CHARACTERISTIC FUNCTIONS AS A BASIS OF DATA SPECIFICATION AND PROOF[+]

by

Jayadev Misra

TR-96a March 1979

+ Research partially supported by NSF Grant DCR75-09842.

University of Texas, Austin, TX 78712

ABSTRACT

A method for specification of data abstraction, using characteristic functions, is proposed in this paper. It is shown that the notion of characteristic functions is a generalization of V-function of Parnas[7]. This technique provides specifications which are simple to understand, yet rigorous enough for formal proofs. Verification of implementation is illustrated with examples.

1. Introduction

A method of data specification is proposed in this paper which uses a generalization of Parnas V-functions [7], called characteristic functions. Characteristic functions are shown to provide a rigorous specification of data, which could be a basis for formal proofs about the data. At the same time the notion of characteristic functions appeals to intuition so that the specification can be understood and constructed quite easily. We provide several examples of specification using characteristic functions. Finally, we show that characteristic functions provide a basis for construction of mappings, for proving data representations similar to that of Hoare [3].

Liskov and Zilles [5] list six criteria for evaluating specification methods: formality, constructability, comprehensibility, minimality, wide range of applicability, extensibility. Axiomatic approach [5] has advantages as a formal, representation independent scheme. However, the specifications are often incomprehensible unless the reader is familiar with the data being designed. Axiomatic approach also creates its own problems of completeness and consistency. Finally, a problem of mutual reference arises; a function is never specified all by itself; a set of functions are specified in terms of each other. Hence a group of functions have to be understood simultaneously. Proof rules have to be applied to a group of functions.

Abstract model approach of Liskov and Berzin [6] represents the data in terms of other, well understood data. This approach has the advantage that the specification is simple to construct and understand. However, the specification turns out to be fairly long and requires an understanding of other types of data. Proofs using this scheme will tend to be quite involved. even though the steps are straightforward.

State_machine approach of Parnas [7] has long been a useful tool for specification and design. However, it was not adequate as a basis for formal proofs. The original idea of the method is to divide the set of functions operating on data into two types: O-function (operating function) and V-function (value returning function). The effect of application of every O-function is described by the changes in the values of the V-function. A user may rely only on the values returned by V-functions to access and examine the data. This scheme is however inadequate, since certain O-functions may not immediately affect the values of V-function, but may have a delayed effect. A notion of hidden V-functions has been introduced in [8] to overcome of this difficulty.

Characteristic functions may be viewed as a generalization of V-functions. The attempt here is to provide a rigorous definition mechanism, based on the notion of equality of data objects of the given type. Every operation's semantic is defined in terms of its effect on characteristic functions values. It can be shown that if the characteristic functions are sufficient to define equality, then they constitute a complete mechanism for specification. Several examples illustrate the application of this technique. Several aspects of verification are considered. It is shown that an implementation may be proven correct, by axiomatizing the characteristic functions in terms of the operations that manipulate the representation; this provides a mapping from representation to the abstract object in the sense of Hoare [3]. Some theoretical aspects regarding alternate characteristic functions are considered.

2. Basic Notion

Consider two sets s_1, s_2. These two sets are defined to be equal on the basis of classical set theory, when their membership tests are identical. If ϵ denotes the membership function [$\epsilon(i,s)$ = true if and only if i is a member of s],

$$s_1 = s_2 \iff \epsilon(i,s_1) = \epsilon(i,s_2), \text{ for all } i.$$

In other words, a set can be uniquely described by specifying the values of $\epsilon(i,s)$, for all i. Semantic of an operation on set can be specified by describing the changes in the values of ϵ. We call ϵ a characteristic function and define the operations on set by describing their effects on ϵ.

Example 1

Data type: SET (of elements of type T)

operations:

INIT: $\rightarrow$ SET

ADD: T x SET $\rightarrow$ SET

DELETE: T x SET $\rightarrow$ SET

SEARCH: T x SET $\rightarrow$ boolean

Characteristic Function: ϵ

Definition of Equality:

$$S_1 = S_2 \iff \epsilon(i,S_1) = \epsilon(i,S_2) \quad \forall i \text{ of type T.}$$

Note that we assume that a notion of equality has been defined on elements of type T.

In the following description, only those values of ϵ which are changed are specified; i,S denote arbitrary objects of type T and SET respectively.

Operation	Remark	Semantic
INIT(S)	S is now the null set.	$\epsilon(i,S)$ = false, $\forall i$ of type T.
ADD(i,S)	'i' is added to S.	$\epsilon(i,S)$ = true.
DELETE(i,S)	'i' is removed from S. [Note: i may not be in S when this is applied.]	$\epsilon(i,S)$ = false.
SEARCH(i,S)	returns true if i is in S.	SEARCH (i,S) = $\epsilon(i,S)$

□

It is important to note that there is no particular semantic or condition attached to ϵ.

In the notation of Parnas, ϵ may be considered a V-function. Then Example 1 is a restatement of a Parnas module for SET. However, it is often impossible for the V-functions alone to specify the data type, i.e. the notion of equality can not be defined solely on the basis of given V-functions. Equivalently, V-function values are insufficient to reconstruct the data object. Characteristic functions can be viewed as a generalization of V-functions which alleviate this difficulty. Next example illustrates these ideas on a STACK.

Example 2:

Data type: STACK (of elements of type T)

Operations:

INIT:	$\rightarrow$ STACK
PUSH:	T x STACK $\rightarrow$ STACK
POP:	STACK $\rightarrow$ STACK
TOP:	STACK $\rightarrow$ T
NULL:	STACK $\rightarrow$ boolean

V-functions: TOP, NULL

It is impossible to uniquely reconstruct a stack S given only TOP(S) and NULL(S).

