Communicating Abstract Data Type Values in Heterogeneous Distributed Programs

Lin Huang and David Alex Lamb
Department of Computing and Information Science
Queen's University
Kingston, Ontario, CANADA K7L 3N6

Abstract

This paper is concerned with the problem of communicating abstract data type (ADT) values in heterogeneous distributed programs. It focuses on addressing two fundamental issues of the problem: the selection of suitable exchange representations and the generation of data converters. Exchange representations are the ways to represent data during transmissions across networks. Data converters are programs transforming data from one representation to another; they are the major facility to deal with heterogeneity in a communication. This paper reports the following results: a term-based exchange representation, which is an abstract notation and so is particularly suitable for communicating ADT values in heterogeneous programs; and methods to generate data converters.

1 Introduction

In this paper we are concerned with the problem of communicating abstract data type values in heterogeneous distributed programs.

An **abstract data type** is a type characterized entirely by a set of functions. It contains a collection of *domains*, a collection of *functions* on those domains, and a collection of *axioms* that the functions satisfy. Abstract data types capture the notion of data abstraction, a software engineering principle that calls for specifying the behaviour of a type without spelling out its implementation.

A distributed program is **heterogeneous** if one or more of the following occur: 1) it resides on a distributed system that involves different types of computers, different operating systems, or different communications networks; 2) it uses different programming languages; and 3) it uses different implementations for the same abstract data type.

In real world computing environments, there is no ideal set of resources — hardware or software — for all applications. A particular resource may be suitable to certain situations but not to others. Heterogeneous distributed programs are likely to be common. As distributed programs become increasingly large and complex, it is advantageous to implement them by the principles of data abstraction — an effective weapon to manage complexity. Since the major means to achieve data abstraction is to use abstract data types (ADT), support of communicating ADT values is important.

Most of the previous work on communication in heterogeneous distributed programs [1, 3, 4, 5, 8, 15, 17, 23, 24, 29] deals with transmitting simple built-in type values. Although there have been a few on communicating ADT values [2, 10, 19], they do not solve the problem well. For example, they require the user to provide data converters.

This paper presents a novel approach to the problem: using a term-based exchange representation and generating data converters. After describing a general approach to the problem, we describe our solution, show how to generate abstractors, then discuss how to improve their performance.

2 A General Approach

This section describes a general approach to the problem and discusses issues of the approach.

Any work on communication in heterogeneous distributed programs must resolve representation heterogeneity, the obstacle to direct transmission of data from one module to another. One general approach is to choose an exchange representation acceptable to all the involved modules and to equip each module with an in-converter and an out-converter, where the out-converter of a module is responsible for converting data from its local representation to the chosen

1063-6927/94 $03.00 © 1994 IEEE

exchange representation, and the in-converter is responsible for the inverse conversion.

The advantages of this approach are as follows. 1) Details of the local representation in a module are hidden from other modules. The sender is only concerned with how to convert the communicated value from a local representation to the exchange representation and the receiver is only concerned with the inverse transformation; Neither needs to know the local representation in the other. 2) Representation changes are localized. A module can change its local representation without worrying about affecting other modules' in- or out-converters. 3) A module only needs two converters. New modules can be added into a program without causing any new converters to be added to the existing modules.

The general approach implies two central issues — implementation of the necessary data conversions and selection of an appropriate exchange representation.

An exchange representation is the way to represent data during transmission and serves as an intermediate representation between the local representation used by the sender and the one used by the receiver. To be suitable for communication in heterogeneous distributed programs, where different representations are involved, an exchange representation should be *abstract* — independent of any local representation — that is, free of the influence of any particular machine, operating system, programming language or user.

The general approach requires in- and out-converters at the sender and receiver. Automatic generation of these converters is important because: 1) It can provide a high degree of representation heterogeneity transparency, thus hiding the data representation differences from the programmer. 2) It can ensure the correctness of converters if the generator is correct, while manual methods are error-prone because of the possible complexity and diversity of local representations. 3) It can improve programming productivity, while manual methods are usually time consuming.

