Facilitating Mixed Language Programming in Distributed Systems

ROGER HAYES AND RICHARD D. SCHLICHTING, MEMBER, IEEE

Abstract—**An approach for facilitating mixed language programming in distributed systems is presented. It is based on adding a generic remote procedure call facility to each language, and the use of a type system to describe procedural interfaces, as well as data to be transferred between procedures. This type scheme also specifies a machine-independent representation for all data. By defining standard mappings for each programming language, the data conversions required for cross-langauge calls may be performed, automatically in most cases, by active agents that provide the interface between program components written in different languages. When necessary, explicit control of the conversation is possible. A prototype implementation of a system based on this approach has been constructed on a collection of machines running Berkeley UNIX®.**

Index Terms—**Data type, distributed systems, mixed language programming, programming languages, remote procedure call.**

I. Introduction

MANY benefits would result if it were easy to invoke a subroutine written in a language different from that of the calling program. Such a facility would broaden the range of standard library routines available to the programmer. It would also allow programmers the freedom to use multiple languages in a single program; the programmer could write each procedure in a language best suited for its task, without worrying about what languages might interface with it. An additional measure of flexibility is achieved if procedures can be placed on different machines in a distributed system. For example, this could open the way to a data analysis program in which the numerical portions are written in Fortran and run on a supercomputer, while the user interface is written in Smalltalk and runs on a user's workstation.

Unfortunately, there are many obstacles that must be overcome before interlanguage procedure invocation can be used in a distributed system. Among the more important are problems that arise because of different machine representations of data, diverse collections of data types in different languages, and incompatible language runtime systems [7]. Included in the latter are differences in such items as storage management schemes, input/output, and parameter-passing conventions.

In this paper, we outline an approach to constructing distributed mixed language programs based on adding a generic remote procedure call facility to each language involved. Specifically, we describe the *Mixed Language Programming* (MLP) system, which employs a distinct process for each program component and a data description language called the *Universal Type System* (UTS) language. This language is used for specifying procedural interfaces, and for describing and representing values transferred between procedures. Our use of multiple processes mitigates the problem of incompatible run-time systems and allows the program to be distributed, while the UTS language solves the problem of differing data representations and collections of types.

The MLP system requires specialized program processing and use of an interprocess communication facility supplied by the underlying operating system. Before invocation, the called procedure is made an active entity; this is achieved by initiating an "envelope" process that encases the procedure. Then, when the call is made, the arguments are translated from the invoking language into UTS format, transferred using interprocess communication to the envelope process of the invoked procedure (potentially across machines), then translated into the language of the invoked procedure. Type checking is performed by comparing the UTS types specified for the parameters with the UTS types specified for the arguments. In addition to handling arguments for which interlanguage translation is straightforward (e.g., integer), MLP also has provisions for more complicated situations (e.g., passing records to a Fortran procedure).

II. The MLP System

A *mixed language program* is a program written in two or more programming languages. Such programs consist of several *program components*, where each program component is composed of one or more procedures written in the same language. The language used to implement a component is called the *host language* for that component. Informally, a component can be thought of as being the unit of compilation, i.e., there is generally one object module per program component.

The *MLP System* is a collection of conventions and programs that allow procedure invocations to be performed with value/result semantics across components of a mixed

Manuscript received May 31, 1985. This work was supported by the Air Force Office of Scientific Research, Air Force Systems Command, United States Air Force, under Grant AFOSR-84-0072. The equipment used in the project was provided by the Department of Defense University Research Instrumentation Program (URIP) under Grant AFOSR-85-0089.

The authors are with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

IEEE Log Number 8717406.

®UNIX is a registered trademark of AT&T Bell Laboratories.

language program in a distributed system.[1] Specifically, the MLP system is composed of:

1) The UTS language
2) Language bindings
3) Agents
4) Interface specifications
5) MLP translators
6) The MLP linker.

Facilities for process creation and interprocess communication required for the implementation of our approach are assumed to be provided by the operating system(s) resident on the machines in the network.

