should require further explanation. Step 4a) ensures that whenever the abstract operation A could legally be applied in the higher level, abstract program (that is, whenever β_{pre} holds), the input assertion of the concrete operation β_{in} will also hold. Step 4b) ensures that if the concrete operation is legally invoked (that is, $I_c(x) \wedge \beta_{pre}(rep(x'))$ holds), then the output assertion of the concrete operation β_{out} is strong enough to imply the abstract post condition β_{post}. The four steps are sufficient but not necessary for the proof.

Hoare's similar technique for verifying the correctness of the implementation of an abstraction differs from the one described above in two respects. First, his approach does not deal explicitly with the issue of the validity of the representation, or distinguish explicitly between the concrete and abstract invariants. Second, he did not break the proof into several steps; we did so because we felt it would add clarity, would allow easier modifications of both **forms** and verifications, and would facilitate mechanical verification. In any case, except for Step 1, the two techniques are equivalent in the sense that from the proofs of one approach, we can derive the proofs required by the other. To obtain the proofs required by Hoare's approach from our proofs, merge Steps 3, 4a), and 4b) using the rule of consequence. Conversely, to obtain our proofs from Hoare's, choose β_{in} to be $\beta_{pre}(rep(x))$ and β_{out} to be $\beta_{post}(rep(x))$. Details are in [40].

In some cases it may be appropriate to show Hoare's combined form directly for each function. Hoare proves the theorem that if Step 2 and the combined form have been shown to hold for the implementation of some abstraction, then a concrete program using this implementation will produce the (representation of the) same result as an abstract program would have.[3] The proof of this theorem uses induction on the number of applications of operations in the abstract program. Our Steps 1 and 2 establish the basis step; Steps 1, 3, and 4 are used to establish the induction.

One might expect from this description of the methodology that the relationship

$$rep(x1) = rep(x2) \supset A(rep(x1)) = A(rep(x2))$$

would be true for arbitrary abstract functions A. Unfortunately, it is false. For example, let x1 and x2 be equal but not necessarily identical representations of a set S (i.e., x1 and x2 contain exactly the same elements, but in different orders); let the function A select an arbitrary element from S. The post condition for A is just $x \in S$, which does not specify uniquely which element to select.

In the next section we shall return to the description of Alphard and in particular to how the various pieces of information required by the proof technique are supplied in a **form**.

[3] Assuming, of course, that both the abstract and the concrete programs terminate.

First, however, we must say a few words about the predicate language in which the β's are expressed. There remains some controversy about the best specification techniques [24]. We do not wish to enter that debate here; we are content to await the emergence of one or more appropriate techniques and then adopt them. Alphard should accommodate more than one, and for the purposes of this paper we have chosen one we are comfortable with.

We shall presume the existence of a suitable collection of recognized mathematical entities such as integers, booleans, sets, sequences, multisets, matrices, and the operations defined on these entities. We assume that they have been defined precisely and that a rich collection of useful theorems has been proved for each. Our specifications will be stated in terms of these mathematical objects; in effect they will characterize a possible implementation in terms of the abstract mathematical entities. Thus, for example, in the next section we shall define an implementation of a (restricted) stack. The specification will characterize the stack operations in terms of operations on a sequence, with the sequence itself used to capture the state of the stack. A brief, informal definition of the notion of a sequence, adapted from [18], is included as an appendix.

Introduction to Alphard

We now explain the Alphard language by developing a definition of stacks and a program which uses stacks. These examples illustrate both the abstract definition facility and the interaction of verification considerations with language. We chose the stack for an example because it is familiar to most readers and because the Alphard program can be compared to other descriptions.

Forms

Imagine that while designing some program we found it desirable to use the notion of a stack—in particular, a stack whose elements are integers. We presume that our language does not contain stacks as a primitive concept, as indeed Alphard does not, so we want to introduce it as a new abstraction. Suppose further that an *a priori* depth limit is known or desired, so we need not define a general stack mechanism, only one which behaves like a stack so long as its depth does not exceed some predetermined maximum.

We shall lean heavily on the verification methodology developed above to explain the rationale for the various components of a **form** definition. We shall present the definition piecemeal, with each piece corresponding to some aspect of the verification technique. Starting at the top, the abstraction of a finite-depth stack of integers will be defined by a **form** such as:

```
form istack(n:integer)=
beginform
...
endform;
```

where "n" is the maximum permissible depth of the stack.

The purpose of such a **form** definition is to introduce a new abstract concept, to give it a name ("istack" in this case), and to define both its abstract properties and its concrete imple-

mentation. Note that we must carefully distinguish between the abstract concept introduced by such a definition and an *instance* of that concept. In general there may be many instances of an abstraction. Instances of abstractions are introduced into an Alphard program in several ways, but a common one is by *declarations*. Thus,

```
local x:istack;
```

has the effect of creating an instance of an istack and giving the name "x" to this particular instantiation. In the jargon of programming languages, this declaration *binds* the name "x" to an instantiation of istack.

We must now decide both what operations the abstract program shall be allowed to perform and what effects these operations shall have. In this case we shall allow only four operations: "push" makes a new entry at the top of the stack, "pop" deletes the current top element of the stack, "top" returns the value of the current top element of the stack, and "empty" returns "true" iff the stack if empty. (Obviously we could have chosen a more comprehensive set, but this will suffice here.)