We introduce two characteristic functions:

DEPTH(S), returns the length of stack S

$\epsilon(i,S)$, returns the i^{th} element from top,

$1 \le i \le \text{DEPTH}(S)$.

Note that the above description of meanings of DEPTH, ϵ are for intuitive understanding only. It may be seen that a stack S can be uniquely reconstructed given DEPTH(S) and $\epsilon(i,S)$, $1 \le i \le \text{DEPTH}(S)$.

Definition of Equality:

$$S_1 = S_2 \iff \text{DEPTH}(S_1) = \text{DEPTH}(S_2) \wedge \epsilon(i,S_1) = \epsilon(i,S_2),\ \forall i$$
$$1 \le i \le \text{DEPTH}(S_1).$$

<u>Operation</u>	<u>Semantic</u>	<u>Remark</u>
INIT(S)	$DEPTH(S) = 0$	
PUSH(x,S)	$DEPTH(S) = DEPTH(S') + 1$	S' denotes the stack before
	$\epsilon(1,S) = x$	the operation.
	$\epsilon(i+1,S) = \epsilon(i,S')$,	
	$\quad 1 \le i \le DEPTH(S')$	
POP(S)	<u>if</u> $DEPTH(S') > 0$ <u>then</u>	Specification says nothing
	$\quad DEPTH(S) = DEPTH(S') - 1$	regarding popping off an
	$\quad \epsilon(i,S) = \epsilon(i+1,S')$,	empty stack, thus we may
	$\qquad 1 \le i \le DEPTH(S)$	assume nothing about the
	<u>endif</u>	resulting stack.
TOP(S)	<u>if</u> $DEPTH(S') > 0$ <u>then</u>	
	$\quad TOP(S) = \epsilon(1,S)$	
	<u>endif</u>	
NULL(S)	$NULL = (DEPTH(S') = 0)$	$\square$

3. Description of the Method

The method of specification may then be summarized as follows.

1. Choose a set of characteristic functions. Intuitively, these should be chosen in such a manner that any instance of the given data type can be uniquely reconstructed given the values of characteristic function; conversely, characteristic function values must be defined for each valid instance of the data type. Note that no particular meaning is attached to the characteristic functions. Thus we need not assume any specific property nor verify that such properties are preserved. We restrict the characteristic function values to be of some simpler type for which equality (and other axioms) already exist. It is however possible to let characteristic functions return values of the same type, which is being defined; this line of approach will not be persued in this paper.

2. Define equality on the given data type, in terms of characteristic functions. This rigorously specifies the condition under which two instances of the data type may be considered equal; this is necessary for verification purposes as well as for informal validation that the chosen characteristic functions are sufficient. Note that the definition of equality may make use of equality on the element types (such as T) out of which this data type is created.

3. Define syntax of each operation, by specifying the domain and the range. In terms of implementation, this specifies those properties of the input, output parameters, which can be checked by a compiler.

4. Define semantic of each operation by specifying changes in the values of

characteristic functions. This may follow the notations given in Examples 1 and 2. A somewhat different notation is suggested in a later section, which is useful for proving facts about data and stating the error conditions. The effects section may be written as a set of assertions on characteristic function values, using conditionals and recursion. Furthermore, the use of new functions, defined in terms of characteristic functions is allowed.

There are two different ways of looking at the proposed scheme. Characteristic functions may be viewed solely as functions and each operation is shown to modify or extract information from the function values. Alternately, it may be imagined that the characteristic functions provide a basis for the representation of the data type and each operation modifies the representation. For instance in Example 1, we may imagine that every set is represented by an (infinite) boolean Vector ϵ ; $\epsilon(i,S)$ represents the value of the i^{th} component. Every operation either accesses or modifies the values of some elements of the vector. Example 2 illustrates that an unbounded number of elements may be modified by a single operation. This view of data is especially useful for an intuitive understanding, though it provides little clue for an efficient implementation. This latter property is required if we have to construct representation - independent specifications.

The proposed method, we believe, meets the criteria for specification set forth in the introduction. This scheme can be made as precise and rigorous as the axiomatic approach, by choosing a proper notation. Furthermore, it is <u>almost</u> representation independent as is the axiomatic scheme. However, the proposed scheme is much more intuitively appealing, since it makes use of an implicit model of the data. Axiomatic approach requires considerable care to produce a complete and consistent specification. Characteristic functions are usually simple to invent; if a particular choice is incomplete, it becomes evident during specification. We suspect that programmers tend to think in terms of representations, which may not be realizable on a computer. Finally, the mutual reference problem has been avoided. This helps us to study each operation alone at a time.

V-functions of state-machine approach are always inadequate when they return a finite number of values from an (a priori) unbounded object. Characteristic functions, though finite is number, have usually proper arguments to enable them to return all possible information about the data object.

The abstract model approach is similar to the proposed scheme, if we view the characteristic functions as a means of representation. However, the former approach involves a finite representation using a specific data structure for representation (such as tuples, sequences, etc.). Hence a proof by this method requires axioms regarding the underlying representation, which is avoided in the proposed scheme.

4. Notational aspects

We propose a notation below, which we have found convenient both for description of and proofs about data types. This notation is quite similar to that of Parnas except for the implicit description of error conditions.

(1) Description of Module:

Name of data type being defined, types of elementary objects, characteristic functions, statement of equality condition.

(2) Description of operations:

This has three parts for each operation.

