XDR Binding to LISP

Sergio Antoy

Portland State University
Department of Computer Science
P.O. Box 751
Portland, OR 97207

James Davidson

Cimflex Teknowledge Corp.
P.O.Box 10119
1810 Embarcadero Road
Palo Alto, CA 94303

Abstract

We describe a binding of the XDR protocol to the Common LISP programming language and the binding's routine's library implementation. The implementation solves compatibility problems arising in the exchange of structured data between modules coded in different programming languages and/or hosted by computing environments with different hardware architectures. Our facility is used for module interconnection within ABE, an environment for the development of distributed, heterogeneous systems, which is briefly described. Our XDR binding to LISP radically differs from other bindings previously proposed in that we use an explicit association mechanism from XDR types to LISP types, which contributes the several important characteristics of the implementation, and might be adopted for bindings to other languages. Complex application programs are easier to write and can efficiently overcome the assumption that a module must know in advance the type of a datum it is going to receive. Our binding retains full compatibility with other XDR bindings, e.g. to C and LISP, with increased power, flexibility, and simplicity of use.

Keywords: XDR, Data representation, Module interconnection, LISP, Protocol binding, Data compatibility, Data abstraction.

1 Introduction

Complex systems must cope with heterogeneity, including interoperation of modules or procedures written in different languages. Such modules must often exchange complex, structured data. One approach to the data-interchange problem in a multi-environment system is to define an external data-structuring standard and libraries of routines in each language or environment for converting to and from the external standard.

In this paper, we describe a data-structuring facility based on the XDR data representation standard [19]. This facility supports definition and transmission of complex data among modules written in LISP and C. As part of this work, we have defined a binding of the XDR protocol to LISP, and extended the protocol in a number of ways.

In Section 2 we present the motivation for such a datatyping facility. In Section 3 we review XDR. In Section 4 we review XDR's binding to the C programming language. In Section 5 we describe the binding that we have provided for the LISP language. In Section 6 we describe some extensions we have made to the protocol and the XDR overall approach. In Section 7 we briefly compare our work with other similar facilities. In Section 8 we review ABE, the context in which our implementation is used. Section 9 presents our conclusions.

2 Motivation

The following scenarios [19] motivate the necessity for an appropriate software interface for interconnecting heterogeneous modules.

The initial scenario consists of two simple C programs, *writer* and *reader*, running in a generic Unix machine. The *writer* program writes, in binary format on the standard output, the integer values 1 and 2. The *reader* program reads, in binary format from the standard input, two integer values and prints them, in decimal format, on the standard output. Piping the output of *writer* to *reader* produces the expected result — "1" and "2" are printed on the standard output.

Now the scenario is modified by the introduction of heterogeneity. The two programs are run on two different machines, say *writer* on a Sun and *reader* on a Vax. Piping is attained through the execution of a remote shell. Figure 1 shows the invocation command and its output. The result is not what a naive observer might have expected. An identical result would occur if each program were moved to the other machine.

```
sun% writer | rsh vax reader
16777216 33554432
sun%
```

Figure 1: Motivation for datatyping and heterogeneity.
Program *writer* runs on a Sun and writes the integer values 1 and 2 in binary format on standard output. Program *reader* runs on a Vax. It reads two integer values in binary format from standard input and prints them in ASCII decimal notation on the standard output. The command pipes the output of *writer* to *reader* through the execution of a remote shell.

To understand the result of Figure 1 it is necessary to analyze the form in which integer values are represented in the two programs. Both programs represent integers as 32-bit, 2's complement words. However, the order of the bytes in a word of each machine is reversed with respect to the other machine. Note that 16777216 is 2^{24} — when the value 1 is read by *reader* the least significant bit winds up in the 24th bit.

3 XDR

3.1 Overview

XDR is a device for solving incompatibility problems arising in interconnecting modules in heterogeneous environments. "Module" and "environment" are used here as generic notions. By *module* we mean a computing unit producing and/or consuming data, for example a FORTRAN subroutine, a Pascal procedure or a C or LISP function. By *environment* we mean the set of characteristics affecting the *internal* form in which typed data are represented within a module. Such characteristics include module language, language compiler, operating system, and machine architecture.

XDR defines a protocol for the representation of typed data in a form, called *external*, which is independent of the peculiarities of the environment of a module producing and/or consuming data. The external representation of a datum consists of a string of bytes. This representation will be described more precisely later. The XDR protocol presupposes the existence, for any environment, of an interface for accessing its functionality. For example, the C interface consists of a library of *routines* for transforming a datum from its internal representation into the external one and vice versa. Transformation and input/output are tightly coupled and we use them interchangeably in our discussion.

Data to be exchanged between interconnected modules are filtered by complementary XDR routines. Figure 2 shows this process. The producing module passes its outgoing data through an *encoding* routine so that the output after the filtering is in external form. The consuming module passes incoming data through a *decoding* routine which converts data from external representation into the representation of the consuming module environment. Although the representation of a datum may differ in the two modules, this process preserves the semantic content of the datum. True modularity is achieved since each module does not make any assumption about the environment of its partner modules. Changing the environment of a module does not require changing its interconnected modules in any way.