The abstract program which uses the notion of an istack will apply these operations to instances of the abstraction. The **form** must provide a precise definition of these operations together with the concrete representation and operations to be used in implementing them. Thus, in general, a **form** is composed of three parts: **specifications**, **representation**, and **implementation**.

```
form istack(n: integer) =
beginform
specifications . . .;
representation . . .;
implementation . . .;
endform;
```

The **specifications** must provide the names of the operations supplied by the **form** together with the types of their arguments and results. In order for the user to be able to understand and use the abstraction solely in terms of the specification, and to permit verification, we must also include 1) a definition of the abstract domain, 2) the initial value of each entity of the abstract type, and 3) the pre and post conditions for each operation. Using the mathematical notion of a sequence we can write:

```
form istack(n: integer) =
beginform
specifications
  requires n>0;
  let istack = < . . . x_i . . . > where x_i is integer;
  invariant 0≤length(istack)≤n;
  initially istack=nullseq;
  function
    push(s:istack, x:integer)
      pre 0 ≤ length(s) < n post s=s'~x,
    pop(s: istack) pre 0 < length(s) ≤ n post s = leader(s'),
    top(s: istack) returns (x: integer)
      pre 0 < length(s) ≤ n post x = last(s'),
    empty(s: istack) returns (b: boolean)
      post b = (s=nullseq);
  representation . . .;
  implementation . . .;
endform;
```

Note how various pieces of information about the abstraction implemented by the **form** are introduced: the **requires** clause specifies β_{req}, the **invariant** clause specifies I_a, the **initially** clause specifies β_{init}, and each of the **function** clauses specifies β_{pre} and β_{post} for that function.[4] Furthermore, no particular implementation is demanded or precluded. Note that the exact size of the stack is parameterized so it can be set for each instantiation. We shall say more about this later, but we note here that not all values of the parameters may make sense. In this case, for example, a stack of negative size is senseless. Restrictions on the parameters are conveniently expressed in β_{req}, that is, the **requires** portion of the **specifications**.

In this case, then, the notion of an istack is explicated in terms of the mathematical notion of a sequence of bounded length. The operation "pop," for example, is defined to produce a new sequence which is just like the old one except that its last element has been deleted. (As before, the primed symbols in the **post** conditions, e.g., s', refer to the value of the (unprimed) symbol prior to execution of the operation.)

The **representation** portion defines the data structure which each instantiation of the **form** will use to represent the abstraction. It also specifies: 1) the initialization to be performed whenever the **form** is instantiated, 2) the **rep** function, which relates concrete to abstract descriptions, and 3) the concrete invariant. Thus, this section provides the major information relating an abstract entity and its concrete representation.

For this example we have chosen a simple representation for the stack. A vector holds the contents of the stack and an integer variable points to the top of the stack.

```
from istack (n: integer)=
beginform
specificiations . . .;
representation
  unique v: vector(integer,1,n), sp: integer init sp ← 0;
  rep (v,sp) = seq(v,1,sp);
  invariant 0 ≤ sp ≤ n;
  states
    mt when sp = 0,
    normal when 0 < sp < n,
    full when sp = n,
    err otherwise;
implementation . . .;
endform;
```

The first clause of the **representation** portion describes the concrete data structure(s) used to represent the abstraction; the key word **unique** used here indicates that the following

[4] To shorten the pre, post, in, and out conditions in this paper, we often omit assertions about variables which are completely unchanged. Thus, for example, we have omitted s=s' from the post condition of top. Such omitted assertions are nevertheless used in the proof steps. We also generally avoid in our proofs the legitimate concerns expressed in the term "clean termination"—such matters as array bounds checks, overflow, division by zero, and other inexecutable operations.

data structure(s) are unique to each instantiation (as opposed to being shared by, or **common** to, all instantiations). The **rep** clause specifies the representation function which maps concrete objects to abstract ones. The **invariant** clause specifies I_c. Also, note the **init** clause attached to the data structure declaration; this is the distinguished operation C_{init} mentioned in the previous section. The initialization operation is automatically invoked whenever an instantiation of the **form** is created, and is responsible for establishing β_{init}. Finally, experience in writing **forms** has shown that it is convenient to add another piece of information to the **representation**: a set of **state** definitions. These states are merely a shorthand for a set of Boolean conditions, but, as we shall see below, they help to accent certain interesting situations.

We would also like to note the use of the names "vector" and "integer" in this example. These are *not* primitive types of the language; they are simply **form** names. They happen to be the names of **forms** which will be automatically provided along with the compiler, but they are not special in any other way. (See [40] for a discussion of primitive types.)

The **implementation** portion of the **form** contains the bodies of the functions listed in the **specifications**, together with their concrete input and output assertions (β_{in} and β_{out}). In defining these functions bodies we make use of the states defined in the **representation** part. The **state** of the representation is determined when any function in the **form** is invoked, but is not reevaluated as changes to the representation are made within a function body. Thus, the state may be used, as in this example, to select one of several possible bodies for a function when it is called.

```
form istack(n: integer) =
beginform
specifications . . .;
representation . . .;
implementation
  body push out (s.sp = s.sp' + 1 ∧ s.v = α(s.v',s.sp,x))=
       mt,normal:: (s.sp ← s.sp + 1; s.v[s.sp] ← x);
       otherwise:: FAIL;

  body pop out (s.sp = s.sp'-1) =
       normal,full:: s.sp ← s.sp-1;
       otherwise:: FAIL;

  body top out (x = s.v[s.sp] ) =
       normal,full:: x ← s.v[s.sp] ;
       otherwise:: FAIL;

  body empty out (b = (sp=0)) =
       normal,full:: b ← false;
       mt:: b ← true;
       otherwise:: FAIL;

endform;
```

Since the states are used to select one of several alternative bodies for a function, the state descriptions may be used as additional input assertions for the body selected. Thus, for Step 3 of the proof we may add to the precondition the disjunction of the (state) conditions that can cause the selection of that body. The notation "α(V,i,x)," which is used in the output assertion of "push," denotes a vector identical to "V" except that V_i=x. Finally, the symbol FAIL used above is intended to connote failure; we prefer to avoid a detailed discussion of the exception mechanism in this paper and hence will avoid further elaboration of this symbol here.