(i) Name and type of the function (if it returns a value) and types of arguments.

(ii) Input specification in terms of characteristic function values. This specifies a condition for valid inputs to this operation. All other inputs result in an error condition, which is not explicitly stated. We will sometimes use assertions of the type (S = S'), which simply denotes that S' is the data object before the operation started.

(iii) Specifiation of output in terms of the effect on the values of characteristic functions. Conditionals, recursion, and definition of new functions using characteristic functions are permitted.

It is straightforward to mechanically generate preconditions and postconditions, for an axiomatic definition of each operation, from the above description. In particular, all the unchanged values of characteristic functions will have to be mentioned.

5. Some examples:

We give two examples to illustrate the various aspects of the proposed scheme. One of the examples is considered in a later section in connection with proofs of implementation.

Example 3 [1]

Symbol table for a block structured language is being defined. There are operations to enter a new block, leave an old block and retrieve the attribute of a symbol. In the following, LEVEL denotes the depth of nesting and GET gives the attribute of a given symbol from a given level (UNDEFINED, if the symbol is not defined at that level).

Data type: SYMBOL TABLE (of element type ID and attributes from ATT U {UNDEFINED}).

Characteristic function: LEVEL: SYMBOL TABLE $\rightarrow$ INTEGER

GET: SYMBOL TABLE x INTEGER x ID $\rightarrow$ ATT U {UNDEFINED}

Equality condition:

S_1, S_2: SYMBOL TABLE, id: ID;

$S_1 = S_2 \Leftrightarrow \text{LEVEL}(S_1) = \text{LEVEL}(S_2)$,

$\text{GET}(S_1, j, id) = \text{GET}(S_2, j, id) \quad 1 \le j \le \text{LEVEL}(S), \forall id.$

```
Procedure  INIT(S:SYMBOL TABLE);

     input spec.:

     output spec.:  LEVEL(S) = 1;

                    GET(S,1,id) = UNDEFINED,∀id:ID


procedure  ENTERBLOCK(S:SYMBOL TABLE);

     input spec.:  S = S';

     output spec.:  LEVEL(S) = LEVEL(S') + 1;

                    GET(S,LEVEL(S),id) = UNDEFINED, ∀id:ID;

                    {GET(S,t,id) = GET(S',t,id), ∀id:ID, 1 ≤ t ≤ LEVEL(S');

                     is implicit}.


procedure  ADDID(S:SYMBOL TABLE, id:ID, att:ATT);

     input spec.:

     output spec.:  GET(S,LEVEL(S),id) = att;

                    {In case of multiple definition of an identifier,

                     only the last definition is retained}.


procedure  LEAVE BLOCK(S:SYMBOL TABLE);

     input spec.:  LEVEL(S) > 1, S = S';

     output spec.:  LEVEL(S) = LEVEL(S') - 1;


function  ISINBLOCK(S:SYMBOL TABLE, id:ID): boolean;

     input spec.:

     output spec.:  ISINBLOCK = (GET(S,LEVEL(S),id) ≠ UNDEFINED);
```

```
function  RETRIEVE(S:SYMBOL TABLE, id:ID): ATT U{UNDEFINED};

     input spec.:

     output spec.:

                    define, find (S,t,id) : {t ≤ LEVEL(S)}

                                   if t = 1 then GET(S,1,id)

                                   else if GET(S,t,id) = UNDEFINED

                                        then find (S,t-1,id) else GET(S,t,id);

     RETRIEVE = find (S,LEVEL(S),id);                                        □
```

Example 4

A data type sorted list (SL) in ascending order is being defined. There are operations to initialize, insert a new element and search for the position of a given element in the list. Two characteristic functions give the length of the list and the value of the element at a given position.

Data type: SL (of integer)

Characteristic functions:

LENGTH: SL $\rightarrow$ INTEGER;

GET: INTEGER x SL $\rightarrow$ INTEGER

Equality condition:

S_1, S_2: SL

$S_1 = S_2 \iff \text{LENGTH}(S_1) = \text{LENGTH}(S_2) \wedge$

$\text{GET}(j, S_1) = \text{GET}(j, S_2),\ 1 \le j \le \text{LENGTH}(S_1);$

```
procedure  INIT(S:SL);
     input spec.:
     output spec.:  LENGTH(S) = 0;
```

<u>procedure</u> INSERT(x:INTEGER,S:SL);

input spec.: $S = S'$; $GET(j,S) \neq x,\ 1 \leq j \leq LENGTH(S)$;

output spec.:

$$LENGTH(S) = LENGTH(S') + 1,$$

$$LENGTH(S') = 0 \Rightarrow GET(1,S) = x,$$

$$x < GET(1,S') \Rightarrow GET(1,S) = x \wedge GET(i+1,S) = GET(i,S');$$
$$1 \leq i \leq LENGTH(S'),$$

$$x > GET(LENGTH(S'),S') \Rightarrow GET(LENGTH(S),S) = x;$$

$$GET(i,S') < x < GET(i+1,S') \Rightarrow GET(j,S) = GET(j,S'),$$
$$1 \leq j \leq i;$$

$$GET(i+1,S) = x;$$

$$GET(j+1,S) = GET(j,S'),\ i < j \leq LENGTH(S');$$

<u>procedure</u> SEARCH(x:INTEGER,S:SL,p:INTEGER)

input spec.: $LENGTH(S) > 0$, $GET(j,S) = x \quad \exists j,\ 1 \leq j \leq LENGTH(S)$;

output spec.: $p = j$;

Note that SEARCH could have been used as a characteristic function, in place of GET. □

6. Existence of characteristic functions

Two theoretical questions in connection with the proposed method are,

(1) whether there always exist a finite set of characteristic functions;

(2) whether the effect of any operation can be described in a finite manner in terms of a set of characteristic functions.