The *binding* between an environment or language and the protocol is the process of associating the protocol's functionality and types to concrete counterparts of the environment. This is a two-step operation: 1) external types (those defined by the protocol) are associated with types available in the environment's programming language, and 2) library routines for converting between internal and external representations are associated with environment's computing units. The latter operation requires assigning a name to each routine's implementation, enumerating its arguments or parameters and their types, specifying input/output conventions, and handling errors.

$$\boxed{M_{(A),i}} \xrightarrow{R_{(A)}(d)} \boxed{E_{(A),T}} \xrightarrow{R_{(X)}(d)} \boxed{D_{(B),T}} \xrightarrow{R_{(B)}(d)} \boxed{M_{(B),j}}$$

Figure 2: XDR overall approach.

Module $M_{(A),i}$ executes in environment A. It produces an instance d of type T which is consumed by module $M_{(B),j}$ executing in environment B. The representation of d in A, say $R_{(A)}(d)$, may differ from $R_{(B)}(d)$, the representation of d in B. An encoding routine, $E_{(A),T}$, depending on A and possibly on the type T, transforms $R_{(A)}(d)$ into $R_{(X)}(d)$, the external representation of d. A complementary decoding routine, $D_{(B),T}$, transforms $R_{(X)}(d)$ into $R_{(B)}(d)$. Although $R_{(A)}(d)$ and $R_{(B)}(d)$ may differ, the semantic content of d is preserved across the two environments by the complementary XDR routines, $E_{(A),T}$ and $D_{(B),T}$.

XDR provides the capability of converting any instance of an XDR type from an environment dependent representation to the external representation and vice versa. The transformation in one direction is the inverse function of the corresponding transformation in the opposite direction. We will see in the next sections that the implementation of this functionality is greatly affected by the characteristics of the programming language in which the implementation is coded.

The origins of XDR can be traced to Xerox's Courier external data format [4]. XDR is the backbone of Sun's remote procedure call facility [16, 20] XDR's types and the general approach to the interconnection problem are well suited to the philosophies of the C programming language and the Unix operating system. XDR's approach is relatively low-level and has a number of limitations, which however also contribute to its efficiency and flexibility. For example, to minimize traffic XDR does not transmit type information along with the data and the receiving module must know in advance which datatypes will be transmitted. Furthermore, XDR represents types on fixed-size units of 4 bytes, a choice that yields some efficiency improvements, discussed later, over other external representation formats, e.g. [10]. XDR is not a module interconnection facility [6, 11, 13, 15], rather it provides low-level capabilities on which module interconnection languages and systems can be based.

The technical details of the XDR definition can be found in [19]. In the next subsection we summarize the protocol types and outline the characteristics of the external representation.

3.2 Protocol Types

There are three main primitive or unstructured XDR types and three structured ones. The primitive types are: integer, floating-point number, and character string. Numbers are represented by either 4 or 8 bytes depending on the precision. For integers, the first byte of the representation is the most significant and bytes are encoded "little endian" style [2]. Signed integers are represented in 2's complement form. Floating-point numbers are represented according to the IEEE standard [9]. Strings are represented by two pieces of information: an integer representing the string length, followed by the sequence of the ASCII representation of the characters.

Structured types are: array—a unidimensional, indexed, bounded-size collection of homogeneous components; structure—an ordered, fixed size collection of heterogeneous components; and linked collections of homogeneous nodes.

These structured types are externally represented in terms of their individual components plus a minimum, if any, of additional information. For example, the representation of an array contains the number of array components. This information is necessary for the decoding routine to know how many array components to expect, and also facilitates dynamic allocation of storage for an array to be decoded.

Certain types commonly found in programming languages, such as boolean and enumerations, are not present in the XDR protocol. However, many of these types are easily mapped into XDR types. For example, the boolean values *false* and *true* are mapped into the XDR integer values 0 and 1 respectively. A non-trivial mapping is necessary for the type *pointer*. Pointers' values are memory addresses which are meaningless outside a specific environment, hence have no natural external representation. Pointers are externally represented only as constituents of linked collections.

The XDR solution consists in replacing pointers with the objects they point to and is subject to some penalty. A pointer is thought of as a variant structure (discriminated union) with a boolean selector. When the selector value is *false*, the corresponding arm is empty. This represents a null pointer. When the selector value is *true*, the corresponding arm consists of the object to which the pointer is pointing. This approach is satisfactory (and reasonably efficient) for linear lists and binary trees, but fails for linked structures with circularity or sharing [8].

4 C Binding

The binding of XDR types to the types of the C programming language is relatively straightforward. In the following we briefly describe how this problem is solved by one implementation [19].

4.1 Protocol

Signed and unsigned integers of various sizes are mapped into corresponding XDR integers of size large enough to avoid loss of information. Floating-point numbers are mapped into corresponding values in the IEEE standard representation. C arrays are inherently unidimensional, hence the mapping is direct. C *structs* are mapped component-wise into XDR structures, as are linked structures. Figure 3 goes into some details of the representation of strings.

4.2 Library

The XDR functionality is provided by C functions. There are two distinctive characteristics of this binding: 1) for each XDR type, a single C function

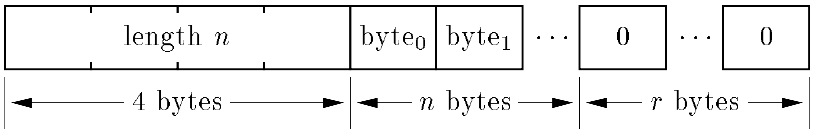

Figure 3: Graphic illustration of the external representation of the XDR type *string*. The length of the string is represented as an XDR integer denoted in the figure with n. The representation of a string is padded with null bytes. r is an integer between 0 and 3 such that $n + r \equiv 0 \pmod 4$.

is used for both encoding and decoding; 2) different C types require different functions which can be associated with the same XDR type. Functions return boolean values to indicate their success or failure to accomplish the required operation.