A. The UTS Language

The UTS language is a data description language that contains two parts. The first part is a collection of types and type constructors. The types include standard types such as integer, float, and string; the type constructors allow the specification of aggregates such as records and more complex entities such as functions. The combination of these allow specification of most types required to support intercomponent communication. The following are examples of legal UTS types.

integer
float
byte[7]
string[12]
array[10,5] **of** integer
record{integer,float,float}
prog(**var** integer, **val** float) **returns**(float)

The second part is a simple language for constructing type expressions or *signatures*[2] that denote sets of types. This can be done by using the alternation operation "or", the symbol "−" to indicate one unspecified bound in an array or string, the symbol "*" to indicate an arbitrary number of such unspecified bounds or an arbitrary number of arguments, and the universal specification symbol "?" to denote the set of all types. Section IV-A describes the use of such type expressions to specify procedures that, for example, can accept arguments of more than one type. The Appendix contains an annotated grammar for UTS types and type expressions.

Values of UTS types are stored in a machine- and language-independent fashion. In general, the representation of a data item contains both the data and a tag indicating the type (and length) of the data. This representation is optimized in certain cases. For example, if the type of each element in an array is identical, storage is conserved by storing only a single indication of the type of the elements, rather than tagging each individual element. Conversion back into the standard representation is handled by the system automatically when required.

[1]Calls within a component are handled by the calling conventions of that component's host language.

[2]Our use of the term signature is similar, but not identical, to its use in the definition of programming languages such as Russell [5] and SR [1].

Two Way Mappings

Fortran type	*UTS type*
INTEGER	integer
DOUBLE PRECISION	float
CHARACTER*n	string[n]
LOGICAL	bool
COMPLEX	record {float, float}
X name(n)	array [n] of x
Y FUNCTION name(X)	prog (x) returns (y)
SUBROUTINE name(X)	prog (x)

Fortran to UTS Mappings

Fortran type	*UTS type*
REAL	float

UTS to Fortran Mappings

UTS type	*Fortran type*
byte	represented as INTEGER
record	—
error	represented as INTEGER
null	—

Fig. 1. Fortran language binding.

The tagged nature of the representation means a data element is, in effect, self-describing. Benefits of such a representation include enhanced reliability in data transfer and the potential for programs to handle values whose types are not completely predetermined. As we shall see, this latter situation can occur when type expressions are used in procedure specifications.

B. Language Bindings

For each host language, a *language binding*[3] is defined consisting of a set of conventions specifying the mappings between the host language types and those of the UTS language. For example, the language binding for Fortran appears in Fig. 1. Two meta-variables x and y are employed in this figure; when capitalized, a meta-variable represents the Fortran type; when appearing as a small letter, it represents the corresponding UTS type.

There are three parts to this binding. The first contains mappings in which the two languages agree, so that each type has a direct counterpart in the other type system. The second contains mappings for Fortran types that do not fall into the first category and the last contains mappings for UTS types that have no exact Fortran equivalent. Note that some of the entries in the third section are marked as merely representations. This indicates that the Fortran type listed is used only to store the data and that the UTS data should not be thought of as obeying the semantics of the Fortran type. The dashes indicate UTS types that cannot be mapped directly into Fortran; the means to handle such types are discussed in Section IV.

C. Agents

For each language binding, code is needed to implement the desired mappings and to act as the interface between program components. These functions are performed by the envelope process of a program component;

[3]The term is borrowed from graphics standards terminology [2].

this process is called the *agent* for the language binding. Each such agent contains two collections of routines: the *outgoing* routines to handle calls to procedures outside the component and to translate arguments from the host language into UTS, and *incoming* routines to handle invocations of procedures and to translate arguments from UTS into the host language.

The agents associated with program components communicate to implement the cross-language call. Specifically, the agent of the invoking component translates the arguments for the invocation into UTS format and uses interprocess communication to send them to the agent of the component containing the invoked procedure. This agent then translates the arguments from UTS into host language types and invokes the associated procedure. Return values are handled similarly. The coupling of the agent with its program component to give the code for the process implementing the component is performed by an MLP translator, while resolving names to translated components is the duty of the MLP linker. Both are described below. The sequence of actions in a cross-language call is illustrated in Fig. 2. Note that each component could be executing on a different machine.