These questions can be answered by appealing to a representation which is a finite description in terms of a finite number of operations. We sketch an alternate proof of (i), which we have found to be a useful intuitive basis for construction of characteristic functions.

Suppose that the value returning functions are inadequate to describe the effect of the operations $F_1, F_2, \ldots F_r$. For instance, POP is the only operation in case of stack, whose effect on TOP and NULL can not be described. We then consider the following expressions, whose values must be possible to get from characteristic functions.

$$\mathrm{TOP}(\underbrace{\mathrm{POP}(\mathrm{POP}\ldots\mathrm{POP}(S)}_{i\,-\,1\text{ times}}\ldots))$$

$$\mathrm{NULL}(\underbrace{\mathrm{POP}(\mathrm{POP}\ldots\mathrm{POP}(S)}_{i\,-\,1\text{ times}}\ldots))$$

In general, we must consider all possible expressions consisting of F_i's in all possible manners. Now if we define characteristic functions,

$$E(i,S) = \mathrm{TOP}(\underbrace{\mathrm{POP}(\mathrm{POP}\ldots\mathrm{POP}(S)}_{i\,-\,1\text{ times}}\ldots))$$

$$N(i,S) = \mathrm{NULL}(\underbrace{\mathrm{POP}(\mathrm{POP}\ldots\mathrm{POP}(S)}_{i\,-\,1\text{ times}}\ldots)) = (\mathrm{DEPTH} = i - 1)$$

then we can capture the values of all such expressions. In general with several functions F_i, the arguments of characteristic functions should provide for expressing all possible combinations of these functions. Then a characteristic function is a super function whose argument is a sequence of

{1..r} which denotes the order in which the functions have been applied.

7. Some aspects of proofs about data

We consider the notion of data invariant introduced by Hoare [3]. We propose rules for verification of data invariant which in our case is a proposition over the values of characteristic functions. Data invariant is used as a basis for proving theorems about the data. Next, we introduce the notion of data descriptor, a proposition that is true for all and only those values of characteristic functions which represent valid data instances. We show that a descriptor is the strongest data invariant. A characterization for alternate set of characteristic functions is given in terms of data descriptor.

7.1 Data invariant

A data invariant is a proposition over the values of characteristic functions which is true of all values of characteristic functions representing valid instances of data. Note that the data invariant may be true for some values of characteristic function, which do not represent any valid instance of data.

Proof of a data invariant I can be accomplished as follows. First, we show that the initializing operation creates values for which I is true; next, we show that assuming I and the input specs. of an operation F, output specs. of F imply I. Sometimes, this proof will be impossible if I is a "weak" invariant -- in such a case, a stronger invariant I^* needs to be invented, such that $I^* \Rightarrow I$ and the invariance of I^* can be proven.

Data invariants are useful for proving theorems about data in general, since in addition to the input specification of an operation F, we can assume that I is true on input.

Example 5 (Contd. from Example 4)

We show that

$\text{LENGTH}(S) \geq 0$ is a data invariant.

Then we need to prove the following theorems.

(i) $T\{\text{INIT}(S)\}\text{LENGTH}(S) \geq 0$

(ii) $\text{LENGTH}(S) \geq 0 \wedge S = S' \wedge \text{GET}(j,S') \neq x,\ 1 \leq j \leq \text{LENGTH}(S)$

$\{\text{INSERT}(x,S)\}$

$\text{LENGTH}(S) \geq 0$

(iii) $\text{LENGTH}(S) \geq 0 \wedge S = S' \wedge \text{GET}(j,S') = x,\ \exists j,\ 1 \leq j \leq \text{LENGTH}(S)$

$\{\text{SEARCH}(x,S,p)\}$

$\text{LENGTH}(S) \geq 0.$

Each of the above theorems follows from the output specifications of the corresponding operations. A somewhat harder invariant to prove is

$\text{GET}(i,S) < \text{GET}(i+1,S),\ \forall i,\ 1 \leq j < \text{LENGTH}(S).$

The proof reduces to showing that the output specification of INSERT, for each of the four possible cases, preserve this invariant. □

7.2 Proofs of theorems about data

Theorems about data can be proven by successively considering the output specifications of each operation. The method is straightforward and is illustrated with an example.

<u>Example 6</u> (Contd. from Examples 4 and 5)

The following theorem states that the position of an element remains unchanged following the insertion of a larger element.

$$x < y,\ x = get(k,S),\ \exists k,\ 1 \le k \le LENGTH(S)$$
$$y \ne get(i,S),\ \forall i,\ 1 \le i \le LENGTH(S)$$

```
{INSERT (y,S);
 SEARCH (x,S,p)}
p = k
```

The theorem can be proven by constructing the output specification following INSERT.

We need the following data invariant, which is not explicitly stated in the following proof.