3 A Solution

Figure 1 shows a specification of the well-known *Stack*[1]. The specification should be understandable to readers familiar with algebraic specifications.

A **construction term** of a value is an expression that contains only constructors and whose evaluation yields the value. As an example, for a stack of three elements, 10, 11 and 12, with 10 at the bottom and 12 at the top its construction term is $push(push(push(new, 10), 11), 12)$.

Specification	Stack
Parameters	Ele: Type
Declaration	Stack[Ele]
Base_types	Boolean, Ele, Integer
Functions	new: $\rightarrow$ Stack
	isnew: Stack $\rightarrow$ Boolean
	push: Stack $\times$ Ele $\rightarrow$ Stack
	pop: Stack $\rightarrow$ Stack
	top: Stack $\rightarrow$ Ele
	size: Stack $\rightarrow$ Integer
Constructors	new, push
Equations	top(push(s, e)) = e
	pop(push(s, e)) = s
	isnew(new) = true
	isnew(push(s, e)) = false
	size(new) = 0
	size(push(s, e)) = size(s)+1

Figure 1: Specification of *Stack*

The **abstractor** of an ADT is a function that takes a value as an input and produces the corresponding construction term as an output.

The **interpreter** of an ADT is a function that takes a construction term as an input and produces the corresponding value as an output.

Our solution is based on the fact that any value of an abstract data type can be built by its constructors. We choose construction terms as the exchange representation, and so we implement in-converters by interpreters and out-converters by abstractors. The communicated value is transformed into its construction term by the abstractor at the sender and the construction term is then transmitted to the receiver. The interpreter at the receiver parses the term and produces the local representation by invoking the constructors in the term in an appropriate order.

For example, suppose the sender implements type *Stack* as a linked structure and the receiver as an array (Figure 2). First, the stack (a linked structure) is transformed into its construction term $push(push(push(new, 10), 11), 12)$ by the abstractor at the sender, then the construction term is transmitted to the receiver, and the receiver's interpreter invokes the operations in the construction term to make its local representation (an array).

Three immediate advantages of this solution are as follows. 1) The exchange representation is a mathe-

[1] The equations in the specification are not complete; for example, there are no error handling equations. We only list the equations that are useful to illustrate our solution.

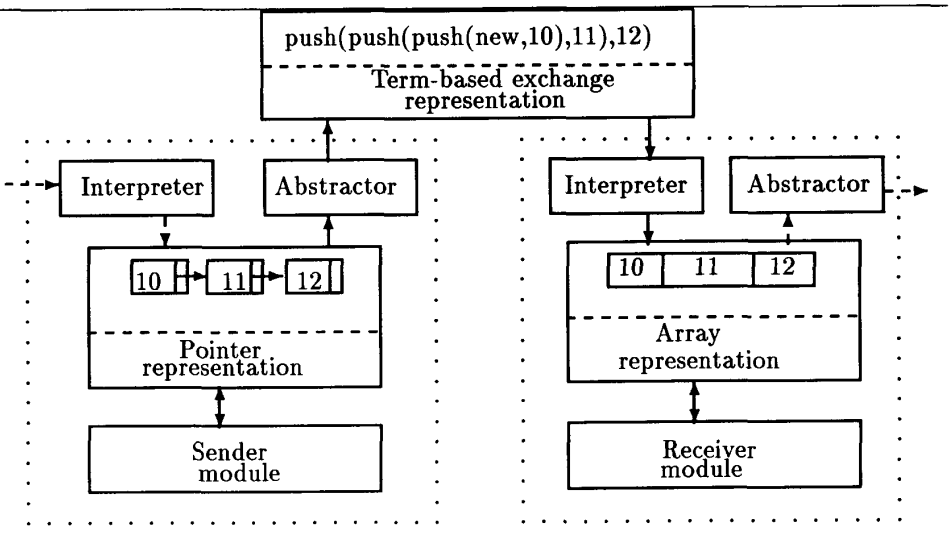

Figure 2: The process of communicating a stack

matical notation — independent of any machine, programming language or implementation — and so is particularly suitable for communication in heterogeneous distributed programs. Pointers do not appear in the exchange representation; communicating data containing them, long considered a difficult task, is no longer a problem. 2) Construction terms are a natural exchange medium for ADTs which have constructors, since any input ADT value could be generated by constructors. 3) In-converters (*i.e.*, interpreters) are simple parsers. In [13, 14], we showed that construction terms of a type can be generated by a context-free grammar, the interpreter is no more than a syntax-directed translator, and the grammar is LL(1). Hence, interpreters can be automatically generated.

