A Value Transmission Method for Abstract Data Types

M. HERLIHY and B. LISKOV
Massachusetts Institute of Technology

Abstract data types have proved to be a useful technique for structuring systems. In large systems it is sometimes useful to have different regions of the system use different representations for the abstract data values. A technique is described for communicating abstract values between such regions. The method was developed for use in constructing distributed systems, where the regions exist at different computers and the values are communicated over a network. The method defines a call-by-value semantics; it is also useful in nondistributed systems wherever call by value is the desired semantics. An important example of such a use is a repository, such as a file system, for storing long-lived data.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, input/output*; D.4.4 [**Operating Systems**]: Communications Management—*message sending*; H.2.5 [**Database Management**]: Heterogeneous Databases—*data translation*

General Terms: Design, Languages

Additional Key Words and Phrases: Message communications, call by value, long-term storage

1. INTRODUCTION

Abstract data types have proved to be a useful technique for structuring systems. They permit the programmer to encapsulate details of the representation of data so that these details can be changed with minimal impact on the program as a whole. It is sometimes useful, especially in large programs, to use different implementations of a data abstraction in different regions of the program. Current languages that support data abstraction, however, limit a program to a single implementation [11, 17]. The reason for this limitation is the difficulty of communicating between the regions using different implementations.

This paper describes a technique for communicating abstract values between such regions. The method defines a call-by-value semantics. The method was developed for use in distributed systems, where the regions exist at different computers and the values are communicated over the network. It is also useful in

This work was supported by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-0661.

Authors' address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

nondistributed systems wherever call-by-value is the desired semantics. An important example of such a use is a repository, such as a file system, for storing long-lived data.

We assume a hardware base consisting of a set of computers, or *nodes*, connected by a network; the only (physically) shared memory between the nodes is the network itself. A distributed program consists of a collection of program *modules* that reside at different nodes and share no memory. A module's address space is a collection of strongly typed *objects*, which are containers for values. The modules communicate by means of a primitive, provided by the language, that causes messages to be transferred over the network.

For purposes of discussion let us assume that messages are sent by executing

$$\textbf{send } C(a_1, \ldots, a_n) \textbf{ to } M$$

Here M is a module, C is the name of a request or operation that M is being asked to perform, and $a_1, \ldots, a_n$ are the actual arguments of operation C. The effect of executing such a **send** statement is that a message containing the name C and the values of $a_1, \ldots, a_n$ is constructed and sent to M. We assume that in M there is an operation definition for each request M is prepared to handle. Thus M contains an operation definition,

```
C = request handler(f1 : T1, . . ., fn : Tn)
    % body of C
    end C
```

When a message containing the request named C is received at M, this request handler is executed,[1] with the formals $f_1, \ldots, f_n$ initialized to contain the values extracted from the message.

In the remainder of this paper we will assume that all arguments of the **send** are transmitted by value. Therefore, when one of these arguments is of abstract type, it will be necessary to somehow "copy" the value of that argument to the receiving module. This paper addresses the question of how that copy is to be made. (Some kind of call by reference for **send** arguments, in which a reference to the argument is sent to the receiving module, is also possible, although we believe call by reference is not very useful in a distributed program. Such a semantics leads to a completely different set of problems, for example, storage management, local versus remote pointers [19], than those discussed in this paper.)

How can the value of an argument of abstract type be communicated in a message? Recall that an abstract type provides a set of abstract objects or values and a set of primitive operations: the primitive operations are the only means of manipulating the values [15]. An implementation of an abstract data type must define how to represent the values and define implementations for the operations in terms of the chosen representation. One possibility for communicating abstract values is to communicate their representations; that is, the system transmits the

[1] We are ignoring issues such as the reliability properties of the primitive, how M schedules the processing of incoming messages, and whether type checking of message and argument types is done at compile time or at run time.

representation in the message and automatically reconstructs the abstract value at the receiving module. Although such a representation may be defined in terms of other abstract data types, ultimately it is expressed in terms of the built-in types of the language. Thus the representation can be transmitted assuming (as we do) that the built-in types can be transmitted.

Nevertheless, transmitting abstract values by transmitting their representations is unsatisfactory for several reasons:

(1) Different modules may use different representations for values of common user-defined types. Security concerns, differing patterns of use, and differing hardware characteristics may all encourage the use of customized implementations of abstract types in modules. The direct transmission of underlying representations provides poor support for constructing customized representations for abstract types.

(2) The underlying representation of a value may be transmissible, while the value itself may not be. For example, it may not make sense to send a value of type "I/O stream" to another node, even if the representation of the stream is transmissible (e.g., an array of integers).

(3) Conversely, a value may be transmissible, while its representation may be unsuited for transmission. For example, a representation used by one implementation may contain information, such as an index into a private table, that would be meaningless to another implementation.

We conclude that what is needed is user control over transmission of abstract values. In the remainder of this paper we discuss a method that supports user control. This method satisfies the following design goals:

Modularity. Transmission of the values of a data abstraction is defined locally as part of the implementation of that data abstraction. This principle is necessary to ensure that knowledge of the representation is local to the implementation.

Usability. The definition of transmission can be given in terms of any convenient data types. The programmer does not need to translate into strings of bits. Furthermore, the programmer need not know about the underlying network and its protocols.

Linearity. The work needed to implement value transmission is linear in the number of different implementations for the data abstraction.

The transmission method is described in terms of the CLU programming language [11, 14] but is applicable to any language that supports data abstraction. We do assume a single language system: all communication is between modules written in the same language.

The method we propose is to define a canonical representation for each type that can be used in messages; each implementation of the type must define how to translate between its internal representation for the values and the canonical representation. Such a method is an obvious approach, but there are many details to be worked out. Our main contribution is that we have worked out these details, including the user interface, specification of transmission and verification of the translation functions, and implementation.

Section 2 describes the basic method in the absence of shared and cyclic data structures, while Section 3 extends the method to include sharing. Section 4 discusses the implementation of message transmission. Section 5 extends the method to handle cyclic objects, while Section 6 extends the method to support storage of long-lived data. Finally, Section 7 contains a summary and discussion of what we have accomplished; it also discusses the relationship of our work to other research, some ways of increasing the efficiency of our method, and areas for further research.

2. THE BASIC METHOD

In this section we describe our basic method of transmitting abstract values. We discuss the meaning of transmission and how to define transmission for an abstract type. We then give examples of the programs that users write to support transmission. Finally, we discuss how to reason about the correctness of the transmission programs. To simplify the discussion, we ignore the problems that arise in transmitting data structures containing shared components and cycles; solutions to these problems are given in later sections.