The first characteristic stems from engineering reasons. Since encoding and decoding are complementary operations, coupling both functionalities in the same piece of code improves reliability and maintainability and allows sharing of portions of the code. The amount of shared code is significant except for the most primitive types. The first argument of a function is a special XDR *stream*. These streams are either for input or for output, but not for both; thus an XDR function is able to infer from the stream characteristics whether encoding or decoding is required.

The second characteristic is a consequence of the relatively strong typing of the C language. Potentially a single XDR library routine could transmit data of several "independent" types, such as for example integer and string. However, the routine's formal argument to pass such data would have to be a union type. As the number and complexity of the independent types grow, using their union becomes inconvenient for the programmer and possibly wasteful. Having one routine for each C type fits well with the XDR philosophy of transmitting a minimal amount of data, in particular, with the assumption that the receiver knows what datatype is coming, and that therefore no type tagging information is required.

5 Binding XDR to LISP

5.1 Approach

The XDR binding to C is based on the association of a library routine to each XDR type. This approach, whose motivation is discussed in the previous section, is very "unLISP-like". The "typical" LISP user is likely to want access to the XDR functionality through a single routine, much in the same way, for example, that the LISP output functionality is accessed through the predefined function *print*. Unfortunately, an analogous user interface for XDR cannot be developed using only the types predefined by the LISP typing system because of certain ambiguities arising in the encoding and decoding process performed by XDR. To resolve this problem we associate a type specifier, called LISP-XDR type and described at length in the next section, to each object transmitted via XDR. In the following we introduce examples of type ambiguities in data representation and discuss the advantages of an explicit association of a type specifier to a data object.

The external representation of a four byte bit string in which the most significant bit of the first byte is 1 can be interpreted either as a negative integer or as a large (i.e. $\geq 2^{31}$) positive unsigned integer. This ambiguity is implicitly resolved in the C binding and other LISP bindings [21] by the particular function (XDR library routine) called for decoding the external representation. In our binding, the XDR library routine is the same for the two different integer types, and the ambiguity is resolved by associating a type specifier with the external representation. The specifier determines the interpretation of the encoded representation and controls the decoding process.

A similar ambiguity occurs in the encoding process. For example, the LISP object internally represented by the integer value 0 can be interpreted as an instance of both the XDR types *integer* and *boolean*. The discrimination between these alternatives is crucial, since according to the XDR protocol the external representation of the object differs for each interpretation. In fact, in LISP the boolean interpretation of 0 has value *true* which is externally represented as the integer value 1.

In our binding, the discrimination between various interpretations of a value is solved by using a type specifier explicitly associated with the value rather than by the library routine called for encoding the value. In other bindings, this association is implicit in the flow of control that leads to the execution of one library routine, or sequence of routines, rather than another. Thus the design choice of introducing type specifiers is a conceptual simplification, since it makes explicit a concept hidden in other bindings, makes code more declarative, since interfacing a program with our binding is considerably more data driven than interfacing a functionally equivalent program with the other bindings mentioned above, and it considerably simplifies the run-time type-check required when a data object is transmitted by a LISP implementation of XDR. We address this last point at the end of the next section after describing how type specifiers are implemented.

In bindings that do not use type specifiers the data protocol of structured types is ultimately defined by the sequence of calls to XDR library routines executed to encode and decode a datum. A type specifier reduces this sequence to a single, unique routine. For non-trivial types, this approach considerably increases software productivity, reliability, and maintainability.

5.2 Protocol

Our binding defines a new LISP type to be associated with each XDR type. LISP types associated to XDR types are called *LISP-XDR types*, are defined through the LISP form *deftype* (Figure 4) and referenced by a form called *type specifier* (Figure 5). For simple, unstructured types, the specifier of a type is a symbol such as *xdr-int* or *xdr-u-int*. Structured XDR types are parameterized and/or composite. The specifier for these types is a form consisting of a leading type specifier symbol and various param-

```
(deftype xdr-int ()
   `(integer ,(- (expt 2 31))
             ,(1- (expt 2 31))))
(deftype xdr-u-int ()
   `(integer 0 ,(1- (expt 2 32))))
```

Figure 4: Examples of LISP-XDR type definitions. Definition of the LISP types associated to the XDR types integer and unsigned integer. The form *deftype* defines a new, named type specifier in terms of other Common LISP type specifiers.

eters depending on this symbol. These parameters usually specify lengths or sizes or are other type specifiers. See Figure 5 for simple examples. Figure 9 shows the type specifier of *list*, an extension of the standard that we will discuss later.

```
(xdr-array xdr-int *)
(xdr-array (xdr-list (xdr-string *)) 12)
```

Figure 5: Examples of type specifiers. The first example specifies the LISP type associated with the XDR type array (of unbounded length) of integers. The length of the array is determined at encoding time by the encoding routine and is present in the external representation. The second example specifies an array of 12 (homogeneous) lists of strings of unbounded length. *list* is not a standard XDR type.