$LENGTH(S) \ge 0 \ \wedge \forall i,\ 1 \le i < LENGTH(S),\ GET(i,S) < GET(i+1,S).$

Now, $x < y,\ x = GET(k,S),\ y \ne GET(i,S),\ S = S',\ \exists k,\ 1 \le k \le LENGTH(S)$
$\forall i,\ 1 \le i \le LENGTH(S)$

{ INSERT (y,S) }

$LENGTH(S) = LENGTH(S') + 1 \ \wedge$

$[y > GET(LENGTH(S'),S') \Rightarrow GET(i,S) = GET(i,S'),\ 1 \le i \le LENGTH(S') \ \wedge$
$GET(LENGTH(S),S) = y] \ \wedge$

$[GET(i,S') < y < GET(i+1,S') = \ GET(j,S) = GET(j,S'),\ \forall j,\ 1 \le j \le i$
$GET(i+1,S) = y$
$GET(j+1,S) = GET(j,S'),\ i < j \le LENGTH(S')]$

$$\Downarrow$$

$LENGTH(S) = LENGTH(S') + 1;$
$x = GET(k,S);$

{SEARCH (x,S,p)}

$p = k$ □

We note that often it would be necessary to assume the truth of some particular data invariant in the precondition of an operation.

7.3 Data descriptor and alternate specifications

A data descriptor is a proposition on the values of characteristic functions, which is true of all and only those values which represent valid instances of data. It follows that every data descriptor is a data invariant though not conversely.

Example 7 (Contd. from Example 4)

$\text{LENGTH}(S) \geq 0$ is a data invariant.

However, we also require for a valid data instance that,

$\text{GET}(i,S) < \text{GET}(i+1,S) \quad \forall i,\ 1 \leq i < \text{LENGTH}(S).$ □

We say that a proposition P is stronger than Q if P => Q; in such a case Q is weaker than P. It may be noticed that if I_1 and I_2 are data invariants then so are

$$I_1 \wedge I_2, \; I_1 \vee I_2.$$

The next theorem shows that a data descriptor D is a strongest data invariant. An important part of the theorem is to show that the set of characteristic function values satisfying D is nonnull, i.e. the data invariants are consistent, provided the output specifications are.

<u>Theorem 1</u>:

Suppose that the input, output specifications are consistent, i.e. there are values of characteristic functions satisfying both input and output. Then the data descriptor D is the strongest invariant.

Sketch of a Proof:

Clearly D is an invariant, since every operation gives us a valid data instance. Conversely, we show that for any other invariant I, D => I. If not, then there exists a set of characteristic function values v for which,

$D(v) \wedge \neg I(v)$.

This contradicts the fact that I is true of all valid data instances.

Next step is to show that $D \neq$ <u>false</u>. Clearly D represents the set of all valid data instances, in particular including the initial set of values specified by the INIT operation. Hence, if all input output specifications are consistent then D represents a nonnull set of values. □

<u>Example 8</u> (Example 4 contd.)

$\text{LENGTH}(S) \geq 0 \wedge \text{GET}(i,S) < \text{GET}(i+1,S)\ \forall i,\ 1 \leq i < \text{LENGTH}(S)$ can be shown to be a data descriptor. □

It is usually simple to come up with a data descriptor; however, it is often difficult to show that a given proposition is indeed a descriptor.

First, it should be shown that D is an invariant. Next, we may use induction on some partial order on the data objects (in terms of characteristic function values which satisfy the descriptor). In example 8, a partial order might be

$$\begin{aligned} S < S' \Rightarrow\ & \text{LENGTH}(S) < \text{LENGTH}(S')\ \underline{\text{and}} \\ & \text{for every } i,\ 1 \le i \le \text{LENGTH}(S), \text{ there exists } j \\ & i \le j \le \text{LENGTH}(S') \text{ such that,} \\ & \text{GET}(i,S) = \text{GET}(j,S'). \end{aligned}$$

The partial order should have as the smallest element INIT(S), which is true in this case.

It should now be shown that all the abstract objects satisfying D could be conceivably created starting with INIT(S). It is sufficient to consider only functions such as INSERT which create larger objects (larger being with respect to <). Thus it should be shown that any given object S could be created by INSERT starting with some smaller object.

Descriptors could be used to provide conditions for implementation and alternate characteristic functions. We do not persue this line of research here, since finding a descriptor seems to be a difficult practical problem. We state an obvious theorem on alternate specifications using different characteristic functions.

Theorem 2:

Let D_1, D_2 be two descriptors corresponding to alternate specifications of the same data object, using different characteristic functions. Furthermore, let the equality definition in both specifications be identical; i.e. $S_1 = S_2$ in one specification if and only if $S_1 = S_2$ in the other specification. Then there exists a 1:1 and unto mapping between the set of values satisfying D_1 and those satisfying D_2.

8. Proving implementations

Hoare[3] has proposed a scheme for proving data representation, in which the concrete representations are mapped to abstract objects through some mapping function. Then the proof of implementation reduces to showing that the operations transform the abstract data in the manner required in the specification.

We adapt this scheme to apply to characteristic functions. We furthermore show that the proof complexity can be considerably reduced by axiomatizing the characteristic functions in the domain of representation. A rather complex example is given to illustrate these ideas.

The intuitive idea behind the method can be explained as follows, with reference to example 4. Given a particular implementation of the operations INIT, INSERT and SEARCH, we are required to show that the implementations meet the specifications. In order to do that, we define the characteristic functions LENGTH and GET in terms of the representation. For instance, if the sorted list is represented by an ordered linked list with a pointer, then we may define LENGTH as a function which returns the length of this linked list and GET as a procedure which retrieves the element from a given position. There is, at this stage, no restriction on what functions we may choose for LENGTH and GET. Intuitively of course we should choose them to denote whatever the characteristic functions were supposed to denote.

At the next step, it is necessary to show that assuming input specifications are true for an operation in terms of the defined characteristic function, then output specifications will also hold, again in terms of the definitions of the characteristic function.