Naming and Scope

The previous subsection dealt with the general organization of **forms**; in this subsection we describe the naming and scope rules. These rules make it possible for a **form** to encapsulate an abstraction through *information hiding*.

Several names are defined in the istack **form**. Some of these are the abstract operations (e.g., "push"), and others are related to the representation (e.g., "sp"). We know that the operation names must be available outside the **form**. In Alphard, however, names such as "sp" are *not* available outside the **form**.

Only names defined in the **specifications** part of the **form** are legal outside the **form** definition (inside is another matter). If names such as "sp" were legal outside the **form**, the abstract program could access, and possibly modify, the concrete representation. If this were allowed, both theoretical and practical difficulties would arise. First, we could not partition the proof technique as described above; specifically, we could not ensure that the concrete invariant was preserved between function invocations. Second, since the representation information would no longer be hidden it would no longer be safe to modify a **form** under the sole restriction that specified properties were preserved. We would instead have to examine all the uses of the abstraction to be sure that the representational information was not being used in some clever, but obscure, way.

Thus, the general scope rules in Alphard are Algol-like, but with two important exceptions.

1) Only those names appearing in the **specification** part of a **form** may be used outside the **form** definition. (All the names defined in a **form** may be used inside the same **form** definition.)

2) Only **form** names obey the usual block-structure convention on entering a **form**. Specifically, only those variables defined outside a **form** which are passed as parameters are accessible inside the **form** body.

These rules ensure that any dependency of the **form** on its environment is explicated in its parameter list. Similar adaptations of the Algol scope rules have appeared in other languages (e.g., CLU and Concurrent Pascal) and have recently been adopted in many more (see [8]).

An Example of a Form Verification: Restricted Stacks

In this section we shall illustrate the verification technique on the istack **form** of the previous section. First, however, let us pull together the pieces of the istack definition:

```
form istack(n: integer) =
beginform
specifications
  requires n>0;
  let istack = < . . . x_i . . . > where x_i is integer;
  invariant 0 ≤length(istack)≤n;
  initially istack=nullseq;
```

function
push(s:istack, x:integer) **pre** 0 ⩽ length(s) < n **post** s=s'~x,
pop(s: istack) **pre** 0 < length(s) ⩽ n **post** s = leader(s'),
top(s: istack) **returns** (x: integer)
pre 0 < length(s) ⩽ n **post** x = last(s'),
empty(s: istack) **returns** (b: boolean)
post b = (s=nullseq);

representation
unique v: vector(integer,1,n), sp: integer **init** sp ← 0;
rep (v,sp) = seq(v,1,sp);
invariant 0 ⩽ sp ⩽ n;
states
mt **when** sp = 0,
normal **when** 0 < sp < n,
full **when** sp = n,
err **otherwise**;

implementation
body push **out** (s.sp = s.sp' + 1 ∧ s.v = α(s.v',s.sp,x))=
mt,normal:: (s.sp ← s.sp + 1; s.v[s.sp] ← x);
otherwise:: FAIL;

body pop **out** (s.sp = s.sp'-1) =
normal,full:: s.sp ← s.sp-1;
otherwise:: FAIL;

body top **out** (x = s.v[s.sp]) =
normal,full:: x ← s.v[s.sp];
otherwise:: FAIL;

body empty **out** (b = (sp=0)) =
normal,full:: b ← false;
mt:: b ← true;
otherwise:: FAIL;

endform;

In the verification of istack, which is given next, the precondition for each body is the conjunction of its **in** clause (which is defaulted to "true") and the union of the state conditions for which that body is selected.

For the form
Step 1: Representation validity
Show: 0⩽sp⩽n ⊃ 0⩽length(rep(x))⩽n
Proof: length(rep(x)) = length(seq(v, 1, sp)) = sp

Step 2: Initialization
Show: n>0 {sp←0} rep(v, 0) = nullseq ∧ 0⩽sp⩽n
Proof: rep(v, 0) = seq(v, 1, 0) = <>, i.e., nullseq

For the function push
Step 3: Concrete operation
Show: (0=s.sp ∨ 0<s.sp<n) ∧ 0⩽s.sp⩽n {s.sp←s.sp+1; s.v[s.sp]←x} s.sp=s.sp'+1 ∧ s.v=α(s.v', s.sp, x) ∧ 0⩽s.sp⩽n
Proof: 0⩽s.sp<n ⊃ 0⩽s.sp+1⩽n

Step 4a): β_{in} holds
β_{in} is true

Step 4b): β_{post} holds
Show: 0⩽s.sp⩽n ∧ 0⩽length(rep(s.v, s.sp'))<n ∧ s.sp=s.sp'+1 ∧ s.v=α(s.v', s.sp, x) ⊃ s=s'~x
Proof: s=rep(s.v, s.sp) = seq(s.v, 1, s.sp'+1) = seq(s.v, 1, s.sp')~s.v[s.sp] = seq(s.v', 1, s.sp')~x = s'~x

For the function pop
Step 3: Concrete operation
Show: 0<s.sp⩽n ∧ 0⩽s.sp⩽n{s.sp←s.sp-1} s.sp=s.sp'-1 ∧ 0⩽s.sp⩽n
Proof: 0<s.sp⩽n ⊃ 0⩽s.sp-1⩽n

Step 4a): β_{in} holds
β_{in} is true

Step 4b): β_{post} holds
Show: 0⩽s.sp⩽n ∧ 0<length(rep(s.v, s.sp'))⩽n ∧ s.sp=s.sp'-1 ⊃ s=leader(s')
Proof: s=rep(s.v, s.sp) = seq(s.v', 1, s.sp'-1) = leader(s'). Note that leader(s') is defined since s.sp'⩾1

For the function top
Step 3: Concrete operation
Show: 0<s.sp⩽n ∧ 0⩽s.sp⩽n{x←s.v[s.sp]} x=s.v[s.sp] ∧ 0⩽s.sp⩽n
Proof: Clear