2.1 The Meaning of Transmission

Transmissibility is a property of a data abstraction and must be stated in the specification of that abstraction. A transmissible data type T can be thought of as having an additional operation,

$$\textit{transmit}: T \rightarrow T,$$

which is implicitly called during message transmission. Given an input data object, *transmit* produces a different data object, which even resides at a different node from the original, whose relationship to the input object is defined by the specification for *transmit.*

Although the exact specification of *transmit* is type dependent, most often what is wanted is that the values of the two objects be equal. (The equivalence relation for values of a data type is defined by the type's specification; see [5, 9].) A *transmit* operation with this property satisfies the relation

$$x = \textit{transmit}(x)$$

and is said to *preserve value equality.*

For the built-in types, the language defines whether they are transmissible and the meaning of transmissibility. Almost all the built-in CLU types are transmissible. In CLU, as in other languages, there are two classes of built-in types: *structured* and *unstructured.* Objects of structured types contain collections of other objects; arrays and records are examples of structured types. Objects of unstructured types contain no other objects; integers, characters, and real numbers are examples of unstructured types.

Transmit for almost all built-in, unstructured types in CLU does preserve value equality. The one exception is *transmit* for real numbers, which, because of round-off errors, does not preserve value equality but guarantees only that the two values differ by very little. Built-in structured types are transmissible if and only if the component types are transmissible. Transmission for structured types involves transmission of components using *transmit* for the component types. For example, if an object x is an array containing elements of type T, then *transmit* for x creates a new array object y with the same bounds as the original, and with elements

$$y[i] = T\$transmit(x[i]).$$

(Here we are using the notation $T\$transmit$ to indicate that the *transmit* operation for type T is being used.) Thus *transmit* for built-in structured types will preserve value equality if and only if *transmit* for the component types does.

For abstract types the specification of *transmit* is user defined. As with the built-in types, it is not always possible to require preservation of value equality. In a later section we present some examples of abstract types whose *transmit* operations do not preserve equality.

2.2 Implementing Transmission

The *transmit* operation is neither called directly in programs nor implemented directly. Instead, the system causes it to be executed whenever a successful message communication occurs. Suppose that an invocation of

$$C = \textbf{request handler } (f_1 : T_1, \ldots, f_n : T_n)$$

is run in response to

$$\textbf{send } C(a_1, \ldots, a_n) \textbf{ to } M$$

Then in this invocation of C, we have

$$f_i = T_i\$transmit(a_i)$$

for $i = 1, \ldots, n$.

Transmission for an abstract type is implemented by users in the following way. As was mentioned in the introduction, a canonical representation is defined for each transmissible type. This canonical representation is given by defining an *external representation type*. For every abstract type that is transmissible we require that an external representation type be defined. The external representation type of an abstract type T is any convenient transmissible type XT. This type can be another abstract type if desired; there is no requirement that XT be a built-in type. Intuitively, the meaning of the external representation is that values of type XT will be used in messages to represent values of type T. The choice of external representation type is made for the abstract type as a whole and is independent of any implementation of the abstract type.

Each implementation of the abstract data type T must provide two operations to map between (its internal representation of) values of abstract type, T, and external representation type, XT. There is an operation

$$encode = \textbf{proc}(t : T) \textbf{ returns } (XT)$$

to map from T values to XT values (for sending messages) and an operation

$$decode = \textbf{proc}(x : XT) \textbf{ returns } (T)$$

to map from XT values to T values (for receiving messages). Intuitively, the

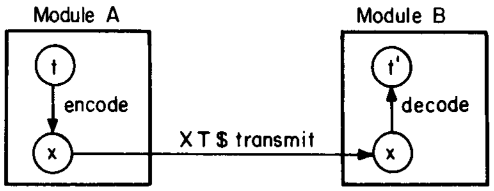

Fig. 1. Meaning of $T\$$*transmit*.

correctness requirement for *encode* and *decode* is that they preserve the abstract T values: *encode* maps a value of type T into the XT value that represents it, while *decode* performs the reverse mapping. More precisely, *encode, decode,* and *transmit* are related by the following identity. For $x \in T$,

$$T\$transmit(x) = decode(XT\$transmit(encode(x))).$$

This identity is illustrated in Figure 1.

Encode and *decode* are not called explicitly by user programs; instead, the language implementation makes these calls in the course of sending and receiving messages. Each actual argument in a **send** statement must be of a transmissible type. If this type is one of the nonstructured, built-in types like integers or characters, then the system knows how to place its value in a message and how to extract its value from a message. For built-in structured types, like arrays, the system will transmit the components, using *encode* and *decode* for the component type, as part of transmitting the structured object. For abstract types the system calls *encode* to obtain an XT object. If XT is an abstract type, then *encode* for XT is called, and this process is repeated until the argument has been translated to a value of built-in type. This value is then transmitted in the message. When a message is received, the reverse happens: a value of built-in type is extracted from the message, and then *decode* operations are called until a T value is obtained. (The language implementation is discussed in Section 4.)

2.3 Examples

As a first example of a typical user-defined type, we consider complex numbers. This type provides operations to create new complex numbers; add, subtract, multiply, and divide complex numbers; compare complex numbers; and obtain the real and imaginary coordinates of a complex number. Both rectangular and polar coordinates are useful representations for complex numbers. The choice of representation depends on the relative frequency of addition versus multiplication.

A good external representation for complex numbers might allow either coordinate system to be used. In CLU this would be expressed by giving type definitions, for example,

```
xrep = variant[xy:xycoords, polar:polarcoords]
xycoords = record[x, y:real]
polarcoords = record[rho, theta:real]
```

Here **xrep** is a reserved word that identifies the external representation type XT; it also stands (as do *xycoords* and *polarcoords*) as an abbreviation for the type

```
complex = transmissible cluster is create, real, imag, add, sub, mul, divide,
                                   equal, . . .
  xycoords = record[x, y : real]
  polarcoords = record[rho, theta : real]
  rep = xycoords
  xrep = variant[xy : xycoords, polar : polarcoords]
  encode = proc(c : cvt) returns (xrep)
    return(xrep$make_xy(c))  % Make an xy variant.
    end encode
  decode = proc(xc : xrep) returns (cvt)
    tagcase xc
      tag xy(x : xycoords): return(x)
      tag polar(p : polarcoords): r : real := p.rho * cos(p.theta)
                                  i : real := p.rho * sin(p.theta)
                                  % Use computed values to create a record.
                                  return(rep${x : r, y : i})
      end
    end decode
  % Definitions of procedures implementing the operations listed in the header appear
  % here.
end complex
```

Fig. 2. Complex number example.

appearing to the right of the equal sign. A *variant* is a built-in CLU type similar to a variant record; an object of this type can be either of type *xycoords*, in which case it is tagged by the identifier *xy*, or of type *polarcoords*, tagged by the identifier *polar*.

In CLU an abstract type is implemented by a special kind of module called a *cluster*. Figure 2 shows part of a cluster that implements complex numbers using rectangular coordinates as the internal representation (**rep**). Here *encode* obtains the internal representation of a complex number (via **cvt**)[2] and builds the external representation for this number, using the *xy* variant. *Decode* must check the tag of the external representation value it receives (using the **tagcase** statement) and do a conversion to rectangular coordinates if it receives the polar form. The internal representation it constructs turns into a complex number when it is returned (via **cvt**).

Transmission for the complex number type cannot preserve strict value equality because of round-off errors, although it is important to define the acceptable margin of error.