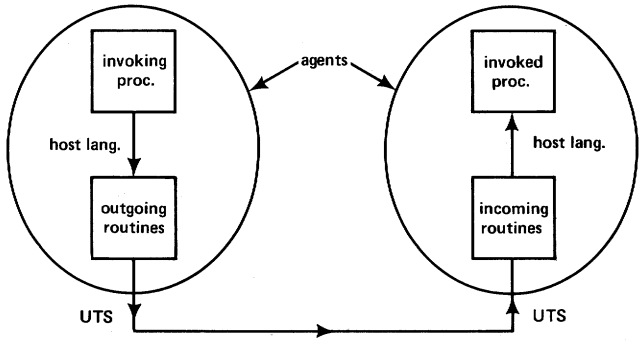


Fig. 2. Sequence of actions in a cross-language call.

The agent for a particular language binding is typically written in a combination of that language and a system programming language. This use of multiple languages may be necessary since the agents need to be able to perform low level data transformations, yet must be callable from the host language. While this means that the implementation of the agents does in some sense constitute a mixed language program in its own right, note that 1) the interface between the two languages is confined to the agent, and 2) the problem must be solved only once for each language binding, rather than for each specific mixed language program. Thus, implementing a fixed set of interface routines gives access to a potentially unbounded set of languages and procedures.

D. Interface Specifications

The *interface specification* of a program component consists of descriptions of the interface of each procedure contained in the component that is to be available for external invocation, together with descriptions of the interface expected for each procedure from another component that could be invoked. In other words, the interface specification is a list of interfaces "exported" from the component together with a list of "imported" interfaces; these lists are analogous in many ways to the import/export lists supported by many programming languages [1], [20].

The imported and exported interfaces are specified in the UTS language as signatures that describe the number and types of the arguments to the given procedure. In particular, the type constructor **prog** is used to construct an *export signature* or *import signature* of the form

prog(*param_type*, *param_type*,. . .) **returns** (*result_type*)

where

param_type ::= {**val** | **res** | **var**} *UTS_exp*
result_type ::= *UTS_exp*
UTS_exp ::= any UTS expression

The qualification **val, res,** or **var** corresponds to the standard parameter-passing notions of copy-in, copy-out, and copy-in/copy-out, respectively; if no qualification is specified, **var** is assumed. The **returns** clause is optional and describes the type of the value returned by a function.

The interface of an exported procedure is specified within the component by including a declaration of the form

export "procname" <export signature>

where "procname" is the name of the procedure to be exported. Such a declaration usually appears in or near the host language procedure declaration. The form of the declaration of an imported procedure's interface is analogous.

import "procname" <import signature>

Import declarations can usually appear in any area of global scope. The actual syntax of these declarations and their placement in the source code depend on the host language.

As an example of the use of these declarations, consider a Fortran procedure "example1" whose interface is to be exported. The beginning of this procedure might then appear as follows.

```
SUBROUTINE example1(i,y,intarray, floatarry)
  EXPORT "example1" prog(var "i" integer, var "y" float, var "intarray" array[-] of integer,
                         var "floatarray" array[-] of float)
  INTEGER i, intarray[*]
  REAL y, floatarry[*]
```

If this routine, in turn, makes an intercomponent call, then the interface of the imported routine would also be declared.

Export declarations can be omitted if the MLP translator for the host language can infer the signature using the source code for the component and the language binding. The source code for the entire component is not even necessary—it is sufficient that the procedure declarations be available, as would typically be the case even for library routines for which complete source code is not provided. The signature should be explicitly specified in cases where such automatic translation is impossible or the resulting signature inadequate. The former would include situations such as Fortran procedure that accepts a record as argument (as discussed in Section IV), while an example of the latter would be an Icon routine [10] whose signature is to include the types of the arguments.

E. MLP Translators and the MLP Linker

MLP programs are translated and linked using specialized programs that supplant the standard compilers, translators, and linkers. There is one *MLP Translator* for each host language that takes as input a program component with interface specifications and produces as output a program component in object form. The *MLP Linker* is then invoked with the names of these translated components in order to produce an executable MLP program.