Step 4a): β_{in} holds
β_{in} is true

Step 4b): β_{post} holds
Show: 0⩽s.sp⩽n ∧ 0<length(rep(s.v, s.sp'))⩽n ∧ x=s.v[s.sp] ⊃ x=last(s')
Proof: x=s.v[s.sp] = s.v'[s.sp'] = last(s'). Last (s') is defined since s.sp'⩾1

For the function empty
Step 3: Concrete operation
(Normal, full) Show:
0<s.sp⩽n ∧ 0⩽s.sp⩽n{b←false} b = (s.sp=0) ∧ 0⩽s.sp⩽n
Proof: 0<s.sp ⊃ false = (s.sp=0)
(Mt) Show:
s.sp=0 ∧ 0⩽s.sp⩽n{b←true} b = (s.sp=0) ∧ 0⩽s.sp⩽n
Proof: s.sp=0 ⊃ true = (s.sp=0)

Step 4a): β_{in} holds
β_{in} is true

Step 4b): β_{post} holds
Show: 0⩽s.sp⩽n ∧ b = (s.sp=0) ⊃ b = (s=nullseq)
Proof: b = (s.sp=0) = (rep(s.v, s.sp)=nullseq) = (s=nullseq)
Q.E.D.

The condition n⩾0 is used implicitly in this proof. The stricter n>0 is needed only to show that the four states are disjoint. Finally, note that the union of the mt, normal, and full states includes I_c and that β_{pre} for each function and I_c specifically exclude the states that would trigger the **otherwise** alternative for the body. We therefore omit verifications involving FAIL.

Generalizing Form Definitions

The **form** of the previous section defines the abstract notion of a stack-of-integers, but what does the fact that the items to be stacked are integers have to do with it? It seems that

the *abstract* notion of a stack ought to be independent of the kinds of things being stacked.[5] We would like to be able to define a **form** such as

```
form stack(T:form, n:integer)=
beginform
...
endform
```

and then create instantiations with statements such as

```
local si:stack(integer,35), sr:stack(real,14);
```

which would make "si" a stack of integers and "sr" a stack of reals.

We shall do essentially this, but as we introduce this facility we must be very careful to retain the validity of the verification technique. In fact, we want to ensure something stronger: that the resulting proofs are not complicated by the introduction of this additional flexibility. Thus, we shall start with a careful examination of the proof appearing in the preceding section.

Specifically, let us observe how the proof depends upon the fact that the items being stacked are integers. A careful reading of the proof reveals that it depends only upon the property of the items that we have an assignment operation which obeys the *assignment axiom.*[6] The reader is encouraged to examine the proof to verify that this is in fact the only property required, and therefore to see that the proof would be valid for any type of item possessing this assignment axiom.

Returning to the language issues, what we want is a means for stating that the parameter "T" above cannot be just *any* **form** name; it must be the name of a **form** which supplies the properties required by the proof (and, of course, by the bodies of the concrete operations). The general mechanism used to accomplish this will be discussed below; for the moment we will consider only the special case which handles the stack example. With this addition the **form** "stack" has become a "type generator" rather than a simple type definition.

We shall append a bracketed list $<a_1,\cdots,a_n>$ to a formal parameter specification to denote that the properties $a_1,\cdots,a_n$ are *required* of a corresponding actual parameter. Thus, in the present case we may write the stack **form** header as:

```
form stack(T:form<↔>, n:integer)=
beginform
...
endform
```

The "<↔>" attached to the **form** parameter asserts that the actual **form** name used in this position must provide an assignment operation. The **specifications** part of the actual parameter **form** must assert the availability of this operation and assure that it obeys the assignment axiom. We shall discuss these issues in greater detail below, but first we shall give the **specifications** of the general stack definition and a verification of a program using it. The **specification** of the general stack differs from the version at the beginning of the previous section only in the italicized lines, which are the ones that previously referred to "istack" or "integer." The **representation** and **implementation** are identical to istack except for the substitution of "T" for "integer" in the vector declaration. The proof of this **form** is identical to that given above.

```
form stack(T:form<↔>, n:integer)=
beginform
specifications
   requires n>0;
   let stack = <...x_i...> where x_i is T;
   invariant 0≤length(stack)≤n;
   initially stack=nullseq;
   function
      push(s:stack, x:T) pre 0 ≤ length(s) < n post s=s'~x,
      pop(s:stack) pre 0 < length(s) ≤ n post s = leader(s'),
      top(s:stack) returns (x:T)
         pre 0 < length(s) ≤ n post x = last(s'),
      empty(s:stack) returns (b: boolean)
         post b = (s=nullseq);

representation ...

implementation ...

endform;
```

Once the stack **form** is defined, programs may declare and use stacks. The following program uses a stack as defined by this **form** to traverse a (finite) binary tree and count its tips. It also uses iteration and an explicit stack of binary trees [6], [25]. A binary tree is defined recursively to be either *nil* or to have a *left son* and a *right son* which are both binary trees. The number of tips is defined recursively by

```
tips(t) = if t=nil then 1 else tips(leftson(t))+tips(rightson(t))
```

We shall not define a binary tree **form** explicitly, but shall presume that it meets at least the **specifications**

```
isleaf(t:binarytree) returns b:boolean post b = (t=nil),
left(t:binarytree) returns u:binarytree pre t≠nil post u=leftson(t'),
right(t:binarytree) returns u:binarytree pre t≠nil post u=rightson(t')
```

We shall also presume a tree assignment operation satisfying the assignment axiom. In stating the maximum permissible depth of the stack we use the height function defined by

```
height(t) = if t=nil then 0
   else 1+max(height(leftson(t)), height(rightson(t)))
```

Suppose the tip counter is specified by

[5] Perhaps one can argue that the fact that all items in a *particular* stack are the same type, e.g., integers, is an abstract property of a stack, but it would be unfortunate if we had to define separate **forms** for stacks of integers, stacks of reals, stacks of characters, and so on.