As a second example, we introduce a *key-item table* that stores pairs of values, where one value (the *key*) is used to retrieve the other (the *item*). The key-item table type has operations for creating empty tables, inserting pairs, retrieving the item paired with a given key, deleting pairs, and iterating through all key-item

[2] **Cvt** indicates a conversion between abstract and internal representation type. **Cvt** may appear only in a cluster; it maps between the **rep** of that cluster and the abstract type implemented by that cluster. When **cvt** appears as the type of an argument, the mapping is from abstract type to **rep** type; when it appears as a result type, the mapping is the other way.

```
data type table[key, item: type] is create, isin, enter, lookup, delete, allpairs
  requires Key and item are transmissible.
           Key$transmit preserves value equality.
           Key is totally ordered by operations equal and lt (less than).
  table is transmissible
    effect If t is a table and k is a key, then:
           isin(t, k) = isin(transmit(t), k)
           isin(t, k) implies lookup(transmit(t), k) = item$transmit(lookup(t, k))
  create = proc( ) returns (table)
    effect returns a new, empty table.
  isin = proc(t: table, k: key) returns (bool)
    effect returns true if there is an entry for k in t, else returns false.
  enter = proc (t: table, k: key, i: item)
    modifies t
    effect inserts the pair (k, i) in t, thus modifying t.
           If k is already in t, changes its associated item to i.
  lookup = proc(t: table, k: key) returns (item) signals (no_entry)
    effect if k is entered in t returns the associated item, else signals no_entry.
  delete = proc(t: table, k: key)
    modifies t
    effect removes the entry for k (if any) from t, thus modifying t.
  allpairs = iter(t: table) yields (record[k: key, i: item])
    effect yields all entries in t, each exactly once, in some arbitrary order.
end table
```

Fig. 3. Informal specification of table.

pairs. An informal specification of key-item table is shown in Figure 3. This type is parameterized by the types of both keys and items. As stated in the **requires** clause of the specification, certain restrictions are placed on these types, namely, both must be transmissible, and keys must be totally ordered. The operation *allpairs* is an iterator [11]; this is a limited kind of coroutine that can be called only by a **for** loop to provide results (in this case, key-item pairs) to its caller one at a time.

Tables can be implemented in many different ways, depending on local resources, or patterns of use. A straightforward choice of external representation of a key-item table is an array of key-item pairs, expressed in CLU by

xrep = *array*[*pair*]

pair = *record*[*k*: *key*, *i*: *item*]

In CLU, *array*[*T*] indicates an array whose elements are of type *T*. The size of CLU arrays can vary dynamically. When a new array is created it is empty, and new elements can be appended at either the high or the low end.

A partial implementation for table using a sorted binary tree representation is shown in Figure 4. Here *encode* and *decode* work on table objects (not on their representations) and make use of other table operations to do the translations. This representation makes use of the fact that pointers are implicit in CLU; in many other languages *nodes* would contain explicit pointers to tables.

```
table = transmissible cluster [key, item : type] is create, enter, seen, lookup, allpairs,
                                                     delete
  where key is transmissible,
        item is transmissible,
        key has lt : proctype(key, key) returns (bool),
                equal : proctype(key, key) returns (bool)
  pair = record[k : key, i : item]
  node = record[k : key, i : item, left, right : table[key, item]]
  rep = variant[empty : null, some : node]
  xrep = array[pair]
  % The internal representation is a sorted binary tree.
  % All pairs in the table to the left (right) of a node
  % have keys less than (greater than) the key in that node.
  encode = proc(t : table[key, item] returns (xrep)
    xr : xrep := xrep$new( )  % Create an empty array.
    % Use allpairs to extract the pairs from the tree.
    for p : pair in allpairs(t) do
      xrep$addh(xr, p)  % Add the pair to the high end of the array.
      end
    return(xr)
    end encode
  decode = proc(xr : xrep) returns (table[key, item])
    t : table[key, item] := create( )  % Create empty table.
    for p : pair in xrep$elements(xr) do
      % xrep$elements yields all elements of array xr.
      enter(t, p.key, p.item)  % Enter pair in table.
      end
    return(t)
    end decode
  % Implementations of table operations appear here.
end table
```

Fig. 4. Partial implementation of table.

The implementor of the key–item table type does not need to *encode* or *decode* the values of keys or items. In fact, our scheme permits the key–item table to be implemented without advance knowledge of the key or item type; we only require that key and item values are themselves transmissible and that key values are ordered, as discussed above.

It is a little more difficult to define whether transmission of key–item tables preserves value equality, because here we are not defining a single type, but a collection of types; each type in this collection corresponds to a particular choice of key type and item type. Tables may be useful even if the key type or item type does not preserve value equality; it is up to the definer of table to decide what requirements to place on the parameter types. Whatever the requirements are, they should be stated in the specification.

For example, for tables we might require that keys, but not items, preserve value equality. (Thus complex numbers could be used as items but not as keys.) With this requirement, *transmit* for tables does not preserve value equality. Instead, the specification of *table$transmit* would be similar to that for arrays: *table$transmit* produces a new table containing the same key values as the original, and each key is associated with the transmitted verson of its original item.

2.4 Reasoning about Transmission

Recall that

$$T\$transmit(x) = decode(XT\$transmit(encode(x)))$$

for $x \in T$. It is not sufficient to prove this identity for a particular cluster, because that would only show that the translations done by *encode* and *decode* in that cluster are compatible. Instead, we need a more general method that works when *encode* and *decode* come from different implementations of the type.

In defining the external representation we have in mind a relationship between the values of XT objects and the values of the abstract type T; that is, the values of the XT objects are intended to represent the abstract values. This relationship can be expressed by means of a function that maps XT values into the abstract values they are intended to represent [8]. First, there may be an external representation *invariant* that defines the subset of XT values that are *legal* external representations. Then we define an *abstraction function* that maps the legal XT values to abstract T values:[3]

$$A_{XT}: XT \rightarrow T.$$

For example, for tables we have

$$\mathbf{xrep} = array[pair]$$

$$pair = record[k: key,\ i: item]$$

The external representation invariant is (informally) that all elements of the array contain different keys, and we can write the abstraction function as

$$A_{XT}(xr) = \{xr[i] \mid i \text{ between low and high bound of } xr\}$$

Here we have chosen to model values of tables as sets of key-item pairs.

We can use the external representation and its associated invariant and abstraction function to define the specification of *encode* and *decode* and thus determine the correctness of *encode* and *decode* implementations. Associated with the internal representation of a cluster are a representation invariant and abstraction function,

$$A_{RT}: RT \rightarrow T,$$

where RT is the internal representation type of T. At this point we have three types (*abstract*, RT, XT), two invariants, two abstraction functions (A_{XT} and A_{RT}), and *encode* and *decode*. These are related to each other by establishing two identities, one for *encode* and one for *decode*. These two identities are what must be shown to establish the correctness of *encode* and *decode*. For types that

[3] The method to be described can easily be generalized to use two abstraction functions, one on the sending side and one on the receiving side. Two functions are useful when *XT$transmit* does something strange, and *encode* and *decode* compensate for this strangeness. It is not clear that such a generalization is of practical interest.

preserve value equality, these identities are likely to be

$$A_{RT}(r) = A_{XT}(encode(r)), \tag{1}$$

$$A_{XT}(xr) = A_{RT}(decode(xr)), \tag{2}$$

where *xr* and *r* are respectively of type *XT* and *RT*, satisfying the external and internal representation invariants. (Note that in these identities the types of *encode* and *decode* differ slightly from those given above. In identity (1), *r* is the concrete **rep** object that represents the abstract object passed to *encode*; in identity (2), *decode* returns the concrete **rep** of the abstract object it constructs.) Intuitively, identity (1) says that the values of both *r* and *encode*(*r*) represent the same abstract value. Identity (2) says that the values of both *xr* and *decode*(*xr*) represent the same abstract value. In addition, *encode* and *decode* must preserve invariants; that is, the result of *encode* must satisfy the external representation invariant, and the result of *decode* must satisfy the internal representation invariant.