The LISP typing system is powerful enough to make the correspondence between LISP-XDR types and XDR types more precise than the correspondence between C types and XDR types. For example, the XDR type *integer* corresponds to various C types, such as *short*, *int*, or *long*. In LISP there is a perfect fit between the type *xdr-int* defined in Figure 4 and the XDR type *integer*. Every instance of the LISP type *xdr-int* has one and only one corresponding instance of the XDR type *integer* and vice versa. Thus, we take further advantage of LISP-XDR types to perform run-time type checking.

The need for this checking stems from the fact that, in contrast to C, the weak typing of LISP does not automatically guarantee that the request to encode an object as an XDR type is consistent with the object's value. Typing errors of this kind may not generate compile- or run-time messages, thus they may go undetected for a long time and, if detected, can be hard to locate in a distributed heterogeneous environment. Since we define LISP-XDR types by means of LISP type specifiers, we can determine the membership of a data object in a LISP-XDR type very easily with the LISP predefined predicate *typep*.

The predicate *typep* has two arguments: an object and a type specifier. On invocation, *typep* returns *true* if and only if the type of the value of the object satisfies the specifier. We have defined the LISP-XDR types in such a way that *typep* performs all the necessary tests, including range checking and other checks which do not arise in a strongly typed language, e.g. that all the components of an array are of the same type. The use of LISP-XDR types in this context makes, once again, code more efficient and considerably more declarative.

5.3 Library

The XDR functionality is provided by a single entry point, *xdrio*, for all conversion operations. *xdrio* has four arguments: *stream*, *specifier*, *object*, and *direction*. The last two arguments are optional. *Stream* is a LISP stream which may be either mono- or bi-directional. *Specifier* is a type specifier for the conversion to be performed. *Object* is a LISP object which usually is specified only for encoding operations. Decoded values are typically returned as values of function calls to *xdrio*. *Direction* specifies the direction of the request, i.e. input (decoding) or output (encoding). This is necessary only for cases in which the stream is bi-directional and an object is provided in the call.

The function *xdrio* performs a number of checks and bookkeeping operations and eventually invokes a type specific function depending on the leading symbol of the specifier. The specific function of an atomic type does any necessary conversion and calls a lower level byte input or output function. The specific function of a composite type decomposes in parallel both the object's value and its specifier into their components and makes a sequence of recursive calls to *xdrio*. For example, the conversion of an array of type t and size n results in one recursive call to *xdrio* to convert the unsigned integer n and in n recursive calls each of which converts one object of type t. These calls may generate new recursive calls to *xdrio*.

This approach differs radically from those of previously proposed bindings [19, 21]. For example, XDR's C users must code a function for each non-atomic type whose values are to be encoded or decoded. This function can be generated semi-automatically by the *rpcgen* utility [20], but the process still requires considerable user intervention and it is prone to errors. The binding of a type t to its encoding and decoding function occurs at compile-time. Changing the type t requires recoding the function and sometimes restructuring its calls. This requires sparse editing of application programs. By contrast, with our binding users need only define a type specifier. Changing a type to be converted simply requires updating its specifier.

The hierarchical approach of *xdrio* simplifies debugging and tracing, since any XDR activity is invoked from this one point. We have defined a set of global switches which can be toggled by the user. These switches control the kind and amount of information to be generated for monitoring XDR activities. For example, the internal and/or external representation of a value converted by *xdrio* can be traced on the **trace-output** stream. Optionally, the value's type specifier can also be traced. Tracing can be individually selected for the input and/or the output direction, and either for the user's calls to *xdrio* or for any call to *xdrio*, including recursive ones. Figure 6 traces a call to *xdrio*.

5.4 Architecture

The basis of our implementation is a set of small LISP modules. Each module is concerned with one XDR type. The module contains the definition of the corresponding LISP-XDR type, specific encoding and decoding functions, and additional forms for system configuration and testing. Under normal circumstances users do not interact with these modules. The functionality of these modules is available through higher level interfaces such as *xdrio*, which executes type checking, controls tracing, sets de-

```
XDRIO :OUTPUT (XDR-ARRAY XDR-INT *)
              #<ART-Q-3 15447572>
  XDRIO :OUTPUT XDR-U-INT 3
4BYTE :OUTPUT   0   0   0   3
  XDRIO :OUTPUT XDR-INT 7
4BYTE :OUTPUT   0   0   0   7
  XDRIO :OUTPUT XDR-INT -12
4BYTE :OUTPUT 255 255 255 244
  XDRIO :OUTPUT XDR-INT 5
4BYTE :OUTPUT   0   0   0   5
```

Figure 6: Example of tracing.
Printout generated on the stream **trace-output** during the execution of a call to *xdrio*. The call is a request to encode an array of integers of length 3. The values of the three components are: 7, -12, 5.

fault values, and performs additional bookkeeping.

We have coded a module for each standard XDR type and for a few extensions. Modules are structured such that an application program must load only the modules of the XDR types which are going to be converted. The appendix shows the module for the XDR type boolean. Although this module is unusually simple, its structure is representative of the "typical" module. The initial portion of the module is for system configuration. It declares the package to which the module belongs, any required module, and symbols to be imported or exported. The *pushnew* form updates a list containing the XDR types which can be converted. *Xdrio* checks this list before calling a conversion function.