An important practical difficulty, which will become evident in the following example, is that a procedural definition of the characteristic functions would lead to an unmanageable problem of verification. It is found to be much easier to verify, when the characteristic functions are defined axiomatically;

for instance, we may say that insertion of an element in the ordered linked list, increases the LENGTH by 1. A set of such axioms can be written down showing the effects of each primitive operation in the representation, on the characteristic functions. Then the method of proof may be summarized as follows:

1. Define a set of axioms, which denote the effect of executing the primitive operations in the representation, on the values of characteristic functions.
2. Prove the implementation of an operation by assuming that the corresponding input specifications hold (plus any implementation invariant) and showing that on termination, the values of characteristic functions imply the output specification.

An example in the next section illustrates these ideas. We should remark here that the proof in most cases turns out to be extremely long and laborious. It can be more easily accomplished by proving several levels of representation, each one being a refinement of the previous one.

9. Examples of Proofs of Implementation

We consider the implementation of a stack by a one-way linked list and implementation of a sorted list by a binary search tree. The first problem is simple; the second one invloves complex update mechanisms. In each case, a sketch of a proof is given by specifying the effects of primitive operations in the representation on the characteristic functions values. An important step, which has not been proven, is to show that the above axioms are consistent. Currently, we have no simple method for such a proof except noting that the axioms will be consistent, if each axiom involves a single, primitive, independent operation in the representation.

Programs are written in a Pascal-like notation: new(p) reserves an area in memory and makes pointer p point to that area; p↑ designates the data to which pointer p is currently pointing; Var p:↑node, denotes that p is a variable of type pointer which always points to an object of type node; p↑.v represents the v-component of a record that p is pointing to.

Example 9 (Example 2 contd.)

A stack will be represented by a one-way linked list of nodes; t will be pointer to the top node of the stack.

```
module    STACK (of type T)
    type  node = record
                   elem: T;
                   next: ↑node
                 end;
    Var t: ↑node;
    Proc init;
    begin
        t:= nil
    end init;
  proc push (x:T);
        Var p: ↑node;
        begin
                new(p);
                p↑.elem:= x;
                p↑.next:= t;
                t:= p
        end;
  proc pop;
        begin
                if t ≠ nil then t:= t↑.next
        end;
  function  top: T;
        begin
                if t ≠ nil then top:= t↑.elem
        end;
```

```
function null: boolean;
        begin
                null:= (t = nil)
        end;
```

The following axioms about DEPTH and ϵ are sufficient to prove the implementation.

1. $t = nil \iff DEPTH = 0$
2. $DEPTH = k,\ p\uparrow.next = t\ \{t := p\}\ DEPTH = k + 1$
3. $x = \epsilon(i),\ p\uparrow.next = t\ \{t := p\}\ x = \epsilon(i + 1),\ t\uparrow.elem = \epsilon(1)$
4. $DEPTH = k,\ t \neq nil\ \{t := t\uparrow.next\}\ DEPTH = k - 1$
5. $x = \epsilon(i),\ i > 1,\ t \neq nil\ \{t := t\uparrow.next\}\ x = \epsilon(i - 1)$ □

The above example is somewhat misleading in the sense that the abstract object, characteristic functions and the implementation are all alike and represent similar ideas. Next example illustrates a rather complex implementation.

Example 10 (Example 4 contd.)

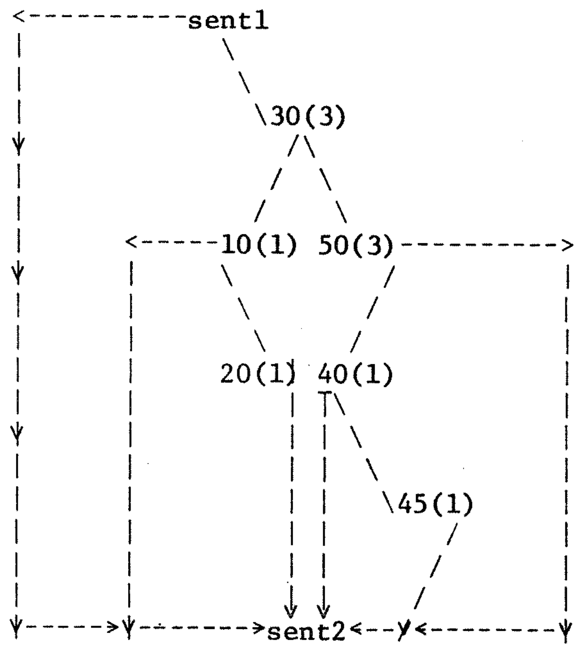
We represent a sorted list by a binary search tree, using an idea essentially due to Crane [4]. The binary search tree has the following structure. Every node in the tree has a value v and a size s associated with it. v represents some element of the sorted list and s denotes the number of nodes in the left subtree of this node plus 1. For any node with value v, the values in its left subtree are all smaller than v and those in its right subtree are all larger than v.

The following tree is a valid representation of the sorted list (10 20 30 40 45 50).

```
          30(3)
           /\
          /  \
     10(1)    50(3)
       \        /
        \      /
      20(1) 40(1)
              \
               \
                \45(1)
```

The number outside brackets represents the value and the one inside the brackets represents the size.

Such a representation admits of efficient implementation of the operations. We first introduce two sentinels (sent1, sent2), which are really dummy nodes, designed to make sure that the tree has at least 2 nodes at any time. Sent1 behaves like the root; its right son is the actual root; sent2 behaves as a sink; every left and right link not pointing to any node, point to sent2. The values and sizes associated with sentinels are irrelevant.