The identities above are inadequate for transmitting types, like complex numbers, that do not preserve value equality. For such types we require "closeness" instead of exact equality. Whatever the requirements on *encode* and *decode*, they must be stated as part of defining the external representation. For complex numbers, then, we might retain identity (1), thus requiring that *encode* not introduce any round-off error, but replace identity (2) by

$$|A_{XT}(xr) - A_{RT}(decode(xr))| < \varepsilon. \tag{2*}$$

In addition, for complex numbers another inexactness results from transmission of reals,

$$|A_{XT}(xr) - A_{XT}(XT\$transmit(xr))| < \delta.$$

It must be proved as part of defining the external representation that the error $\varepsilon + \delta$ is small enough to satisfy the specification of *complex$transmit*.

The identities and *transmit* are related by the following theorem.

THEOREM. *Given the specification of XT$transmit, and assuming that encode satisfies identity* (1) *and that decode satisfies identity* (2), *then encode*(*XT$transmit*(*decode*(*x*))) *satisfies the specification of T$transmit for all* $x \in T$.

This theorem is proved just once for a type; this proof should be given at the time the external representation is chosen and the identities are stated. The later proofs of the individual implementations of *encode* and *decode* justify the assumptions of the theorem.

2.5 Conclusion

A major advantage of our transmission method is modularity: transmission can be implemented, and then proved or understood, one module at a time. First note that the choice of external representation must be made independently of any implementations. Then, given the information about the external representation, including the invariant and abstraction function, and identities (1) and (2), the correctness of *encode* and *decode* can be shown locally for each cluster.

Note that transmissibility constrains the possible implementations of a type as follows. All values produced by a particular implementation must be transmissible, and all transmissible values must be represented. That is, for each *RT* value there must exist a corresponding *XT* value, and vice versa. We believe this constraint should apply to the built-in types as well as the abstract ones. Of course, the built-in types may have different implementations at different nodes. For example, suppose the external representation for integers were 32 bits. Then on a 16-bit machine, two words could be used to represent an integer, while on a 36-bit machine, only 32 of the 36 bits could be used to represent integers. It is worth noting that the constraints imposed by transmissibility are similar to those imposed by transportability.

3. SHARING

In the previous sections objects of a given type were viewed merely as containers for values of that type, and the identities of the objects themselves were ignored. This model is insufficient for many interesting data types, such as graphs, in which *sharing* of subcomponents is of interest. In this section we extend our scheme to provide the user with a simple means to control the effect of transmission on the internal sharing structure of values. The scheme is general enough to support transmission of cyclic structures, but we defer discussion of cycles to Section 5.

To model sharing, we must extend the definition of type to include the notion of an *object name*; each object of a type has a name as well as a value. Intuitively, if x and y have the same name, then they are exactly the same object. Changing the value of one will change the value of the other. We use the notation

$$x == y$$

to indicate name equality of objects x and y. (The distinction between name equality and value equality is the same as the distinction between EQ and EQUAL in LISP [16].)

If the value of x contains the name of y, then y is said to be a *component* of x. When an object is named more than once, we say the object is *shared*. Sharing comes in two varieties: intraobject sharing and interobject sharing. Intraobject sharing occurs when components are shared within a single object, as in Figure 5a. Interobject sharing occurs when components are shared between distinct objects. An example is given in Figure 5b. Here a record has two components: a table containing cells as items, and an individual cell that happens to be one of the cells in the table.

Shared structures cause the following question to be raised about transmission: Should transmission preserve the internal sharing structure of the transmitted object? For example, suppose that a table contains integer cells as items, and that two (or more) keys share the same cell (see Figure 6a). In the transmitted table, should the two keys share the same cell (as in Figure 6a), or should they be associated with different cells (as in Figure 6b)? Note that different behavior will result in the two cases: if the keys share a cell, then a change to that cell looked up using one key will be observed if the cell is looked up using the other key, while if the keys do not share the cell, the change will not be observed.

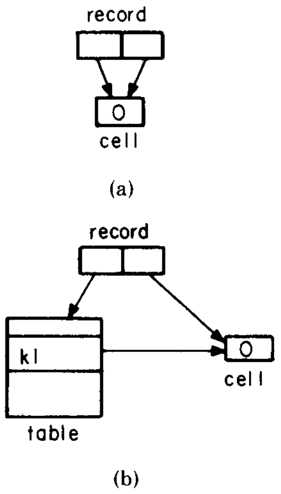

Fig. 5. Illustrations of sharing: (a) intraobject sharing; (b) interobject sharing.

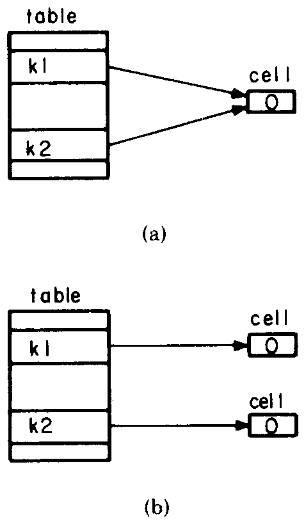

Fig. 6. Effects of transmission on sharing structure: (a) two keys sharing a cell containing the value 0; (b) two keys referring to two different cells, each containing the value 0.

Probably for the key-item table, and for types in general, sharing should be preserved. In general, preservation of intraobject sharing is a necessary part of preserving value equality, while preservation of interobject sharing may be needed to preserve value equality of a containing object. Nevertheless, only the definer of the type can decide about sharing preservation, since the decision affects the behavior of the transmitted data. Whatever the decision, it should be stated in the specification of *transmit.*

What support should we give for communicating sharing structure? While it is not necessary to provide any support, we believe that users will usually want sharing to be preserved, and that a language definition should make the most common and useful functionality easy to implement. Furthermore, explicit preservation of sharing by the language user in the absence of such support is difficult. For example, by choosing a more complicated external representation type, users implementing key-item tables could explicitly encode the intraobject sharing structure within individual tables; nevertheless, a user wishing to preserve interobject sharing as in Figure 5b would find it difficult to do so.

Therefore, we extend our scheme to preserve sharing structure automatically. First, we redefine the *transmit* operation discussed earlier to take a second argument: an *environment* that identifies the objects whose sharing matters. The environment allows us to delimit the context within which sharing should be preserved. For example, it appears useful to have the *transmit* operations preserve sharing when transmitting a single argument, or perhaps an entire message, but it does not appear useful to preserve sharing between objects sent in distinct messages. Secondly, we define *transmit* for all built-in types to preserve both intra- and interobject sharing.