4 Generating Abstractors

Given a type T, denote its abstractor by abs_T. By generating an abstractor abs_T we mean to produce an implementation of abs_T which is defined by a recursive equation without side effects, like any function definition. The recursive equation can be used as a rewriting rule of abs_T. In other words, the generated abs_T is executable. This section presents two methods for generating abstractors: generation from algebraic specifications, and generation from "implementations" (that is, from abstraction functions in abstract models).

4.1 Generation from Specifications

In generation from specifications, a generator analyzes the specification of a given type T and derives the abstractor, which is composed purely of calls on T's functions. The abstractor is independent of any implementation of T and so can be used by all its implementations. Thus, in this method, only one abstractor is needed for a given type.

In the following, we classify specifications into two classes: *symmetric* and *asymmetric*, and discuss abstractors for each class respectively.

4.1.1 Symmetric Specifications

Def. 1 Let T be a type with m constructors $cons_1, \ldots, cons_m$:

$$cons_i : T_{i,1} \times \ldots \times T_{i,n_i} \longrightarrow T$$

For a given constructor $cons_i$ $(1 \leq i \leq m)$, if the specification of T contains a function $g : T \longrightarrow$

$T_{i,j}$ $(1 \leq j \leq n_i)$ with an equation of the form of $g(cons_i(x_1, \ldots, x_j, \ldots, x_{n_i})) = x_j$, g is called the **j-th normal selector** of $cons_i$, denoted by $sel_{i,j}$. Constructor $cons_i$ is **invertible** if all its normal selectors $sel_{i,1}, \ldots, sel_{i,n_i}$ are defined.

In other words, if a constructor is invertible, for a value constructed by the constructor, its components can be recovered. For example, to constructor *push* in type *Stack*, functions *pop* and *top* satisfy the conditions in Definition 1 and they are its first and second normal selectors. Therefore, *push* is invertible.

Def. 2 Assume T has m constructors as in Definition 1. If the specification of T contains a function: $disc : T \longrightarrow D$, where D is a set with m elements $\{d_1, \ldots, d_m\}$, and if the specification contains m equations: $disc(cons_i(x_{i,1}, \ldots, x_{i,n_i})) = d_i$, then *disc* is called a **discriminator** of T and D is called the **discrimination set.**

By this definition, one may verify that *isnew* in *Stack* is a discriminator.

Def. 3 The specification of a type is **symmetric** if the type has a discriminator and all the non-constant constructors are invertible.

Intuitively, a symmetric specification means the symmetry of constructors with selectors.

We now present an algorithm to check if the specification of T is symmetric.

For each non-constant constructor *cons*
  For *i* from 1 to the number of the arguments of *cons*
    If *cons* does not have the *i*-th normal selector (Definition 1)
      Return "The specification is asymmetric."
  End_for
End_for
If T has a discriminator (Definition 2)
  Return "The specification is symmetric."
Else Return "The specification is asymmetric."

The normal selector and discriminator tests can be done based on Definitions 1 and 2; we have implemented them in our prototype system [14].

Many specifications should be symmetric. For example, the specifications of *Stack*, *Deque*, *list* in [6] are symmetric; the specification of ordinary *Binary_tree* is also symmetric. Furthermore, all "categorically constructed" data types are inherently symmetric (For every constructor of a categorical type, the inverse is also defined in the specification.) Workers in this area [7, 22] argue that data types should be categorically constructed because 1) functions are derived from the constructors, 2) equations are derived rather than built in an ad hoc way, 3) the equations are guaranteed to be complete, and 4) functions can be easily parallelized.

If the specification of a type is symmetric, for a value of the type, we can determine which constructor has constructed the value by applying the discriminator to it, and select each component of the value by applying the corresponding normal selectors to it. This is exactly the way in which an abstractor works.