In addition to performing the tasks of the standard language translator, an MLP translator performs several other basic functions.[4] First, it inserts the code for the language-specific agent that coordinates execution of the program with the agents of the other components. Second, it finds the import and export signatures in the component and places them at a known location in the emitted file. These signatures are then used by the linker to, among other things, perform type checking.

The final function of the translator is to change calls to procedures in other components into calls on the local agent. This is done by replacing each call to an external routine with a host-language dependent sequence of statements. The cornerstone of this sequence is a call to the agent's outgoing routine mlp_invoke, which accomplishes the external invocation. This call may be made directly from the host language or may need to be made by an escape from the language. The former is possible if the host language is sufficiently flexible to permit an argument list of variable length and type, while the latter will be necessary for languages such as Pascal where strict type checking must be circumvented. The statements surrounding the call perform whatever actions are necessary to pass values in both directions between the agent and the host language. For example, if the calling program is written in Lisp, the statements following the call take the return values from the list returned by mlp_invoke and copy them into the appropriate variables; this action reconciles Lisp's call-by-value rules with the value/result semantics of a MLP cross-component invocation.

The task of mlp_invoke itself is to communicate the arguments to the envelope process of the requested procedure and receive the reply. The reply will contain the return value of the called procedure (if any) and the values of arguments that need to be copied back into the variables supplied in the call. These arguments are those that have been specified with qualifiers **var** or **res** in the export signature. The way in which mlp_invoke performs these functions is, of course, highly dependent on the operating system and interprocess communication mechanism being used.

The final part of the system, the *MLP linker*, takes a list of program components that form a mixed language program and 1) performs static type-checking of the arguments and parameters of the inter-component calls, and 2) inserts prologue and epilogue code into the main component to ensure successful initialization and termination of the program. The first is done by determining whether the export signatures associated with the invoked routines match the import signatures supplied for the same routines in the invoking program components.

While it is intuitively clear what it means for a match to occur in simple situations (i.e., when the signature are identical), the richness of the UTS language requires a more formal notion. Recall that a signature describes a set of legal data types. An import signature matches, or *unifies*, with the corresponding export signature if and only if the set denoted by the former is a subset of the set denoted by the latter. This ensures that if the two signatures unify, then an object passed as an argument to an invocation will be legal given the interface specifications of the invoked routine. The idea that signatures may describe more than a single type is important for routines that can accept polymorphous input and is discussed further below.

The second function of the MLP linker is to insert prologue and epilogue code into the program component containing the code for the main program. The purpose of the prologue code is to initiate on the correct machines the processes that compose the program and to establish interprocess communications links, while the epilogue code sends a notification to each constituent process upon termination of the program. This notification is used as the cue to terminate the process.

The details of the prologue and epilogue code depend heavily on the facilities provided by the operating system. As a typical example, consider a network of machines running Berkeley UNIX. One possible organization of MLP on such a system would have a server process on each machine to handle initiation of processes and establishment of network addresses ("sockets") for agents. Prologue code in the main component would use the servers to initiate the processes for the called components. The server would allocate a socket address, initiate the required process, and return the socket address to the invoker. The called process, which encapsulates the procedure, would listen at the specified socket for invoca-

[4]An MLP translator can be implemented either by adding a preprocessing phase to the standard translator, by modifying the standard translator itself, or as a combination of the two.

tions. mlp_invoke would then perform the call by employing the "sendto" system call to deliver the argument packet to the called process. The epilogue code would send a termination message to each socket upon termination of the program.