As an example, we state the sharing properties for CLU arrays. Languages provide different methods for allowing objects to name other objects. In many languages this is done by using explicitly introduced pointer objects. In CLU, however, the concept of an object name is fundamental: all CLU data objects have names as well as values, and objects may refer to other objects by containing their names. For example, CLU arrays contain the names of their elements; thus the CLU type *array*[T] would correspond to "array of pointer to T" in other languages. Similarly, tables contain the names of key and item objects.

In the following, a and a' are arrays, i, i', j, j' are integers, t and t' are of type T (the element type of the array), and E is an environment. We have

(1) *Intraobject sharing.* Let $a' == transmit(a, E)$, $i' == transmit(i, E)$, and $j' == transmit(j, E)$. Then

$$a[i] == a[j] \Rightarrow a'[i'] == a'[j'].$$

(2) *Interobject sharing.* Let $a' == transmit(a, E)$, $i' == transmit(i, E)$, and $t' == transmit(t, E)$. Then

$$t == a[i] \Rightarrow t' == a'[i'].$$

As mentioned above, the user must define what transmission means for abstract types, but probably both inter- and intraobject sharing should be preserved. For example, in transmitting tables we want to preserve intratable sharing for items (any particular key appears in a table at most once, so intratable sharing for keys is not an issue) and interobject sharing for both keys and items. The specification of the sharing properties of transmission for tables is similar to that shown for arrays.

When implementing transmission for abstract types in which sharing preservation is desired, a good strategy is to choose an external representation type for which *transmit* preserves sharing as well as value equality. Such a choice will simplify the job of writing *encode* and *decode*. To preserve sharing, *encode* and *decode* merely move the component objects into the external or internal representation, respectively (e.g., *encode* stores in the external representation the names of the component objects contained in the internal representation.) The *encode* and *decode* operations for tables shown in Figure 4 preserve sharing in this way.

4. IMPLEMENTATION

Although the meaning of value transmission is controlled by the user, much of the actual work is performed by the run-time system that underlies the language implementation. As was mentioned earlier, each transmissible type can be thought of as having a *transmit* operation, but *transmit* is actually implemented by the system, using the *encode* and *decode* operations provided by the user. In implementing *transmit* the system performs a number of tasks, including sharing detection, message formatting, and interaction with the communication medium.

In this section we describe an implementation of *transmit*. This implementation differs from our actual running implementation in some minor details; it is simpler to describe but less efficient.

Message transmission involves translating an arbitrary graph structure (the object) into linear form (the message), and vice versa. A number of algorithms have been developed for efficient linearization of graphs [1, 4]. Our algorithm differs from these because of the necessity of applying user-defined translation operations (*encode* and *decode*) during object traversal.

Modules exchange information through a *communication substrate*, which is the language implementation's interface to the communication medium. The communication substrate encapsulates such functions as routing and addressing of messages, the observation of transmission protocols, and the detection and correction of transmission errors. We do not describe the communication substrate in detail, because its structure will be highly dependent on specific characteristics of the communication medium and the nodes.

A *message stream* data type is provided by the communication substrate to implement message construction and interpretation. Information enters and leaves a message stream in discrete units called *tokens*. The message stream type has operations to insert and extract various kinds of tokens. When a module executes a **send** statement, the system constructs an output message stream. For each actual argument object in the message the system traverses the object, placing tokens in the stream. The communication substrate then delivers the message stream to the receiving module's node, where the tokens are removed from the stream, and the transmitted objects are reconstructed.

A *map* data type is used to detect sharing; it implements the environment discussed previously. A different map is used for each argument, since we choose to preserve sharing only within the individual arguments (rather than within the message as a whole). Each token in a message stream has an associated *stream address*. A map is a table that associates (names of) objects and stream addresses. When the system begins to transmit an argument object's value, an empty map is created to keep track of component identities. As the object is traversed, (the name of) the object itself, and each of its components, is recorded in the map, along with the stream address of the start of the corresponding sequence of tokens. Similarly, when a message is interpreted, each reconstructed component is entered in the map, along with its associated stream address; again a different map is used for each argument.

There are three kinds of tokens: data tokens, header tokens, and back reference tokens. *Data tokens* are used to transmit the values of unstructured language-defined types such as integers, booleans, or strings.

For both abstract types and language-defined structured types, the first token of an encoded abstract value or structured value is always either a header token or a back reference token. A *header token* marks the start of an encoded value of structured or abstract type. Header tokens may contain type-dependent information; for example, a variant header token contains the value of the tag. An array header token contains the low bound of the array, and the number of elements in the array; the low bound is needed to reconstruct the value of the array, and the size is needed both to reconstruct the array and to interpret the subsequent tokens in the stream (i.e., to determine how many of the subsequent tokens correspond to array elements).

Back reference tokens denote shared components. A back reference token contains the stream address of the encoded value of an object. At the sending module, when an object's value is reduced to tokens, the (name of the) object and the stream address of its header token are entered in the map. If the object is encountered again, this stream address is extracted from the map and placed in the stream in a back reference token. At the receiving module, each reconstructed component is entered in the map with its stream address. When a back reference is encountered, the previously reconstructed object is extracted from the map.

For each transmissible type *T*, the language implementation provides *put* and *get* operations, to translate between *T* values and sequences of tokens. These two operations are part of the language implementation; they are not implemented by users of the language. The *put* operation for arrays does the following:

1. If the array object has already been entered in the map, place a back reference token in the stream, and return.
2. Otherwise, enter the object and the current stream address in the map.
3. Place a header token containing size information in the stream, and invoke *put* for each component object.

Here is the *get* operation for arrays:

1. Remove the first token from the stream. If it is a back reference, return the indicated object from the map.
2. Otherwise, create an empty array, and enter the new object and the current stream address in the map.
3. Invoke the *get* operation for each component object, and initialize the array.

Put and *get* for the other structured types are similar. The *put* operation for an abstract type uses the following algorithm:

1. If the object has already been entered in the map, place a back reference token in the stream, and return.
2. Otherwise, enter the object and the current stream address in the map.
3. Place a header token in the stream, use *encode* to construct an external representation, and invoke the external representation's *put* operation.

Finally, *get* for an abstract type uses the following algorithm:

1. Remove the first token from the stream. If it is a back reference, return the indicated object from the map.
2. Otherwise, construct the external representation by invoking its *get* operation, and then use *decode* to create the abstract object.
3. Enter the new abstract object and its stream address in the map, and return the object.

These algorithms suffice to preserve acyclic sharing. (Cyclic sharing is slightly more complicated and will be addressed in the next section.)

As an example of the message format, in Figures 7 and 8 we show a table object and its associated message representation. In this example the table has integer keys and cells as items, and the external representation of cells is *record*[*val*: *int*].

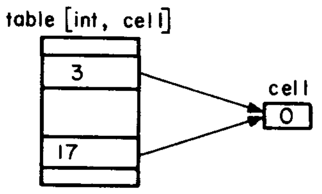

Fig. 7. Example table object.