Prop. 1 *Suppose the specification of T is symmetric. The abstractor skeleton for T is*

abs_T(v:T) =
if *disc(v)=*d_1
$\mathbf{cons}_1(abs\_T_{1,1}(sel_{1,1}(v)), \ldots, abs\_T_{1,n_1}(sel_{1,n_1}(v)))$
else if *disc(v)=*d_2
$\mathbf{cons}_2(abs\_T_{2,1}(sel_{2,1}(v)), \ldots, abs\_T_{2,n_2}(sel_{2,n_2}(v)))$
$\vdots$
else $\mathbf{cons}_m(abs\_T_{m,1}(sel_{m,1}(v)), \ldots,$
$abs\_T_{m,n_m}(sel_{m,n_m}(v)))$

For example, the abstractor of *Stack* generated from its specification would be

abs_Stack(v:Stack) =
  if *isnew(v)* **new**
  else **push**(*abs_Stack(pop(v))*, *top(v)*)

4.1.2 Asymmetric Specifications

In practice, many specifications are asymmetric. The specification of type *Queue* in [6] is such an example. In [12, 14], we studied typical asymmetric types and identified conditions that guarantee the generation of abstractors. These conditions cover all the abstract data types specified in the Larch Shared Language Handbook [6], which include *Array, Bag, Graph, Map, PriorityQueue, Relation, Sequence, Set, Queue, Tree,* and *String.* Because of space limits, we only present one pattern to show the flavor of the work. All the proofs are omitted here and can be found in [12, 14]. First, we introduce a notation for abbreviating construction terms.

Def. 4 Let $f : T \times E \longrightarrow T$. For $x : T$; $e, e_1, \ldots, e_n : E$, $f^*(x, \langle e_1, \ldots, e_n \rangle)$ is defined by

$$f^*(x, \langle\rangle) = x$$

$$f^*(x, \langle e_1, \ldots, e_n \rangle) = f(f^*(x, \langle e_1, \ldots, e_{n-1} \rangle), e_n)$$
if $n > 0$

In this notation, the expression $f(f(\ldots f(x, e_1), \ldots, e_{n-1}), e_n)$ can be abbreviated as $f^*(x, \langle e_1, \ldots, e_n \rangle)$.

We now consider types with two constructors: $new :\longrightarrow T$ and $cons : T \times E \longrightarrow T$, where E is a base type.

By our notation, the construction term for a value v of such a type would be expressed in $\texttt{cons}^*(\texttt{new}, \langle e_1, \ldots, e_n \rangle)$. To convert v to the construction term, we must be able to select every e_i $(1 \leq i \leq n)$ from v through an iteration process on v [20].

Def. 5 A function $sel : T \longrightarrow E$ is called a **value-sensitive** selector, if, for every sequence $\langle e_1, \ldots, e_n \rangle$ there exists some i in $[1 \,..\, n]$ such that, for every permutation $\langle e'_1, \ldots, e'_n \rangle$ of $\langle e_1, \ldots, e_n \rangle$,

$$sel(cons^*(new, \langle e'_1, \ldots, e'_n \rangle)) = sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = e_i$$

Given $v = cons^*(new, \langle e_1, \ldots, e_n \rangle)$, a value-sensitive selector selects a component from v based on the component's value.

Thm. 1 *Let sel be a selector. If its equations are like:*

$$sel(cons(new, e)) = e$$
$$sel(cons(cons(x, e_1), e_2)) = \begin{cases} e_2 & \text{if } p(e_2, sel(cons(x, e_1))), \\ sel(cons(x, e_1)) & \text{otherwise.} \end{cases}$$

where p ($p : E \times E \longrightarrow Boolean$) is a comparison function on E: and if p is a total order on E, then sel is value-sensitive.

Intuitively, if *sel* meets the above condition, for a given $v = cons^*(new, \langle e_1, \ldots, e_n \rangle)$, $sel(v)$ will always pick the smallest element in $\{e_1, \ldots, e_n\}$, under the total order p.