The central portion of a module defines a LISP-XDR type and an equality predicate for instances of the type. The equality predicate, which is used in various contexts, depends on the interpretation of a value rather than the value itself. For example, in LISP the values 0 and 1 are equal when interpreted as boolean instances.

The following portion of the module is conditionally loaded only for testing purposes. We have found random testing useful for finding bugs during the code development and for verifying the installation of our implementation in new environments. The test is performed on either one environment or across two environments. A driver program randomly generates a type (specifier) t and an instance v of t. In a single environment v is encoded into a string stream and then decoded back. The result of the decoding is compared against v. Both encoding and decoding calls to *xdrio* are in the scope of t. Testing across two environments is similar, but the receiving environment is not in the scope of t, thus, since t is randomly generated, there is no straightforward way of decoding v. We describe testing involving several environments in the context of an extension to XDR's use discussed later. The generation of random type specifiers and instances is trivial for booleans, but more complex for structured types. For example, the random specifier of an array must contain the size of the array and the type of the components, which are both randomly generated. The type of the components may be a structured type as well, which implies recursive generation.

The final portion of the module contains the input/output function and other auxiliary functions. These perform the actual conversion between internal and external form. For example, the XDR representation of a boolean value consists of an unsigned integer, thus the auxiliary functions of the module for the XDR boolean type perform the transformation from boolean to unsigned integer and vice versa. For each LISP-XDR type, the input/output function has the same name as the leading symbol of the type specifier. This feature enables *xdrio* to dispatch type specific functions easily and without global information.

6 LISP Extensions

In this section we discuss some extensions, available in our binding, to XDR and its use in LISP. We have extended the XDR protocol with two new types: *symbol* and *(homogeneous) list*. These types are heavily used in LISP, thus the need to exchange instances of them arises frequently in distributed applications.

A LISP symbol is a data object that has three user-visible components: the *property list*, the *print name*, and the *package cell*. The external representation of the property list presents non-trivial problems. Since "Common LISP does not use a symbol's property list as extensively as earlier LISP implementations did" [17], we have not retained the property in the external representation. Thus, the external representation of a symbol consists of either 2 or 3 pieces of information: 1) the symbol's print name encoded as an XDR string; 2) the symbol's internment and visibility status encoded as an XDR unsigned integer; and 3) the symbol's package encoded as an XDR string. The value of the second piece follows these conventions: 0 when the symbol does not belong to any package, 1 when the symbol is internal to some package, and 2 when the symbol is external to some package. The third piece is absent when the status is 0.

LISP symbol is not a meaningful type in C, thus we expect that symbols will be exchanged only between LISP application programs. By contrast, lists are ubiquitous in programming. A homogeneous LISP list of elements of type t is the C equivalent of a pointer to a structure s consisting of a component of type t and a pointer to s. Accordingly, the external representation of the type *list* is equal to that of a C (discriminated) *union*. It consists of a variable number of structured data with two components: a boolean discriminant and one arm whose type is the type of the elements of the list. The arm value is present in the external representation if and only if the value of the discriminant is *true*. The standard XDR protocol does not contain a type *list* only because C is not able to process user-defined generic types. Our extension is very practical in LISP and is appropriate for languages with generic declarations, such as Ada, as well. Figure 9 shows the definition of the specifier for the type *list*.

The "extension" we discuss next is not about XDR functionality, but about the use of XDR made possible by type specifiers in LISP environments. Application programs exchanging data using some XDR bindings [19, 21] generally work on the assumption that the receiving module knows in advance the type of a datum it is going to receive. We call this approach *type-synchronous*. This approach stems partly from the XDR decision, opposite to ASN.1's [10] (briefly described in the next section), to not include type tags with the external representation. It also stems from the characteristics of the

Figure 7: Graphic illustration of the external representation of the XDR type *symbol* provided by our binding. *Print name* is the representation of an XDR string, the symbol's print name. *Visibility status* is the representation of an XDR integer encoding the symbol's visibility status as a value in $\{0, 1, 2\}$. *Package cell* is the representation of an XDR string, the symbol's package cell. This component of the representation of a symbol is present only when *visibility status* is not 0, that is, the symbol belongs to some package.

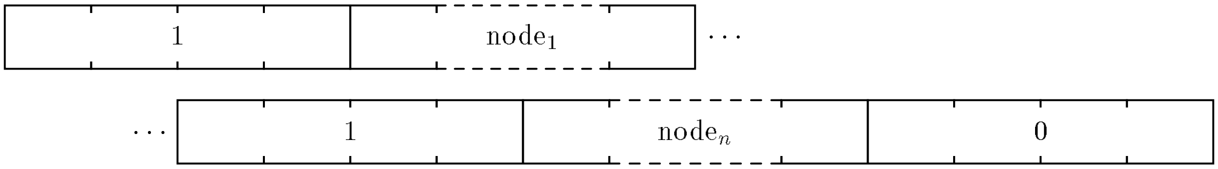

Figure 8: Graphic illustration of the external representation of the XDR type *list* provided by our binding. A homogeneous list is represented as an XDR discriminated union. The discriminant is represented as an XDR integer whose value is either 0 or 1. The value 1 indicates that a node follows, whereas the value 0 indicates the termination of the list.

environment, such as strong typing, and the binding, for example when data protocol is defined by a sequence of calls to library routines, that would make a different approach quite inconvenient.