Token #	Token Type	Token Value	Explanation
1	Header		Header for table
2	Header	1,2	Header for array (Table external representation with low bound 1 and size 2)
3	Header		Header of first array element (a record)
4	Data	3	The first key
5	Header		Header for the first item
6	Header		Header for item's external representation (a record)
7	Data	0	The item's value
8	Header		Header for second array element
9	Data	17	The second key
10	BackRef	5	The shared item

Fig. 8. Example of message representation for table.

The above implementation is based on the assumption that message transmission is type-checked at compile time. Run-time type checking would require type information to be sent in messages (see Section 6).

5. CYCLIC OBJECTS

The naming relation among objects can be modeled as a directed graph. If the graph corresponding to an object contains a directed cycle, we say the object has a cyclic value. Simple examples of objects having cyclic values are circular lists and doubly linked lists, where each element names both its predecessor and successor. The method described so far does not fully define value transmission for cyclic values. In this section we extend our scheme to encompass cyclic values, subject to a simple restriction on the *decode* operations of such types.

The *put* operation described in Section 4 handles both acyclic and cyclic objects. The first thing *put* does when it encounters a new object to be transmitted is to place a header token for the object in the message stream and enter the (name of the) object in the map along with the address of the header token. If a reference to the object is encountered later, a back reference token is placed in the message stream. In the case of a cyclic object, the object referred to has not yet been fully reduced to tokens, but this causes no difficulty.

A problem does arise when trying to reconstruct a cyclic object. *Get* does not enter an object in the map until it has been fully reconstructed. When reconstructing a cyclic object, *get* will encounter back reference tokens that refer to objects that are not fully reconstructed. Since these objects are not present in the map, the implementation in Section 4 treats the situation as an error.

Just as it is not necessary for the system to preserve sharing, the system need not help the programmer in transmitting cyclic values. In this case the programmer must encode cyclic values into acyclic external representations. We believe that programmers would find this awkward in practice.

Therefore, our system provides a certain amount of help in transmitting cyclic objects. As mentioned above, the occurrence of a cycle during reconstruction is indicated when the back reference token is extracted from the message stream but the object referred to has not yet been entered in the map. At this point, *get* creates an *uninitialized object*, that is, an object that has a name, but no value, and enters it in the map with the stream address contained in the back reference token. Reconstruction can then continue with this uninitialized object used in place of the as yet nonexistent object that is needed to close the cycle. The uninitialized object, however, can be used only in a very restricted way: its name can be used, but not its value (since it has none). An operation *uses the value* of an object of type *T* if it causes a *T* operation to be applied to that object. The only way a *T* operation can be applied to an uninitialized object is through the user-defined *decode* operation. In general, when an object containing a cycle is reconstructed, its *decode* operation may not use the value of any other object in the cycle. After *decode* operations have been applied to each object in the cycle, all objects are considered to be initialized, and arbitrary operations can then be applied.

The restriction on operations is by no means the only way to define transmission for cyclic values. Generally, if the operation decoding an object uses the value of another object, the latter must be decoded before the former. Our method reconstructs acyclic objects in exactly such a "leaf-to-root" order. Our restriction on cyclic objects can be expressed as the requirement that no such dependency relations exist among any members of a cycle. In [7] a less restrictive scheme is described, in which we permit dependency relations among objects in a cycle but require that the closure of those relations be acyclic. Lazy evaluation is used to determine the order in which objects are decoded. Although the scheme described here is more restrictive, it is simpler both in implementation and description. Furthermore, experience suggests that implementations that violate our more conservative restriction are unlikely to be useful in practice.

It is possible to implement transmission for an abstract type in such a way that message construction or interpretation may fail to terminate. Although we feel that nonterminating *transmit* operations are unlikely to be a problem in practice, for completeness we will review some issues pertaining to termination. We will assume that user-defined *encode* and *decode* operations always terminate. For an abstract type *T* having external representation type *XT*, *T*$*transmit* is defined in terms of *XT*$*transmit*. The simplest way to guarantee that *T*$*transmit* always terminates is to choose an *XT* whose *transmit* operation is already known to terminate. The resulting *T*$*transmit* will not be recursively defined. The complex number example given above reflects such a choice. For recursively defined types it may be convenient to choose an *XT* that contains *T* components. In such a case, *T*$*transmit* and *XT*$*transmit* are mutually recursive, and demonstrating termination is more difficult. Recursive invocations of *T*$*transmit* can stop in two ways: the collection of reachable *T* objects can be exhausted, for example, by reaching the end of a linear list, or an object may be encountered for the second time, for example, by closing the cycle of a circular list.

6. LONG-TERM STORAGE

The method described above is useful for storing data in long-term storage, such as a file system. One module can store an item of type *T*, while another, using a different internal representation for *T*, can fetch it later. While in the file system, the data is stored in its message format as uninterpreted bits. The advantage over standard schemes is that these bits have a system-known meaning: they are the message format of the external representation of *T*. This information can be used to prevent user errors in interpreting the bits, for example, interpreting bits belonging to one type as if they were of another type, or lack of agreement among modules as to what the external representation is for a given type.

We must extend our method to permit programs to extract uninterpreted data from messages, but this is easy to do. We introduce a new built-in type called *image*, and allow values of this type to be communicated in messages (*images* are transmissible). Values of type *image* are pairs: a bit string containing a value of type *T* in message format, and a bit string representing the type *T* itself. *Images* have only two operations. The first operation,

create = **proc**[*T*: *type*](*x*: *T*) **returns** (*image*)

is used to construct an *image*. It is parameterized by the type of the argument; this type must be transmissible (transmissibility can be checked at compile time). *Create* produces the message format both for its actual argument, *x* (using the *put* operation for *T*), and for *T* itself (using the *put* operation for type *type*, that is, types are transmissible). The second operation,

extract = **proc**[*T*: *type*](*x*: *image*) **returns** (*T*) **signals** (*wrong_type*)

is parameterized by the expected type, which must be transmissible. *Extract* decodes the type part of its input (using the *get* operation for types), compares the result with *T*, and signals *wrong type*[4] if the types do not match. Otherwise, it returns the object of type *T* that it reconstructs from the message format (using the *get* operation for *T*).

We expect that *image* objects will be created most frequently when sending a request to store data to a module that provides uninterpreted storage. Later the stored data would be retrieved by sending a read request to that same module; the response to the read request would provide the *image* stored earlier. The stored value could then be extracted from the *image*, provided the reading module knows the type of the stored data. Within the module providing storage, the actual type of the data is not of concern, since that module deals only with uninterpreted *images*. A practical consequence of this lack of concern is that the code of that module can be written, compiled, and loaded without knowledge of the types of data it will store.

Since the module that retrieves data may not be the module that stored it, the

[4] Here we are making use of exceptions. *Extract* can terminate in two ways: either normally (returning a *T* value) or exceptionally by signaling *wrong type*. See [13] for more information.

type information in *images* must be meaningful at all modules. Support for this property requires a universal name space for types. If the file system and its users are distributed, the name space must be distributed. The issues involved in implementing such a name space are similar to those that arise in distributed name servers in general and will not be discussed here.