Def. 6 Let $del : T \longrightarrow T$. *del* is called a **deletor** if for every sequence $\langle e_1, \ldots, e_n \rangle$,

$$del(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = cons^*(new, \langle e_1, \ldots, e_{l-1}, e_{l+1}, \ldots, e_n \rangle)$$

for some l, $1 \leq l \leq n$. That is, a deletor removes a component from a given value.

Def. 7 Let $sel : T \longrightarrow E$ be a selector and $del : T \longrightarrow T$ a deletor. They are a **complementary pair** (complementary to each other), if for any $v = cons^*(new, \langle e_1, \ldots, e_n \rangle)$, there exists l $(1 \leq l \leq n)$ such that

$$sel(v) = e_l \text{ and}$$
$$del(v) = cons^*(new, \langle e_1, \ldots, e_{l-1}, e_{l+1}, \ldots, e_n \rangle)$$

That is, $sel(v)$ and $del(v)$ work on the same component — the component selected by $sel(v)$ is the one deleted by $del(v)$.

Thm. 2 *For a selector $sel : T \longrightarrow E$ and a deletor $del : T \longrightarrow T$ if all the equations of sel and del are in the following form, they are a complementary pair.*

$$sel(cons(new, e)) = e$$
$$sel(cons(cons(x, e_1), e_2)) = \begin{cases} e_2 & \text{if } p(e_2, sel(cons(x, e_1))), \\ sel(cons(x, e_1)) & \text{otherwise.} \end{cases}$$
$$del(cons(new, e)) = new$$
$$del(cons(cons(x, e_1), e_2)) = \begin{cases} cons(x, e_1) \text{ if } p(e_2, sel(cons(x, e_1))), \\ cons(del(cons(x, e_1)), e_2) \text{ otherwise.} \end{cases}$$

where p is a total order comparison function on E.

For example, *head* and *tail* of the priority queue in [6] form a complementary pair.

Prop. 2 *Given a type T, if sel and del are a complementary pair and T has an equation $cons(cons(x, e_1), e_2) = cons(cons(x, e_2), e_1)$, then the abstractor skeleton for T is*

abs_T(v:T) =
  if *isnew(v)* **new**
  else $\texttt{cons}(abs\_T(del(v)), abs\_E(sel(v)))$

This proposition can be used to generate the abstractor of types like *Priority_queue*.

4.2 Generation from Implementations

Implementing an ADT requires representing its values using values of another type called the **implementation type** and defining its functions using the functions of the implementation type. The specifier would normally give an *abstraction function* [11, 27, 21, 25, 26] relating values of the implementation type to values of the ADT. We can easily generate abstractors from such abstraction functions.

```
Implementation Stack
Representation
  Stack = Record[buf:Array[Ele], ptr:Natural]
Abstraction function
  absF_Stack(r) = abs1(fetch.buf(r), fetch.ptr(r))
    where abs1(a,n) = if n=0 then new
      else push(abs1(a,n-1), fetch(a,n)) end end
Definitions
  new =
    store.ptr(store.buf(newRec, newArray), 0)
  push(s, e) =
    store.buf(s1, store(fetch.buf(s1), fetch.ptr(s1), e))
    where s1=store.ptr(s, fetch.ptr(s)+1) end
  top(s) = fetch(fetch.buf(s), fetch.ptr(s))
  pop(s) = store.ptr(s, fetch.ptr(s)-1)
  isnew(s) = ( fetch.ptr(s)=0 )
  size(s) = fetch.ptr(s)
```

Figure 3: Implementation of *Stack*

Figure 3 gives an implementation of *Stack* using *Record* with fields buf (of type *Array*) and ptr (of type *Natural*), where *Record* and *Array* have the same meaning as their counterparts in imperative languages like Pascal, but their operations are functional.