[6] The assignment axiom is

$$P_e^x \{x \leftarrow e\} P$$

if x is a simple variable. For subscripted variables the meaning of x[i] := e is x := α(x, i, e) as in [18].

```
function tipcount(t:binarytree) returns count:integer
  post count=tips(t)
```

then the **body** of the function tipcount might be

```
body tipcount out (count=tips(t))=
begin
  unique s:stack(binarytree, max(height(t),1)), x:binarytree;
  x←t; count←1;
  invariant tips(t) = count - 1 + tips(x) + SIGMA_{u∈s}tips(u);
  while ¬ empty(s) ∨ ¬ isleaf(x) do
    if isleaf(x) then (count←count+1; x←top(s); pop(s))
      else (push(s, right(x)); x←left(x));
end
```

Throughout the **body** of tipcount the stack s means the abstract definition in terms of a sequence. In particular, $SIGMA_{u\in s}f(u)$ means 0 if s=nullseq and otherwise

$$f(last(s)) + SIGMA_{u\in leader(s)}f(u).$$

We shall verify the concrete operation of this body (i.e., proof Step 3). Note first that the **requires** clause (n>0) of the stack **form** is satisfied. We shall use the usual proof rule for the **while** statement.[7] Four verification conditions suffice; they are in the form obtained by backward substitution with each function operation of a **form** replaced by its **post** condition.

1) (entry to **while**)
   Show: $tips(t) = 1 - 1 + tips(t) + SIGMA_{u\in nullseq}\,tips(u)$
   where "nullseq" is obtained from the **initially** clause of stack.
   Proof: The SIGMA term is 0.

2) (**while** to exit)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u\in s}tips(u) \wedge \neg(s\neq nullseq \vee x\neq nil) \supset count = tips(t)$
   Proof: The SIGMA term is 0 because s=nullseq. tips(x)=1 since x=nil.

3) (**while** through **then** to **while**)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u\in s}tips(u) \wedge (s\neq nullseq \vee x\neq nil) \wedge x = nil \supset tips(t) = count + 1 - 1 + tips(last(s)) + SIGMA_{u\in leader(s)}tips(u)$
   Proof: x=nil means s≠nullseq whence last(s) and leader(s) are defined (i.e., the **pre** conditions for top and pop are satisfied). x=nil also means tips(x)=1. The conclusion follows by the definition of SIGMA.

4) (**while** through **else** to **while**)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u\in s}tips(u) \wedge (s\neq nullseq \vee x\neq nil) \wedge x\neq nil \supset tips(t) = count-1 + tips(leftson(x)) + SIGMA_{u\in s\sim rightson(x)}tips(u)$
   Proof: x≠nil means the **pre** conditions of both left(x) and right(x) are met. x≠nil also means tips(x) = tips(leftson(x)) + tips(rightson(x)). The conclusion follows by the definition of SIGMA. It remains to show that the **pre** condition of push is met. To do this it is convenient to add two terms to the **while** assertion:

   $$length(s) + height(x) \leqslant height(t)$$
   $$s=\langle s_1, \cdots, s_k\rangle \wedge 1\leqslant j\leqslant k \supset j + height(s_j) \leqslant height(t)$$

   Assuming these two terms are indeed invariants (proof omitted), the **pre** condition is met because x≠nil means height(x) ⩾ 1, i.e. length(s) < height(t).

Q.E.D.

[7] The **while** rule is $\dfrac{P \wedge B\,\{S\}\,P}{P\,\{\textbf{while } B \textbf{ do } S\}\,P \wedge \neg B}$. This is a special case of the Alphard iteration construct; it behaves as you would expect a **while** to behave. A more general iteration mechanism, which allows the author of a **form** to specify how iterations involving objects of that type are carried out, is described in [32].

Protection and Access Control

The "<>" notation introduced above is clearly an extension of the familiar notion of type checking in programming languages; in this section we shall try to show its relation to the protection facilities of modern operating systems, especially those using the *capability* based protection model. (The notion of incorporating protection in languages appears in [2], [26].) In the foregoing discussion we stressed the restrictions imposed on actual parameters by the appearance of the "<>" notation in a formal parameter list. We did not discuss either the restrictions it imposes on the body of the subroutine (or **form**) or the precise nature of what may appear between the angle brackets. Those issues will be treated here as well.

Note that "x:X<p>" appearing in a formal parameter list is intended to assert that the body depends on property p, and *only* on property p, of the parameter. (The word "property" is used intuitively here, but will be given a technical meaning below.) Now, from our earlier discussion we know that the only visible properties of an abstraction are those specified in its **specifications** part. Thus we require that the name "p" be one of the names defined in the **specifications** part of the **form** X. Furthermore, since the abstraction being defined claims to depend *only* on the property p, we shall restrict the body of the abstraction to use only this property. That is, all uses of x in contexts other than p(x,···) are illegal. (Note that this is a purely syntactic, compile-time, check. Also note that we must check that any functions called by the body of the abstraction, where x is a parameter to that function, must also require no more than "p" access to it.)

In the terminology of operating systems the **specifications** part of a **form** defines a set of *accesses* to *objects* of the type defined by the **form**. The "<>" notation defines both the access rights *required* of the actual parameter and *allowed* to the body. Once the actual parameter has been bound to the formal at execution time the formal becomes the name of a *capability* [20], [22] for the actual. At compile time the formal parameter specification may be viewed as a *template* [39] for legal actuals.

The analogy with the capability-based model of protection is not yet complete. In an operating system it is generally possible to *restrict* access rights; the "<>" notation permits us to do this at formal/actual parameter binding, but may also be useful in other contexts. For verification purposes, for example, it may be convenient to know that in some block no side-effect producing operations are applied to a specific variable.