7. DISCUSSION

This paper is based on the premise that in a large program it is desirable to permit different regions to use different internal representations for abstract data. We further argued that communication of abstract data between such regions must be user controlled. We then presented a method that gives users the needed control, but provides system support to simplify the user's job.

In the introduction we identified some goals that such a method must satisfy, and our method does satisfy these goals. Our method is modular: each abstract data type implementation can be developed and verified independently of any other implementations, including other implementations of its data type and implementations of used abstractions (e.g., the external representation). Our method satisfies the linearity property, since for each new implementation only the *encode* and *decode* operations for that implementation need be written. Finally, our method is easy to use. Ease of use comes partly because the external representation is just another (possibly abstract) data type, so the user need not worry about translation into low level types (e.g., bit strings). Ease of use is enhanced by the system support for transmission of shared and cyclic data structures.

In the remainder of this section we discuss some issues related to our work. First, we discuss the relationship of our work to other research. Next, we discuss some ways of improving the efficiency of transmission. Finally, we conclude with a discussion of some areas for further research.

7.1 Related Work

Our scheme for value transmission is the first to provide a complete treatment of the many related issues including user-defined transmission of abstract values, specification and correctness criteria for transmission, and system support for shared and cyclic data objects. Our scheme is based on the straightforward idea of a standard intermediate representation for each type. A canonical representation has been used for transmission of built-in types in various protocols [3, 18]. It has been proposed for abstract types by Habermann [6], but the details were not worked out.

Several protocols have been developed for transmission of typed information across the ARPANET. In each of these, only values from a predefined set of types are transmissible. The Procedure Call Protocol developed for the National Software Works [18] is the most ambitious, being capable of transmitting such values as character strings, integers, and lists. The TELNET protocol is used for transferring character information, and the File Transfer Protocol is used to transfer files [3]. In these protocols the sender converts the information to be sent into a standard representation that is either statically determined or agreed upon by negotiation. Upon receipt the receiver converts the standard representation into whatever local representation it uses. Our scheme builds on the results of these efforts, since we assume that the underlying language implementation can faithfully transmit such language-defined values as strings, or arrays of integers, independently of their machine-level representations.

An alternative to standard intermediate representations is direct translation between representations. Fabry [2] develops a scheme for replacing implementations of abstract data types while the ambient system continues to run. During the transition period between implementations it is possible that different representations for objects of the same type may coexist. In Fabry's scheme each object is tagged with an implementation number, and each data type implementation includes a translation operation from the representation used by the previous version to its own representation. Whenever an object using an old representation is encountered, a chain of translation operations is invoked to convert the object into the current representation for that type.

Like our scheme, Fabry's scheme has the property that the amount of work needed to install a new implementation is independent of the number of previously existing implementations. Nevertheless, it has a major defect: the lack of modularity inherent in the idea that each implementation must be known by the programmer of its successor. Note that there may be no natural order among implementations, making it difficult to assign version numbers. For example, two implementations with different efficiency characteristics may be developed in parallel.

7.2 Efficiency

Message transmission has three distinct steps: first, constructing a message stream in memory at the sender's node; second, transmitting the message to obtain a message stream in memory at the receiver's node; and third, reconstructing the objects from the receiver's message stream.

A simple measure of efficiency is the number of passes required to perform these steps. To perform step 1, our implementation requires a single recursive traversal of the object by the sender: a similar single traversal is required for step 3. At each level a user-defined translation operation is applied, a certain amount of bookkeeping is done to detect sharing, and some tokens are entered in or removed from the message stream.

This section describes a few simple optimizations that serve to improve the performance of the implementation. The result of these optimizations is that the efficiency of transmission is related to the generality required of transmission. We begin by discussing optimizations for steps 1 and 3, and then we discuss some for step 2.

Sharing preservation represents a cost in steps 1 and 3, since objects must be entered into and retrieved from maps. In the current CLU implementation, object identities are compared by a simple pointer comparison, and standard hashing and retrieval techniques are used to make the map types efficient. The cost of sharing preservation can be further reduced by observing that it is not necessary to preserve sharing of *immutable* objects. The value of an immutable object can never change. If two immutable objects have the same value, then they are indistinguishable from one another, and it is not possible to observe whether they are actually the same object. CLU provides a number of immutable types, both unstructured (e.g., *int*, *bool*) and structured (e.g., *sequence*, *struct*). The *put* and *get* operations of these types do not need to preserve sharing and therefore need not enter or retrieve information in the map. The decision whether to preserve sharing of immutable objects is based on a time–space trade-off. By not preserving sharing, *put* and *get* execute more efficiently. By preserving sharing, we may save storage and reduce message size. For small immutable objects, for example, *int*, *bool*, both space and time can be saved by not preserving sharing.

A simple compile-time optimization can further reduce the cost of sharing preservation, as well as the number of procedure invocations. We anticipate that many data abstractions will be implemented having identical internal and external representations, and having *encode* and *decode* operations that only perform type conversions between the abstract and representation types. We call such operations *trivial* translation operations. Trivial operations can easily be detected at compile time. We can eliminate the expense of sharing preservation for types having trivial *encode* and *decode* operations. The *put* and *get* operations of such a type need not check for sharing, as any sharing will eventually be detected at a lower level.

The number of procedure calls in steps 1 and 3 can be reduced by observing that it is not necessary to call trivial translation operations at run time. In the CLU run-time system, objects contain no abstract type information, since type correctness is enforced statically by the compiler.[5] Since trivial translation operations just return their arguments unchanged, their invocations have no effect, and *put* and *get* need not call them. (Calls of nontrivial operations can be eliminated by in-line substitution, i.e., replacing the call with the body of the called routine.)

The following optimization for trivial operations depends on the use of a different message stream format from that described in Section 4. In our running implementation the message format does not use header tokens for abstract objects; the algorithms in use are slight variations of those described in Section 4. With the revised format, *put* and *get* for abstract types only do sharing detection and calling of *encode* or *decode*. Since for trivial operations neither of these need be done, *put* and *get* can be replaced by trivial operations that just invoke the *put* and *get* of the external representation type. As a further optimization, when separately compiled modules are bound together, or if the modules are compiled together, invocations of trivial *put* and *get* operations can be replaced by direct invocations of the external representation's *put* and *get*, eliminating levels of procedure linkage.

In summary, the only types whose *put* and *get* operations need to incur the cost of sharing preservation are the mutable built-in types and abstract types having nontrivial *encode* and *decode* operations. Furthermore, *put* and *get* need not be called at all for abstract types with trivial *encode* and *decode*.

It is important to realize that the above optimizations are performed separately for sender and receiver. For example, suppose the sender used only trivial

[5] The conversion between abstract and representation types indicated by the use of **cvt** does not cause any code to be executed at run time; it simply changes how the compiler does type checking.

operations so that only *put* operations for built-in types actually need to be invoked during transmission. These optimizations would be discovered either at compile time of the code for the sender (if all this code were compiled at once) or at load time for the sender. Similarly, the receiver's optimizations are done at compile or load time for the receiver's code. But no matter how many optimizations are done at either end, the message format produced at run time by the sender will be readable by the receiver.