III. An Example

To illustrate more clearly the way in which the components of the MLP system interact, consider the following typical sequence of steps that would be used to write and execute a mixed language program in which a procedure written in Franz Lisp [9] invokes the Eispack Eigenvalue routine *cg* written in Fortran[5] [16]. First, interface specifications in the UTS language must be determined for both *cg* and the Lisp program; while this could be performed automatically by the MLP translators, for expository purposes we describe the process as if done explicitly by the programmer. In Fig. 3, we show the beginning of the source code for *cg*; in Fig. 4, the Fortran declaration of the procedure is repeated, augmented by the export declaration with its UTS signature. A Lisp function that might be used to facilitate the invocation of *cg* within a Lisp program appears in Fig. 5. The signature in the import declaration for *cg* in the Lisp component would be similar to the signature in the export from the Fortran component, although it is likely that the names of the parameters would be omitted. Also, the exact syntax of the declaration would mirror that of Franz Lisp rather than Fortran.

The next step is to submit the augmented components to the MLP translators for the respective host languages. For the purposes of discussion, we will assume that the MLP-related functions are implemented as a separate preprocessor to the standard translator. For *cg*, this preprocessor will first locate the interface specifications, and then, using these signatures, construct an agent. This agent will receive invocations of *cg* from other agents, translate the arguments from the UTS form supplied to it by the other agents into Fortran data types and representation, and then perform a normal Fortran call to the *cg* routine. Note that this preprocessing need take place only once, most likely as part of the installation of the Eispack routines.

The process for the Lisp program component is similar. However, Lisp is so flexible that we can do most of the work of the agent in the host language. In particular, the MLP translator can define a macro "cg" that expands to

```
(mlp_invoke
  'cg
  '(dim dim ar ai wr wi matz zr zi t1 t2 t3 ierr)
  cg-sig)
```

where the value of cg-sig is a representation of the signature for "cg", used to guide the argument translation and copying performed by mlp_invoke. Note that mlp_invoke must be given an unevaluated argument list because the values returned for **var** parameters must be copied back into the corresponding Lisp variables.

[5]Such an interaction might be useful, for example, in a symbolic algebra package such as Macsyma [14] in which numerical calculations are supported, but are not considered the primary function.

```
      SUBROUTINE cg(nm,n,ar,ai,wr,wi,matz,zr,zi,fv1,fv2,fv3,ierr)
C
      INTEGER n,nm,is1,is2,ierr,matz
      REAL ar(nm,n),ai(nm,n),wr(n),wi(n),zr(nm,n),zi(nm,n),fv1(n),fv2(n),fv3(n)
C
C     This subroutine calls the recommended sequence of subroutines from the Eigensystem
C     subroutine package (Eispack) to find the Eigenvalues and Eigenvectors (if desired)
C     of a complex general matrix.
C
C     On input-
C
C       nm  must be set to the row dimension of the two-dimensional array parameters as
C       declared in the calling program dimension statement,
C
C       n  is the order of the matrix  a=(ar,ai),
C
C       ar  and  ai  contain the real and imaginary parts, respectively, of the complex
C       general matrix,
C
C       matz  is an integer variable set equal to zero if only Eigenvalues are desired,
C       otherwise it is set to any non-zero integer for both Eigenvalues and Eigenvectors.
C
C     On output-
C
C       wr  and  wi  contain the real and imaginary parts, respectively, of the Eigenvalues,
C
C       zr  and  zi  contain the real and imaginary parts, respectively, of the Eigenvectors
C       if matz is not zero,
C
C       ierr  is an integer output variable set equal to an error completion code described
C       in section 2b of the documentation.  The normal completion code is zero,
C
C       fv1, fv2, and fv3  are temporary storage arrays.
C
```

Fig. 3. Beginning of *cg* routine.

```
      SUBROUTINE cg(nm,n,ar,ai,wr,wi,matz,zr,zi,fv1,fv2,fv3,ierr)
C
      EXPORT "cg" prog ( val "nm" integer, val "n" integer, val "ar" array[-,-] of float,
                         val "ai" array[-,-] of float, res "wr" array[-] of float,
                         res "wi" array[-] of float, val "matz" integer,
                         res "zr" array[-,-] of float, res "zi" array[-,-] of float,
                         res "fv1" array[-] of float, res "fv2" array[-] of float,
                         res "fv3" array[-] of float, res "ierr" integer)
C
      INTEGER n,nm,is1,is2,ierr,matz
      REAL ar(nm,n),ai(nm,n),wr(n),wi(n),zr(nm,n),zi(nm,n),fv1(n),fv2(n),fv3(n)
```

Fig. 4. Fortran header and associated export declaration.