A full treatment of a mechanism which provides the type of protection we desire may be found in [21]; the Alphard mechanism is essentially identical to that discussed there. For our present purposes we shall simply note that the "<>" notation is permitted in several additional contexts, two of which are discussed below, and in these contexts implies only a rights restriction (not also a requirement as in formal parameter specifications). These contexts are declarations and actual parameters. Consider the declaration:

```
local i:integer<+,-,=,←>;
```

This declaration defines a variable of type integer to which only the operations "+," "-," "=," and "←" may be applied. Any other operations defined by the integer **form** will be illegal–specifically such things as "*," "/," and relational tests. Such a declaration might be used for a variable which is intended only for use as a counter, for example.

By attaching a rights restriction to the actual parameter of a subroutine invocation the user may ensure that only certain operations are applied by the subroutine. Thus, in the program:

```
begin
local i:integer;
...
f(i<+,-,*>);
...
end;
```

the main program has all access rights to the variable "i," but restricts the operations that may be performed by "f" to those listed. This is, perhaps, a somewhat strained example since the more common case will be to restrict side-effect producing operations; hopefully, however, it illustrates the point. Once again let us emphasize that this is a purely static, compile-time check. At compile time, the rights *permitted* by the actual parameter are compared to those *required* by the formal; if the former are not a superset of the latter a compile-time error message is generated. There is no run-time overhead.

The "<>" notation provides a means of specifying the *required* properties of actual parameters. We shall now introduce *questionmark identifiers* to relax the specifications of formal parameters so that the binding of certain properties may be deferred. A questionmark identifier is (syntactically) simply a "?" immediately followed by an identifier, e.g., "?xyz." Consider the skeletal function definition

```
function f(a:?T) ...
```

The use of "?T" permits us to specify that the abstract computation defined by "f" does *not* depend on the type of the parameter "a." That is, the function will operate as specified by its input/output assertions independent of the type of the actual parameter. (The execution, and possibly the compilation, of "f" *will*, of course, depend on the type of the actual parameter.)

Defining occurrences of such identifiers appear in formal parameter lists and are assigned a meaning from the corresponding actual parameter. Multiple occurrences of the same ?identifier are required to have the same meaning in the same scope. *Applied occurrences* of ?identifiers may appear anywhere in the scope of their definition–thus, for example, they may be used to declare variables of the same type as an actual parameter.[8]

Now let us turn to the question of what may be written between the angle brackets, especially in the context of a

[8] There are somewhat pathological situations involving recursive procedures in which this scheme will not work; in particular in these cases it is not possible to determine the proper types at compile time. We choose to ignore these pathologies here.

formal parameter specification, and to the interaction of the "<>" and the ?identifiers. To this point we have simply written the *name* of a property, which is generally a function name. This is sufficient in the cases where the type of the formal is specified, but not when the type is characterized by a ?identifier. Consider an example which involves less suggestive names than those used previously:

function f(a:?T<h>)= . . .;

The intent is, as before, that the function "f" depend only on the fact that the actual parameter be of a type which provides an "h" operation, but not on the name of the type itself. But suppose that the type of the actual parameter does provide an operation named "h," but it has nothing to do with the operation which the writer of "f" had in mind. In fact, the writer of "f," or alternatively the correctness of "f," depends on some input-output relation of the "h" operation. Thus, we permit properties appearing in the angle brackets to be described in exactly the same manner as properties appearing in the **specifications** part of a form definition. For example,

function f(a:?T<h(T,integer) **returns** (b:boolean)
pre β_1 **post** β_2>)= . . .;

When such specifications appear the problem of validating the legality of an actual parameter is more complex than it was previously. We must not only establish that the **form** defining the type of the actual parameter provides a property named "h," but also that: 1) its parameters and result are of the appropriate type and 2) that the precondition required in the specification of that property is implied by β_1 and that the post condition of that property is sufficient to imply β_2. We do not foresee this proof as part of the compilation process, but rather as another proof required in the verification of the program.

Conclusions

We have presented the basic components of an Alphard **form** and explained the reasons for and uses of each component. We have illustrated the development, verification, and use of a rather general-purpose **form.** We have also shown how this abstraction may be used in the implementation of another program, and how the specification of the abstraction is used in the verification of this program; although the example is small, we hope it illustrates how the decomposition methodology supported by Alphard permits the verification of a large system to be broken into manageable steps.

The length of this paper has prevented us from discussing many important issues in Alphard and their relation to verification. For example, we have not explored modification of the stack example. Suppose, however, we were to change the **representation** and the **implementation**, although not the **specifications**. The verification of the stack **form** changes, of course, but both the program using it, tipcount, and the verification of that use would be totally unchanged.

The full generality of the **form** concept has not been displayed in this paper. Other papers [32], [40] provide a broader discussion of the Alphard language and our experiences with it. (The latter report is an expanded version of this paper.) They contain more examples of Alphard programs and further evidence of the interaction between additional language mechanisms (especially iteration) and verification issues.

It should be noted that, although this paper may appear to be an explication of a particular language, Alphard, in fact the strategy we illustrated is applicable to most of the "data abstraction" mechanisms now beginning to appear in many languages. We would like to emphasize, however, that merely adding an abstraction facility to an existing language is unlikely either to produce a coherent design or to achieve all the goals set out in the introduction. The degree of interaction between methodological and verification concerns during the Alphard design has been substantial, and we doubt that similar results could have been achieved without the freedom to make drastic changes to nearly all aspects of the initial language definition. We have been surprised and extremely pleased at the degree to which methodology and verification have reinforced each other both to produce a coherent language design and to enable us to reach our other goals.