Our implementation models messages as streams to permit flexible storage management by the communication substrate. Since messages are constructed in a single pass, the amount of storage used in step 2 can be controlled by *incremental transmission*. When sending the value of a large object, the system may begin transmitting the message before it is fully constructed and may begin reconstructing the value before the message is fully received.

Further optimizations are possible when the sender and receiver reside at the same node (i.e., share memory). First of all, step 2 can be eliminated in this case; the message stream produced by the sender can be built directly in the receiver's address space. In addition, if the sender and receiver use the same implementation for the built-in types, the message stream can be eliminated. Rather than placing a token in a stream to denote an integer value, the sender can just copy the integer into the receiver's address space. Values of built-in type can be communicated by direct copying of underlying representations, rather than by translation into tokens. This technique conserves storage at the sender's end, since the message is constructed in the receiver's address space, and it saves work at the receiver's end, since tokens do not have to be interpreted.

One can imagine doing more optimizations when the sending and receiving modules use the same implementations of the abstract types being communicated in a **send**. We expect that such optimizations will not be practical in our system [12]. We permit modules to be created dynamically, so we can detect a module's nodes of residence only at run time. Furthermore, we do not require that all modules at a node have the same implementations of abstract types. Therefore we expect that the effort saved by this final optimization will not justify the effort needed to determine that the optimization can be applied.

7.3 Further Research

In this section we discuss some areas in which further research is needed.

Our method limits each module to a single implementation for each type of abstract data it uses. Suppose that we eliminated this restriction, so that a single module might use objects of the same type but with different representations. Note that such generality requires a way of determining the version of an object at run time, so that it can be operated on by operations that understand its representation or translated to the representation needed by the operation. Although our method could be used to do translations, such an approach is probably too inefficient. Alternative methods, not related to the work described in this paper, must be explored if this problem is felt to be important.

Another limitation of our method is that we make no provision for a change in the external representation. What happens if a change in the external representation is needed? One possibility is to use Fabry's method. This requires that each *XT* object be tagged with a version number; whether this is acceptable requires further study. Another possibility is to perform a system change that replaces all existing implementations of the type in question with new ones using the new external representation and converts all existing *XT* objects to the new format. Such a change must be done *atomically*: it must either be completed entirely or have no effect. Probably it is practical to use this approach only in an environment that supports atomic actions. Our current research concerns defining and implementing such an environment [10, 12].

Our work has assumed a single language system. An important practical problem concerns communication between programs written in different languages. At present such communication is usually accomplished through files of uninterpreted bit strings. Perhaps our method could be extended to handle interlanguage communication. Such an extension would be an improvement over current techniques, partly because it would offer a standard method for doing such communication, and partly because communication could be carried out at a higher level than bit strings.

Finally, the way in which the work of transmission is divided between the system and the user is of interest in its own right. We view the system-provided *put* and *get* operations as templates: the user-defined *encode* and *decode* operations fit into a system-defined pattern. This kind of pattern is useful for more than transmission. Fairly obvious examples of the utility of template-driven operation calls are support for user-defined copying of abstract objects, or for testing two abstract objects for equality. Another possible use is for control of human-readable display of abstract data. The use of templates as a general control mechanism is worth further study.

ACKNOWLEDGMENTS

Special thanks are owing to Russ Atkinson, who helped in the original formulation of the transmission method, and Craig Schaffert, who helped formulate the verification criteria. Sheng Yang Chiu, Julie Lancaster, Brian Oki, Gene Stark, Bob Scheifler, Bill Weihl, and the referees gave helpful criticisms of earlier drafts of this paper.

REFERENCES

1. CLARK, D.W. Copying list structures without auxiliary storage. Tech. Rep., Carnegie-Mellon Univ., Pittsburgh, Pa., Oct. 1975.
2. FABRY, R.S. How to design a system in which modules can be changed on the fly. Proc. 2nd Int. Conf. on Software Engineering, San Francisco, Calif., Oct. 1976, pp. 470–477.
3. FEINLER, E., AND POSTEL, J. *Arpanet Protocol Handbook.* Network Information Center, SRI International, Menlo Park, Calif., Jan. 1978.
4. FISHER, D.A. Copying cyclic structures in linear time using bounded workspace. *Commun. ACM 18,* 5 (May 1975), 251–253.
5. GUTTAG, J.V., AND HORNING, J.J. The algebraic specification of abstract data types. *Acta Inf. 10* (1978), 27–52.
6. HABERMANN, N. Dynamically modifiable distributed systems. Proc. Distributed Sensor Net Workshop, Pittsburgh, Pa., Dec. 1978, pp. 111–114.
7. HERLIHY, M. Transmitting abstract values in messages. Tech. Rep. MIT/LCS/TR-234, Laboratory for Computer Science, M.I.T., Cambridge, Mass., May 1980.
8. HOARE, C.A.R. Proof of correctness of data representation. *Acta Inf. 1* (1972), 271–281.

9. KAPUR, D. Towards a theory for abstract data types. Tech. Rep. MIT/LCS/TR-237, Laboratory for Computer Science, M.I.T., Cambridge, Mass., May 1980.
10. LISKOV, B.H. On linguistic support for distributed programs. Proc. Symp. on Reliability in Distributed Software and Data Base Systems, Pittsburgh, Pa., July 1981, pp. 53-60.
11. LISKOV, B.H., ATKINSON, R.A., BLOOM, T., MOSS, J.E., SCHAFFERT, J.C., SCHEIFLER, R.W., AND SNYDER, A. *CLU Reference Manual*, Lecture Notes in Computer Science 114. Springer-Verlag, New York, 1981.
12. LISKOV, B.H., AND SCHEIFLER, R.W. Guardians and actions: Linguistic support for robust, distributed programs. Proc. 9th Ann. ACM Symp. on Principles of Programming Languages, Albuquerque, N.M., Jan. 1982, pp. 7-19.
13. LISKOV, B.H., AND SNYDER, A. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE-5*, 6 (Nov. 1979), 546-558.
14. LISKOV, B.H., SNYDER, A., ATKINSON, R.R., AND SCHAFFERT, J.C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564-576.
15. LISKOV, B.H., AND ZILLES, S.N. Programming with abstract data types. Proc. ACM-SIGPLAN Conf. on Very High Level Languages, *SIGPLAN Notices* (ACM) *9*, 4 (Apr. 1974), 50-59.
16. MOON, D.A. *MacLISP Reference Manual, Revision O.* Project MAC, M.I.T., Cambridge, Mass., Apr. 1974.
17. PRELIMINARY ADA REFERENCE MANUAL. *SIGPLAN Notices* (ACM) 14, 6 (June 1979).
18. SCHANTZ, R.E., AND MILLSTEIN, R.E. The foreman: Providing the program execution environment for the national software works. Tech. Rep. No. 3442, BBN, Jan. 1977.
19. SOLLINS, K. Copying complex structures in a distributed system. Tech. Rep. MIT/LCS/TR-219, Laboratory for Computer Science, M.I.T., Cambridge, Mass., May 1979.

Received December 1980; revised August 1981 and April 1982; accepted April 1982