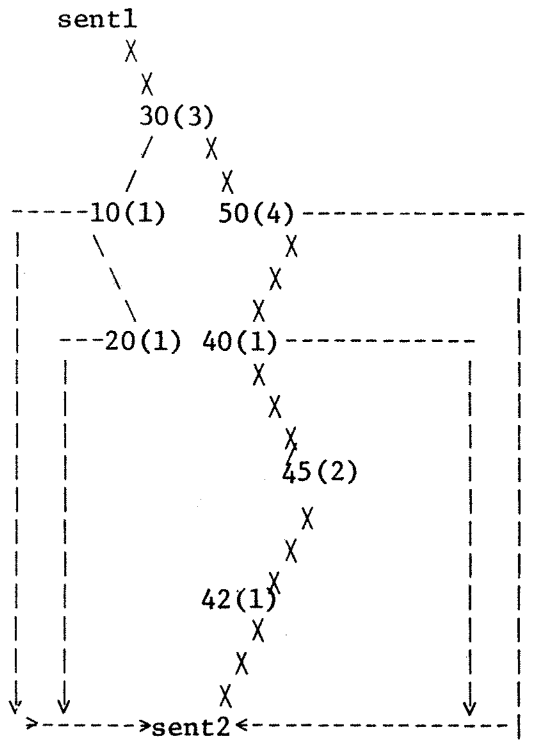


INIT creates both sentinels and links them properly. INSERT works as follows. We initiate a binary tree search starting from the right son of sent1, with the input value x. If x is smaller than the values at this node, we proceed left, otherwise we proceed right. Whenever we proceed left, we increase the size of that node, since the value x would ultimately be inserted somewhere in the left subtree of that node. We stop on reaching the sentinel sent2 and insert x as a new node and link it properly to the last node seen. SEARCH for a value a proceeds similarly, however whenever a right branch is taken from a node with size s, s is added to a running sum. Finally the size of the node with value x is added to this sum, which gives the position of x.

Consider the effect of

INSERT(42).

The path taken is cross hatched. Note that the size of every node, wherever a left branch was taken is increased by 1.



Next consider the effect of

SEARCH(45).

The sequence of nodes where right branches were taken are

30 with size 3

40 with size 1

45 has a size of 2.

Hence the position of 45 = 6.

A Pascal-like program implementing this structure is given below.

```
module  SL
     type node = record
                    v: integer;
                    left, right: ↑node;
                    size: integer
                 end;
     Var sent1, sent2: ↑node;
     Proc init;
     begin
          new (sent1); new (sent2);
          sent1↑.left:= sent2; sent1↑.right:= sent2;
          sent1↑.v:= 0
     end init;
```

```
Proc  insert (x: integer);
    Var p, q, r: ↑node;
    begin
        sent2↑.v:= x; q:= sent1; p:= q↑.right;
        {sent2 holds x, so that we are assured x is in the tree;
         q is the father of p}
        while p ≠ sent2 do
        begin
            q:= p;
            if x < p↑.v then begin
                                 p↑.size:= p↑.size + 1;
                                 p:= p↑. left
            else
                p:= p↑.right
        end;
        new(r);
        r↑.v:= x; r↑.left:= sent2; r↑.right:= sent2; r↑.size:= 1;
        if x < q↑.v then q↑.left:= r else q↑.right:= r;
    end insert;
```

```
Proc  Search (x: integer, pos: integer);
     Var p: ↑node;
     begin
          pos:= 0; p:= sentl↑.right;
          while p↑.v ≠ x do
          begin
               if x < p↑.v then p:= p↑.left
               else begin
                    pos:= pos + p↑.size
                    p:= p↑.right
               end
          end;
          pos:= pos + p↑.size
     end search
end  SL;
```

LENGTH can be defined as a function whose value is the number of nodes in the tree. Similarly, GET(i,x) can be defined to return the value v at the root, if the size s at the root equals i, else if i $<$ s then apply GET(i,x) to the left subtree of the root, otherwise (i $>$ s) apply GET(i-s,x) to the right subtree of the root. These procedures can be defined in the same language, used to describe the representation.

It however becomes extremely difficult to verify the implementation with such a definition of LENGTH and GET. Instead, we consider the primitive operations in the representation and describe their effects on LENGTH and GET.

We need the following axioms about LENGTH.

1. Sent1↑. right = sent2 <=> LENGTH = 0
2. for p, a pointer to any node in the tree p $\neq$ sent2, LENGTH = k, p↑.left = sent2, r $\neq$ sent2 {p↑.left:= r} LENGTH = k + 1
3. LENGTH = k, p↑.right = sent2, r $\neq$ sent2{p↑.right:= r} LENGTH = k + 1
4. All other primitive operations keep LENGTH invariant.

We need one more axiom about new.

$\text{sent2} \neq \text{nil}\ \{\text{new}(r)\}\ r \neq \text{sent2}.$

It can now be shown that

$T\ \{\text{init}\}\ \text{LENGTH} = 0$

$\text{LENGTH} = k\ \{\text{insert}\ (x)\}\ \text{LENGTH} = k + 1$

The first theorem follows by using axiom 1. The second follows by using for insert,

input spec.: $\text{sent1}\uparrow.v = 0,\ x > 0,\ \text{LENGTH} = k$

loop invariant: $[x < q\uparrow.v \Rightarrow p = q\uparrow.\text{left}] \wedge [x > q\uparrow.v \Rightarrow p = q\uparrow.\text{right}]$

Then axioms 2, 3 can be applied to show that since a link (left/right) of q that previously pointed to sent2 now points to r, $r \neq \text{sent2}$, LENGTH is increased by 1. Note that termination of the loop needs to be proven.