Appendix

Informal Definition of Sequences

$<s_1,\cdots,s_k>$	denotes the sequence of elements specified; in particular, "<>" denotes the empty sequence, "nullseq."
$s \sim x$	is the sequence which results from concatenating element x at the end of sequence s.
length(s)	is the length of the sequence "s."
first(s)	is the first (leftmost) element of the sequence "s."
trailer(s)	is a sequence derived from "s" by deleting the first element.
last(s)	is the last (rightmost) element of the sequence "s."
leader(s)	is a sequence derived from "s" by deleting the last element.
seq(V,n,m)	where "V" is a vector and "n" and "m" are integers, is an abbreviation for the sequence "$<V_n,V_{n+1},\cdots,V_m>$"; alternatively, seq (V,n,m) = seq(V,n,m - 1) $\sim V_m$.

Note: first, trailer, last, and leader are undefined for "<>."

Acknowledgment

We owe a great deal to our colleagues at Carnegie-Mellon University and the University of Southern California Information Sciences Institute, especially: M. Barbacci, D. Good, J. Guttag, P. Hilfinger, D. Jefferson, A. Jones, D. Lamb, D. Musser, K. Perdue, K. Ramakrishna, and D. Wile. We would also like to thank J. Horning and B. Liskov and their groups at the University of Toronto and the Massachusetts Institute of Technology, respectively, for their critical reviews of Alphard. Finally, we very much appreciate the perceptive responses that a number of our colleagues have made on an earlier draft of this paper.

REFERENCES

[1] F. T. Baker, "Chief programmer team management of programming," *IBM Syst. J.*, vol. 11, pp. 56–73, 1972.

[2] P. Brinch Hansen, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973, ch. 7.

[3] —, "Concurrent Pascal report," *Information Science Rep.*, California Institute of Technology, 1975.

[4] —, "The programming language concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199–207, June 1975.

[5] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[6] R. M. Burstall, "Program proving as hand simulation with a little induction," in *Proc. IFIP Congress 74*, 1974, pp. 308–312.

[7] O.-J. Dahl and C. A. R. Hoare, "Hierarchical program structures," in *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972, pp. 175–220.

[8] *Proc SIGPLAN/SIGMOD Conf. Data: Abstraction, Definition, and Structure and Supplement to the Proc.*, Mar. 1976.

[9] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT*, vol. 8, pp. 174–186, July 1968.

[10] —, "Go to statement considered harmful," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 147–148, Mar. 1968.

[11] —, "Notes on Structured programming," in *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972, pp. 1–82.

[12] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Mathematics*, vol. 19, J. T. Schwartz, Ed., American Mathematical Society, 1967, pp. 19–32.

[13] J. Goldberg, Ed., *Proc. Symp. High Cost of Software*, Stanford Res. Inst., Stanford, CA, Sept. 1973.

[14] D. Gries, "On structured programming—A reply to Smoliar," *ACM Forum, Commun. Ass. Comput. Mach.*, vol. 17, pp. 655–657, Nov. 1974.

[15] J. V. Guttag, "The specification and application to programming of abstract data types," Ph.D. dissertation, University of Toronto, Toronto, Ont., Canada, CSRG Tech. Rep. 59, Sept. 1975.

[16] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *USC Information Sciences Inst. Tech. Rep.*, 1976.

[17] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576–580, 583, Oct. 1969.

[18] —, "Notes on data structuring," in *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972, pp. 83–174.

[19] —, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271–281, 1972.

[20] A. K. Jones and W. A. Wulf, "Towards the design of secure systems," *Software-Practice and Experience*, vol. 5, pp. 321–336, 1975.

[21] A. K. Jones and B. H. Liskov, "An access control facility for programming languages," Massachusetts Institute of Technology Computation Structures Group Memo 137 and Carnegie-Mellon University Tech. Reps., 1976.

[22] B. Lampson, "Protection," in *Proc. 5th Princeton Conf. Information Sciences and Systems*, 1971, pp. 437–443.

[23] B. H. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, pp. 50–59, Apr. 1974.

[24] —, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7–19, Mar. 1975.

[25] R. L. London, "A view of program verification," in *Proc. Int. Conf. Reliable Software*, Apr. 1975, pp. 534–545.

[26] J. H. Morris, Jr., "Protection in programming languages," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 15–21, Jan. 1973.

[27] —, "Types are not sets," in *Proc. ACM Symp. Principles of Programming Languages*, 1973, pp. 120–124.

[28] D. L. Parnas, "Information distribution aspects of design methodology," in *Proc. IFIP Congr. 1971*, Booklet TA-3, pp. 26–30.

[29] —, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comp. Mach.*, vol. 15, pp. 1053–1058, Dec. 1972.

[30] —, "A technique for software module specification with examples," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 330–336, May 1972.

[31] S. A. Schuman, Ed., "Proceedings of the international symposium on extensible languages," *SIGPLAN Notices*, vol. 6, Dec. 1971.

[32] M. Shaw, W. A. Wulf and R. L. London, "Abstraction and verification in Alphard: Iteration and generators," Carnegie-Mellon University and USC Information Sciences Institute Tech. Reps., 1976.

[33] J. Spitzen and B. Wegbreit, "The verification and synthesis of data structures," *Acta Informatica*, vol. 4, pp. 127–144, 1975.

[34] B. Wegbreit and J. M. Spitzen, "Proving properties of complex data structures," *J. Ass. Comput. Mach.*, vol. 23, pp. 389–396, Apr. 1976.

[35] G. M. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand-Reinhold, 1971.

[36] B. L. Whorf, "A linguistic consideration of thinking in primitive communities," in *Language, Thought, and Reality*, J. B. Carroll, Ed. Cambridge, MA: MIT Press, 1956.

[37] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, Apr. 1971.

[38] W. Wulf and M. Shaw, "Global variables considered harmful," *SIGPLAN Notices*, vol. 8, pp. 28–34, Feb. 1973.