Type-synchronism might be inadequate for some applications, and may limit some of the advantages of distributed environments. In an alternative approach, for example adopted by ASN.1, which we call *type-asynchronous*, the sending module is free to send data of different types in any order without previous agreement with the receiving module. This implies a more complex external representation and some loss of efficiency. We have capitalized on our binding and LISP's weak typing to build a simple and efficient interface, based on XDR, for type-asynchronous transmission. This feature is based on a simple idea hinging once again on the concept of type specifier.

The idea is to represent as an XDR datum the type of an XDR datum. The type of this type, supplied by our binding with the name of *xdr-type*, defines the type whose instances are the XDR type specifiers. This notion of "higher order type" is unfamiliar to C programmers, since they cannot use it in C and other similar imperative languages, but is common in some modern programming languages, e.g. ML [12]. The application of this concept is made trivial in our binding by the definition of *xdr-type* and a simple interface to a specialized pair of sending/receiving modules we have implemented for this purpose. The type-asynchronous sending module, similar to *xdrio*, is called with an object d and its type specifier s. It makes two standard calls to *xdrio* for output. The first call has s as object

```
(deftype xdr-list (elem-type)
  "An XDR-shippable homogeneous list."
  (let ((type-fn (gethash `(:list ,elem-type) *type-hash-table*)))
    (unless type-fn
            (setf type-fn  (gensym)
                  (gethash `(:list ,elem-type) *type-hash-table*)  type-fn
                  (symbol-function type-fn)
                  #'(lambda (object) (typep-elem-list object elem-type))))
    `(or null (and cons (satisfies ,type-fn)))))

(defun TYPEP-ELEM-LIST (object elem-type)
  (or (null object)
      (and (typep (car object) elem-type)
           (typep-elem-list (cdr object) elem-type))))
```

Figure 9: Definition of the specifier for the LISP-XDR type *xdr-list*.

The key form of the definition of the LISP-XDR type *xdr-list* is the last line of the definition of *xdr-list*. A complication arises because the predicate *satisfies* takes only one argument which must be a function of only one argument. For each type t of the elements of a list a different function is needed. The first time that the test to determine if an object is of type *list of type t* is performed, the function identified by *type-fn* in the definition of *xdr-list* is created and stored in a table. This function is acceptable to *satisfies* and checks that all the elements of a list have the same type t. Successive tests with the same type t retrieve the function from the table and use it against the object to be tested. The type *array* is handled similarly.

and *xdr-type* as specifier. The second call has s as specifier and d as object. The type-asynchronous receiving module makes two calls to *xdrio* too. The first call inputs the type specifier that is used in the second call to correctly interpret the external representation of the datum.

Type-asynchronous transmissions greatly simplify random testing across different environments. The sending environment randomly generates a type t and an instance v of t. v is sent in type-asynchronous mode, which implies that t is encoded and sent before v. The receiving module, after decoding the encoded representation of t, is able to decode the encoded representation of v. To complete the testing, the transmission occurs in the opposite direction for the comparison with the initial value of v. Thus, using type-asynchronous transmission, random testing across different environments is as simple and practical as random testing in a single environment.

Our approach to type-asynchronous transmissions seems to offer the best of two worlds. It is fully compatible with XDR, thus type tags can be avoided when they are not needed or where they are not practical, for example in C, and type tagging may be temporarily turned off or used more sparingly without changing either protocol or the supporting software. It is more efficient than the "tight binding" approach, for example adopted by ASN.1 and described later, where each datum has its own type tag. For example, if an array a of n components of type t is transmitted in type-asynchronous mode, we transmit t only once as a component of the descriptor of a, while tight binding approaches transmit t n times—once for each component of a. Finally, when efficiency is at a premium, further optimizations are possible. For example, when a large, known number of instances of the same type are going to be transmitted, their type descriptor can be sent only once. In strictly type-asynchronous transmissions a copy of the type tag is sent with each instance.

7 Related Work

An extensive comparison of external data formats is presented in [14]. Here we limit our comparison to ASN.1, which takes a different approach, and to the Symbolics binding of XDR, which though similar in scope differs substantiaslly in the implementation.

7.1 ASN.1

ASN.1 [10] provides a language specifying data types, and a set of rules for representing those data types in binary form. Like XDR, ASN.1 supports both primitive and constructed types. ASN.1 includes a macro facility, which extends the grammar of the language. Compared to XDR, ASN.1 is usually considered to be a richer description language. However, there are still a few types that cannot be specified in ASN.1, such as fixed length arrays.

ASN.1 differs from XDR in the methods for representing external data, in two significant ways:

1. Each ASN.1 datum is *tagged*, with the type of the included datum. This information is specified in the form of an integer, which is assumed to be meaningful to the receiver of the datum. The tag also includes a class field, indicating the context in which the type is defined.

2. An ASN.1 datum is encoded in the minimum number of octets. To support this minimal encoding, the length of the encoded datum is transmitted before the datum itself, again in the minimum number of bytes. (In certain cases an indefinite length can be specified, which means the data must terminate with an end-of-contents marker.)

Data are thus encoded in a *Tag-Length-Value* format. Constructed data are encoded in the obvious way: the *Value* field contains complete ASN.1 encoded data—i.e., embedded *Tag-Length-Value* forms.