It follows by axiom 4 that LENGTH remains invariant as a result of search.

We next prove the theorems about the effect of **insert** on GET. A sketch of proof will be given. We need the following axioms: $t \in \text{rsub}(p)$ denotes that t is a pointer to a node in the right subtree of p or $t = p$.

1. $GET(0) = 0$

2. $GET(i) = x, x = t\uparrow.v, t \in rsub(p)$

   $\{p\uparrow.size := p\uparrow.size + 1\}$

   $GET(i + 1) = x$

3. $GET(i) = x, x = t\uparrow.v, t \notin rsub(p)$

   $\{p\uparrow.size := p\uparrow.size + 1\}$

   $GET(i) = x$

4. $GET(i) = q\uparrow.v, q\uparrow.left = sent2, r \neq sent2, GET(j) = y$

   $\{q\uparrow.left := r\}$

   $GET(i - 1) = r\uparrow.v, GET(i) = q\uparrow.v, GET(j) = y, j \neq i - 1$

5. $GET(i) = q\uparrow.v, q\uparrow.right = sent2, r \neq sent2, GET(j) = y$

   $\{q\uparrow.right := r\}$

   $GET(i + 1) = r\uparrow.v, GET(i) = q\uparrow.v, GET(j) = y, j \neq i + 1$

Four theorems need to be proven corresponding to insertion into an empty list, insertion of a smallest element, insertion of a largest element and insertion of an element which is neither largest nor smallest. The loop invariant would basically say that all nodes not in the subtree of p (or equal to p), have already attained their final positions. The remaining portion of the program following the loop, would either use axiom (4) or (5) above to show that the inserted element has the correct position.

In addition to the axioms, several invariants on implementation have been used implicitly: all values in the tree are distinct; size is equal to the number of nodes in the left subtree plus 1 etc. Furthermore termination needs to be proven along with every partial correctness proof. □

The axioms themselves need to be proven consistent. This can be done by suggesting implementations for each of the characteristic functions and proving that the axioms are satisfied. This however has turned out to be a fairly difficult problem. If every axiom involves an independent, primitive operation of the representation, they will be guaranteed consistent since they will not conflict with each other.

This example was worked out to illustrate the difficulty of a formal or semiformal proof, with the available tools. Hoare's mapping function becomes extremely difficult to state when the representation is considerably different from the implicit model used in the specification. One possible solution is to implement each characteristic function and then verify theorems of the form

$$\text{LENGTH} = k \ \{\text{insert } (x)\} \ \text{LENGTH} = k + 1,$$

by sequentially writing the three programs corresponding to LENGTH, insert, LENGTH (renaming local variables to avoid any naming conflict); attaching assertions LENGTH = k at the end of the first program and LENGTH = k + 1 at the end of the last program. Thus the problem has been converted to a general program proving problem. Needless to say, this technique buys nothing in terms of simplicity, intuitivity or improvement on the suggested methods.

We have found that the axiomatic approach, proposed here, is suitable in most cases arising in practice. It may justifiably be argued that the example studied was quite complex and must therefore have a complex verification process.

The only method for simplifying the verification process, we believe, is to consider several levels of implementation, each being a refinement of the previous one.

Conclusion and summary

A method of specification of data abstraction is suggested in this paper, using characteristic functions as a basis. Semantic of each operation on data is specified by describing its effect on the values of characteristic functions. Given a complete set of characteristic functions, a rigorous definition of equality can be given. The proposed method is shown to be both rigorous and intuitively appealing.

Some aspects of verification are considered with emphasis on proving theorems about data,independent of the representation. Data invariant, expressed in terms of values of characteristic functions, is an important concept in any such proof methodology. Finally, proofs of implementation are considered. It is suggested that the proof can be accomplished by stating certain axioms describing the effect of the primitive operations in the representation, on the values of characteristic functions. A sketch of a proof is presented for a complex implementation of a sorted list.

It has been our experience that specification should not only be rigorous but it should be intuitively justifiable, so that it can serve as an effective means of communication between the specifier, implementer and the user. Rigorous mechanisms, which defeat intuition, are often found to lead to errors in their usage. The proposed method, we believe, meets most of the objective criteria for good specification.

Acknowledgement: Author is indebted to Prof. R. T. Yeh, Univ. of Texas at Austin, for his enthusiasm and encouragement of this work.

References

1. Guttag, J.V., D. Musser, E. Horowitz, "Abstract Data Types and Software Validation," Information Sciences Institute, ISI/RR-76-48, August 1976.
2. Hoare, C.A.R, "An Axiomatic Basis for Computer Programming," Comm. ACM, Vol. 12, No. 10 (Oct. 1969), pp. 576-583.
3. Hoare, C.A.R., "Proof of Correctness of Data Representations," Acta Informatica, Vol. 1, No. 4 (1972), pp. 271-281.
4. Knuth, D.E., "The Art of Computer Programming, Vol. 3, Sorting and Searching," Addison-Wesley, 1973.
5. Liskov, B.H., S. Zilles, "Specification Techniques for Data Abstractions," IEEE Trans. On Software Engineering, SE-1, Vol. 1 (1975), pp. 7-19.
6. Liskov, B.H., V. Berzins, "An Appraisal of Program Specifications," unpublished manuscript.
7. Parnas, D.L., "A Technique for the Specification of Software Modules with Examples," Comm. ACM, Vol. 15, No. 5 (May 1972), pp. 330-336.
8. Robinson, L. et al, "On Attaining Reliable Software for a Secure Operating System," Proc. of Intl. Conf. on Reliable Software (1975), pp. 267-284.