[39] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 337–345, June 1974.

[40] W. A. Wulf, R. L. London, and M. Shaw, "Abstraction and verification in Alphard: Introduction to language and methodology," *Carnegie-Mellon University* and *USC Information Sciences Inst. Tech. Reports*, 1976.

[41] S. N. Zilles, "Abstract specifications for data types," IBM Research Laboratory, San Jose, CA, Jan. 1975.

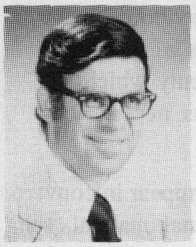
William A. Wulf was born in Chicago, IL, in 1939. He received the B.S. degree in engineering physics and the M.S. degree in electrical engineering from the University of Illinois, Urbana, and the Ph.D. degree in computer science from the University of Virginia, Charlottesville.

He was an Instructor of Computer Science and Applied Mathematics with the University of Virginia. He is currently Professor of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. His primary research interest is in the design and implementation of computer software (especially operating systems and compilers), languages suitable for software implementation, and hardware architecture for the efficient execution of these software systems. Related research interests include protection structures, programming methodology, and compiler optimization. He has directed the construction of several large software systems and is currently directing the construction of C.mmp, a 16-processor multiprocessor computer system, and its associated software.

Dr. Wulf is a member of the Association for Computing Machinery and the SIGPLAN Executive Committee. He is also Chairman of the IFIP Working Group on Machine-Oriented Higher-Level Programming Languages and is a member of the Editorial Board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Ralph L. London was born in Johnstown, PA, on April 17, 1936. He received the B.A. degree from Washington-Jefferson College, Washington, PA, and the M.S. and Ph.D. degrees from Carnegie-Mellon University, Pittsburgh, PA, all in mathematics, in 1958, 1960, and 1964, respectively.

He is Project Leader and Research Staff Member at the University of Southern California Information Sciences Institute, Marina del Rey, CA, and is also Associate Professor of Computer Science at the University of Southern California, Los Angeles. Prior to 1972 he was Assistant



Professor and later Associate Professor of Computer Sciences at the University of Wisconsin, Madison. During 1971–72 he was a Research Associate at the Artificial Intelligence Project, Stanford University, Stanford, CA. His research involves the verification of computer programs, particularly the building of interactive computer programs to assist in program verification.

Dr. London is a member of the Association for Computing Machinery (ACM), Phi Beta Kappa, and the International Federation of Information Processing Working Group on Formal Description of Programming Concepts. Currently an ACM National Lecturer on program verification, he has served on the program committees of the 1972 ACM Conference on Proving Assertions about Programs, and the 1975 International Conference on Reliable Software.

Mary Shaw was born in Washington, DC, in 1943. She received the B.A. degree in mathematics from Rice University, Houston, TX, in 1965, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh PA, in 1972.

She is currently Assistant Professor of Computer Science at Carnegie-Mellon University. Her primary research interests lie in the areas of programming methodology and programming language design, particularly software tools for program development and the interactions between specifications, language, and correctness.

Dr. Shaw is a member of the Association for Computing Machinery and Sigma Xi. She is also a member of the Committee of Examiners for the Graduate Record Examination's Advanced Test in Computer Science.

Software Development

HARLAN D. MILLS

Abstract—**Software development has emerged as a critical bottleneck in the human use of automatic data processing. Beginning with ad hoc heuristic methods of design and implementation of software systems, problems of software maintenance and changes have become unexpectedly large. It is contended that improvement is possible only with more rigor in software design and development methodology. Rigorous software design should survive its implementation and be the basis for further evolution. Software development should be done incrementally, in stages with continuous user participation and replanning, and with design-to-cost programming within each stage.**

Index Terms—**Design-to-cost programming, software design, software development, software maintenance, top-down development.**

Twenty-Five Years of Data Processing

The Data Processing Explosion

IN THE PAST twenty-five years a whole new data processing industry has exploded into a critical role in business and government. Every enterprise or agency in the nation of any size, without exception, now depends on data processing hardware and software in an indispensable way. In a single human generation, several hardware generations have emerged, each with remarkable improvements in function, size, and speed. But there are significant growing pains in the software which connects this marvelous hardware with the data processing operations of business and government.

Had this hardware development been spaced out over 125 years, rather than just 25 years, a different history would have resulted. For example, just imagine the opportunity for orderly industrial development with five human generations of university curriculum development, education, feedback for the expansion of useful methodologies and pruning of less useful topics, etc. As it is, we see a major industry with minimal technical roots, because almost no one in a responsible position has an original university education in the subject, and the universities have no experience in even knowing what to teach. In comparison, it is worth noting just how many years and how much give and take has gone into the development of the current mathematics curriculum to support engineering and the physical sciences—at least the 125 years imagined earlier.

Even so, from ground zero, the technical and industrial progress of society in 25 years of data processing is impressive. But the needs and frustrations are so great that some perspective is in order to better understand how we got here and where we might be going.

Data Processing Then

Before the last 25 years, these same enterprises and agencies conducted their operations without automatic data processing, while still processing data in sufficient amounts to manage their affairs. But the data processing was done by people. Even if desk calculators, or tabulators, were used here and there, people still inspected intermediate results, and applied their common sense, where necessary, to correct obvious mistakes. If data processing instructions were faulty, or missing, people used common sense, again, to make the operations work. In other words, data processing systems were forgiving systems, because of the intelligence used in their execution.

Such forgiving systems permit the evolution and natural selection of data processing improvements in an orderly way. If an improvement is proposed, it is easily adopted with little risk, because unforeseen side effects will usually be noticed and suppressed by people. As a result, data processing is done,

Manuscript received April 14, 1976; revised July 21, 1976.
The author is with the IBM Corporation, Gaithersburg, MD 20760.