In [13, 14], we presented a notation for writing abstraction functions and presented an algorithm for transforming an abstraction to an abstractor. In generation from implementations, a particular implementation of T is given. The generator analyzes the relationship between T and the implementation type, and derives the abstractor for that implementation. As an example, the abstractor generated from the implementation given in Figure 3 would be

```
abs_Stack(r: Record[buf:Array[Ele], ptr:Integer])
= abs1(fetch.buf(r), fetch.ptr(r))
where abs1(a:Array[Ele], n:Integer) =
        if n=0 then new
        else push(abs1(a,n-1), fetch(a,n))
```

Since *abs_Stack* uses *fetch.buf* and *fetch.ptr* — functions of the implementation type *Record*, it is only applicable to that particular implementation. We may view it as a hidden function of the implementation.

4.3 Discussion

Generation from specifications is more desirable than generation from implementations, since abstractors generated from specifications are implementation-independent; the same abstractor works regardless of the implementation. However, specifications may not always provide the right functions for generating such an abstractor. Sometimes, details of an implementation need to be considered. Generation from implementations means there is always a way to generate abstractors, albeit one per implementation. The two methods are complementary to each other.

Using abstraction functions to generate abstractors does not place an excessive burden on the user. In the literature, much work calls for the user to provide the abstraction function for every implementation of a type. For example, abstraction functions are an integral part of *Alphard* programs [27]; they are an essential piece of information contained in internal module documents [25]; and they are strongly recommended to be provided as comments in *CLU* programs [21]. In addition, defining abstraction functions itself should not be a problem to the user, since when designing an implementation of a type, he understands the relationship between the implementation type and the type to be implemented.

5 Performance Issues

The performance of our approach depends on the size of construction terms and the efficiency of abstractors and interpreters. In this section we suggest some ideas to improve the performance.

Compressing construction terms can reduce the amount of information passed across networks. There are several ways to do so. Since construction terms themselves are ASCII strings, they can be certainly compressed by existing textual data compression techniques [9, 28]. If a construction term is of the form $f(f(\ldots f(new, a_{1,1}, \ldots, a_{1,m}), \ldots), a_{n,1}, \ldots, a_{n,m})$, then the function symbols f and new can be deleted from the term; thus our example of $push(push(push(new, 10), 11), 12)$ can be represented as $10, 11, 12$, similar to the conventional value-based representation.

At first glance, abstractors and interpreters seem inefficient, but a closer look reveals their performance could be acceptable. To communicate an ADT value in a heterogeneous distributed program, no matter what method is used, the components of the value must be selected and put in a flattened form before transmission, which must be parsed to reconstruct the value. Abstractors and interpreters exactly perform these functions. This suggests their performance could be roughly the same as other methods. IDL

work [18] used ASCII exchange representations, but showed that depending on the degree of similarity between the source and destination, a hierarchy of readers and writers for increasingly more efficient exchange representations could be automatically generated.

In our approach, abstractors are defined in a purely functional style. This sacrifices performance for clarity. One may explore ways to express abstractors in an imperative style, such as using iterators [20].

The main advantages of using term-based exchange representations are that they are abstract and converters can be generated. This provides high transportability and relieves programmers from coding converters by hand, thereby increasing programmer productivity and ensuring the correctness of converters. Thus, some loss in performance might be a reasonable cost. Where performance is a priority, low-level exchange representations should be considered.

6 Related Work

Many systems support communicating built-in type values in heterogeneous programs [1, 3, 4, 5, 8, 15, 17, 23, 24, 29]. They are mainly concerned with resolving system-level representation heterogeneity. There have been several work on communicating ADT values [10, 19, 16]; they use value-based exchange representations.

The idea of using term-based exchange representations was independently discovered by us and Blaine and Goldberg. After the first author of this paper formulated the basic idea, the second author discussed it with Purtilo, who referred us to an internal work note by Blaine and Goldberg [2]. As far as we know, their work and ours are the only ones using term-based exchange representations. Nevertheless, there are significant differences between their work and ours. In general, we are attempting to tackle the problem of communicating ADTs systematically, while they only mention the use of a term-based exchange representation. We go beyond their work to consider automatic generation of converters, communication between parties with mismatched implementations of the communicated data, and improvement of performance.

7 Conclusions

The major results of our research are *an abstract exchange representation* and *generation of converters.*

We have chosen construction terms as the exchange representation. The term-based representation is a mathematical notation, independent of any machine, operating system, programming language or user.