These differences produce some potential efficiency differences between ASN.1 and XDR.

The fixed-length encoding of XDR allows potential speedups because of the match between XDR and the architecture of the underlying machines. XDR uses 32-bit encodings, which match a large class of hardware architectures. Conversely, the use of variable length encodings, which must be processed one octet at a time, can make the encode/decode operations more expensive.

The absence of tagging in XDR can result in a space saving in the encoded form, although this is offset by the space requirements of using fixed-length encodings.

Some form of tagging is required to support certain datatypes, such as discriminated unions. Further, tagging enables ASN.1 features such as optional elements in constructed types. In other circumstances, tags may be redundant. For example, the elements of a homogeneous list or array must be tagged in ASN.1, although the type is known in advance.

Our binding supports a selective form of type tagging, for the type-asynchronous transmissions described in the previous sections. This is different from the form used by ASN.1. In our case, a complete type descriptor, in the form of an XDR type specifier, is transmitted before the data. The data themselves are transmitted in standard XDR form, and decoded using the type descriptor. In the case of ASN.1, each datum at each level is encoded together with its tag and length, which leads to the performance issues mentioned above. Our approach gains some of the benefits of tagging, while retaining the efficiency of XDR.

Compared to ASN.1, the XDR language is more limited. For example, integers are generally limited to 32 bits. In practice, that has not proved to be a limitation in our applications.

ASN.1's generality and power have costs; the richness of the language can make it harder to construct specifications, and impede the development of tools to deal with it [10]. Further, some of the more complex datatypes – e.g., arbitrary precision integers – are not supported in many implementations.

7.2 Symbolics XDR

Symbolics [21] also provides an XDR binding to LISP, for Symbolics systems. It supports roughly the same set of data types as described here, including homogeneous lists.

That approach differs from ours in that it is specific to Symbolics platforms; our approach uses standard Common LISP functionality, and is therefore portable to any Common LISP implementation. Additionally, the Symbolics facility, which is oriented towards the needs of RPC, does not provide for the kind of type-asynchronous transmission described in the preceding section. Finally, the Symbolics implementation does not follow our approach of using a single top-level function for all calls to the xdr routines; again, in the context of RPC this is less necessary.

The Symbolics version does provide extra functionality in some ways; in particular, the user has somewhat more flexibility in being able to associate different LISP types with a given external datatype.

8 Applications

The binding we have described has been designed and implemented within the context of *ABE* [5, 7], a systems integration tool developed by Cimflex Teknowledge Corporation. ABE, among other things, addresses the development of complex, heterogeneous, distributed systems. ABE is founded upon the notion of *module*, a persistent, autonomous, event-driven object, which communicates with other modules through typed, directional ports. Module interfaces are independent of:

the module's implementation language; whether the module is running on a local or remote machine; and the machine's architecture. ABE currently supports modules on both Symbolics and Sun computers, in both LISP and C.

Modules' ports can be typed with XDR types. Modules are generally built around data abstractions consisting of structured objects with encapsulated behavior. These abstract types are implemented as Coral *objects* [1] or C++ *classes* [18]. Coral is an object-oriented extension of Lisp developed at Cimflex Teknowledge; it is a predecessor of the Common LISP Object System and it has similar functionality. Modules' ports are often typed with Coral objects and C++ classes. These are defined in terms of XDR types. ABE provides facilities for the transmission of instances of these through our binding implementation.

The demands of ABE went beyond those of RPC, which is why we worked out a solution that supports the type-asynchronous capability. However, in many cases the receiver-knows-what-is-coming assumption was valid, which is one reason we did not use full ASN.1. We favored XDR because it was already implemented for C, and only a LISP implementation had to be provided. The more restricted set of XDR datatypes was adequate for our needs.

Our implementation finds applications outside ABE as well. Some implementations of Common LISP, e.g. [3], support calls to subroutines coded in a different language. However, the types of the arguments allowed in the calls are sometimes severely limited. A facility based on our binding for allowing parameters of all the XDR types is relatively straightforward to implement. It only requires the primitive capability of the languages' implementations to exchange uninterpreted bytes or words. The XDR approach has some advantages over an implementation-dependent external call. In particular, this approach has more uniform coverage of types and independence of the direction of the transmission.

9 Conclusions

We have defined and implemented a LISP binding for the XDR protocol. The binding defines a new set of LISP types, the *LISP-XDR types*, and an interface to convert instances of these types between internal and external representations. The LISP implementation is robust and reliable, because the LISP type system permits a perfect match between LISP types and XDR types. It is also simple and flexible to use, because the LISP typing strategy does not force the arguments of a function to have a type. In comparison with the C version, the implementation of our binding is more compact, simpler to use, as reliable, and provides tracing and testing features.

Our binding offers minor extensions to the XDR protocol, and significant ones to the use of XDR. First, we have defined new XDR types which are frequently used in LISP application programs. Second, we have introduced the concept of XDR type specifier that makes application programs shorter, more declarative, and safer, and enables XDR to be use in type-asynchronous mode. This has two applications: it allows communication in cases where the receiving module does not have complete type information; and it allows run-time cross-environment type checking.

Our implementation is based on a set of modules with simple and uniform structure for handling specific XDR types and on higher level functions which provide a convenient interface for these modules. This architecture supports true modular configuration and the ability of the system to test itself. Our implementation is used, stand-alone, for multi-language data interchange and is a component of the ABE system integration tool.