In-converters are easily generated, thanks to the use of the term-based exchange representation. We have developed two methods to generate out-converters (abstractors). Those methods would work in the following way. An abstractor generator stores patterns and abstractor skeletons of types which can be handled in a library. Given the specification of an ADT, the abstractor generator abstracts the pattern of the equations and matches the pattern against those stored in the library. If a match occurs, the abstractor is produced. If no match occurs, the abstractor generator would ask the user for an abstraction function, from which the abstractor is produced.

Our approach is tied to ADTs, and has limited use for communicating simple types.

Efficiency is important to the practicality of our approach, so it merits particular attention. We have suggested techniques to improve efficiency in the last section. But further work is still needed.

Our generation from specifications method for asymmetric types is effective on a case-by-case basis. More work is needed on looking for general conditions that guarantee automatic generation of abstractors for asymmetric types.

References

[1] B.N. Bershad, D.T. Ching, E.D. Edward, E.D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[2] L. Blaine and A. Goldberg. Interoperability of abstract data values. Technical report, Kestrel Institute, January 1991.

[3] J.R. Callahan and J.M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, SE-17(6):626–635, June 1991.

[4] K. Geihs and U. Hollberg. Retrospective on DACNOS. *Communications of the ACM*, 33(4):439–448, April 1990.

[5] P.B. Gibbons. A stub generator for multi-language RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, January 1987.

[6] J.V. Guttag and J.J. Horning. An LSL handbook. Systems Research Center, Digital Equipment Corporation, 1991.

[7] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[8] R. Hayes and R.D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(12):1254–1264, December 1987.

[9] G. Held and T.R. Marshall. *Data Compression: Techniques and Applications.* John Wiley & Sons Ltd., Rockville, MD, 1991.

[10] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[11] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[12] L. Huang and D.A. Lamb. Generating abstractors for abstract data types. Technical Report 92-331, Department of Computing and Information Science, Queen's University, Kingston, Ontario, July 1992.

[13] L. Huang and D.A. Lamb. Generating abstractors from abstraction functions. Technical Report 92-344, Department of Computing and Information Science, Queen's University, Kingston, Ontario, December 1992.

[14] Lin Huang. *Communicating Abstract Data Type Values in Heterogeenous Distributed Programs.* PhD thesis, Dept. of Computing & Information Science, Queen's University,Canada, September 1993.

[15] M.B. Jones, R.F. Rashid, and M.R. Thompson. Matchmaker: an interface specification language for distributed processing. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 225–235. ACM, 1985.

[16] G.E. Kaiser and B. Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2):201–264, April 1992.

[17] D.A. Lamb. *Sharing Intermediate Representations: The Interface Description Language.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1983.

[18] D.A. Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):267–318, July 1987.

[19] D.A. Lamb. An introduction to IDL. Technical Report 88-237, Department of Computing & Information Science, Queen's University, Kingston, Ontario, November 1988.

[20] D.A. Lamb. Specification of iterators. *IEEE Trans. Software Engineering*, 16(12):1352–1359, December 1990.

[21] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press and McGraw-Hill Book Company, Cambridge, MA/New York, NY, 1986.

[22] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen, September 1990.

[23] D. Notkin, A.P. Black, E.D. Lazowska, H.M. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[24] G.S. Novak, F.N. Hill, M.L. Wan, and B.G. Sayrs. Negotiated interfaces for software reuse. *IEEE Transactions on Software Engineering*, SE-18(7):646–653, July 1992.

[25] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report 90-287, Department of Computing and Information, and TRIO, Queen's University, Kingston, Ontario, September 1990.

[26] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag Berlin Heidelberg, Berlin and New York, NY, 1990.

[27] M. Shaw, editor. *ALPHARD: Form and Content.* Springer-Verlag New York Inc., New York, NY, 1981.

[28] J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Chichester, West Sussex, 1988.

[29] M.W. Strevell and H.G. Cragon. Data type transformation in heterogeneous shared memory multiprocessors. *Journal of Parallel and Distributed Processing*, 12(2):164–170, June 1991.