Our contribution is not limited to an easy-to-use, complete, fully compatible, and extended binding of XDR to the LISP programming language. The proliferation of both new programming languages and heterogeneous, distributed applications will likely continue in the future. This implies that both new XDR bindings and new external data representation facilities will be proposed and implemented. Some of our design choices and implementation techniques might be appropriate or advantageous for these efforts.

10 Acknowledgements

The authors thank Jay Lark for his assistance in dealing with the subtleties of Common LISP. Comments and suggestions of Lee Erman have been incorporated in this paper

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, and the Defense Advanced Research Projects Agency under Contract F30602-85-C-0135.

During parts of the preparation of this work one author, Sergio Antoy, has been a member of the technical staff of Teknowledge, Inc., has been on leave of absence from the Consiglio Nazionale delle Ricerche and has been a permanent faculty member of the Virgina Polytechnic Institute and State University.

References

[1] Cimflex Teknowledge Corporation, Knowledge Systems Division. *CORAL Reference Manual*, 1990.

[2] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, October 1981.

[3] Allegro Common Lisp Reference Manual. Franz Inc., 1987.

[4] Courier: The remote procedure call proposal. *Xerox System Integration Bulletin*, XSIS 038112, Xerox Corporation, December 1981.

[5] L. Erman, J. Lark, and F. Hayes-Roth. Abe: An environment for engineering intelligent systems. *IEEE Trans. on Soft. Eng.*, 14-12:1758–1770, Dec 1988.

[6] N. Habermann and D. Notkin. Gandalf: Software development environment. *IEEE Trans. on Soft. Eng.*, 12-12:1117–1127, Dec 1986.

[7] Frederick Hayes-Roth, Lee D. Erman, Scott Fouse, Jay S. Lark, and James Davidson. Abe: A cooperative operating system and development environment. In Mark Richer, editor, *AI Tools and Techniques*, pages 323–356. Ablex, Norwood, NJ, 1989.

[8] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. on Progr. Lang. and Syst.*, 2-4:527–551, 1982.

[9] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, Aug 12 1985. The IEEE, NY.

[10] Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization, 1987. ISO 8824 and 8825, Information processing systems.

[11] M. Jones, R. Rashid, and M. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proc of the 12th Annual ACM Symp. on Principles of Prog. Lang.*, 1985.

[12] J. C. Mitchel and R. Harper. The essence of ML. In *15th ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Lang.*, pages 28–46, San Diego, CA, January 1988.

[13] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Comm of the ACM*, 31-3:258–273, 1988.

[14] C. Partridge and M. T. Rose. A comparison of external data formats. In E. Stefferud, O. J. Jacobsen, and P. Schicker, editors, *Message Handling Systems and Distributed Applications*. Elsevier Science (North-Holland), Amsterdam, 1989.

[15] J. Purtilo and J. Atlee. Program reuse by interface adaptation. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 208–217, 1990.

[16] RPC: Remote procedure call protocol specification version 2. ARPA Network Information Center, June 1988.

[17] G. L. Steele Jr. *Common LISP: The Language*. Digital Press, 1984.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1987.

[19] XDR: External data representation standard. RFC1014. Sun Microsystems, Inc., Network Information Center, June 1987. Also available as External Data Representation: Protocol Specification, in *Network Programming*, March 1990.

[20] Rpcgen Programming Guide, in *Network Programming*. Sun Microsystems, Inc., March, 1990.

[21] Symbolics Inc. The Remote Procedure Call Facility. *Networks*, January 1990.

Appendix

Module for the LISP-XDR type *xdr-bool*. The modules of our binding's implementation comprise four major components: 1) declarations for system configuration, 2) a LISP-XDR type definition and equality predicate, 3) random specifier and instance generators (conditionally loaded for testing), and 4) conversion and input/output functions.

```
(in-package :xdr)
(export '(xdr-bool) :xdr)

(pushnew 'xdr-bool *LOADED-XDR-TYPES*)

(deftype XDR-BOOL ()
  '(member nil t))

(defun EQUAL-XDR-BOOL (type bool1 bool2)
  "Returns non-nil if the two boolean values are equal."
  (declare (ignore type))
  (if bool1 bool2 (not bool2)))

#+:testing-xdr
(defun XDR-BOOL-RANDOM-DESCRIPTOR ()
  "Returns a random boolean descriptor."
  'xdr-bool)

#+:testing-xdr
(defun XDR-BOOL-RANDOM-INSTANCE (&optional type)
  "Returns a random boolean instance."
  (declare (ignore type))
  (zerop (random 2)))

(proclaim '(inline bool-to-u-int u-int-to-bool))

(defun BOOL-TO-U-INT (object)
  "Converts a boolean to an unsigned integer."
  (if object 1 0))

(defun U-INT-TO-BOOL (object stream)
  "Converts an unsigned integer to a boolean."
  (case object
    (0 nil)
    (1 t)
    (t (xdr-io-error 'NOT-A-BOOL stream object 'xdr-bool :input))))

(defun XDR-BOOL (stream direction type-descriptor object level)
  "XDR I/O for booleans."
  (declare (ignore type-descriptor level))
  (ecase direction
    (:output (4b-output stream (bool-to-u-int object)))
    (:input (u-int-to-bool (4b-input stream) stream))))
```