Processing by the MLP linker is the final step before this mixed language program is available for execution. First, the linker must determine if the signatures of *cg* supplied with the code for *cg* and the Lisp program unify. They do trivially for this example. Second, the linker inserts prologue and epilogue code in the Lisp program component to handle initiation and termination of the two processes. As mentioned in the previous section, the exact details of this code are system dependent.

Once this mechanism has been set up, execution of each cross-language call proceeds as shown in Fig. 2.

IV. More Sophisticated Facilities

The MLP mechanisms described above are sufficient for handling cross-language calls in which argument types in the interface specification translate in a straightforward manner into types of the host language. However, there are situations where a simple mapping is not sufficient. For example, it would be nice to be able to write a single routine that returns the summation of values in an array

```
; a helper function, used to call the imported EISPACK routine cg.
; it does temp and result allocation, and argument and error-return checking.

; use-cg finds the eigenvalues and (if matz is nonnil) the eigenvectors of a complex general
; matrix. The first two arguments, ar and ai, are ; respectively the real and imaginary
; parts of the input matrix. They must be square flonum-block arrays.

(defun use-cg (ar ai matz)
  (prog (dim wr wi zr zi t1 t2 t3 ierr ardims)
        (cond                   ; determine if ar, ai are arrays
          ((not (arrayp ar))
                (error "must be array"))
          ((not (arrayp ai))
                (error "must be array")))
        (setq ardims (arraydims ar))
                                ; get list of ar's dimensions
        (cond                   ; determine if ar square flonum-block array
          ((!= (length ardims) 3)
                (error "must be rectangular array"))
          ((!= (cadr ardims) (caddr ardims))
                (error "must be square array"))
          ((!= (car ardims) 'flonum)
                (error "must be flonum-block array")))
        (setq dim (cadr ardims))
                                ; size of input arrays
        (cond                   ; determine if ar, ai have identical type and shape
          ((not (equal ardims (arraydims ai)))
                (error "arrays must have same size and type")))
        (*array 't1 'flonum-block dim)    ; create temp arrays
        (*array 't2 'flonum-block dim)
        (*array 't3 'flonum-block dim)
        (*array 'wr 'flonum-block dim)    ; create result arrays
        (*array 'wi 'flonum-block dim)
        (cond
          (matz
                (*array 'zr 'flonum-block dim dim)
                (*array 'zi 'flonum-block dim dim))
          (t
                (setq zr nil)
                (setq zi nil))
        )
                                ; do the call
        (cg dim dim ar ai wr wi matz zr zi t1 t2 t3 ierr)
        (cond                   ; determine if call worked
          ((not (zerop ierr))
                (error "EISPACK error" ierr)))
        (return (list 'wr 'wi 'zr 'zi))
  ))
```

Fig. 5. Lisp function to invoke *cg*.

where the type of the array can vary. Such a routine could then sum an integer array on one invocation, a float array on a second, and a complex array on a third. There are only a few languages (e.g., Icon) that can easily handle such polymorphous input. As a second example, it would be nice to be able to write a Fortran procedure that could accept an argument of UTS type **record** (generated, for example, by an invocation written in Pascal), even through the language does not include a record type constructor.

The MLP system supports the construction of such general purpose routines in two ways. First, the UTS language allows a signature to denote a set of types rather than simply a single type. This allows interface specifications in which the type of an argument can vary from call to call. Second, utility routines are defined that can be invoked from the program to determine the type of an argument and to manipulate values that do not translate directly into host language types. These routines allow the programmer to control exactly how values of such types are translated and when they are to be accepted into the program.

A. Underspecification

Signatures in the UTS language can denote more than a single type by using the alternation construct "or", the universal specification symbol "?", or the unspecified bounds symbols "−" and "*". For example, the following type expressions denote the indicated sets of types.

Type Expression	*Denoted Set*
integer or float	all integers and floats
string[3−]	strings of length 3 or more
array[−] **of** (integer or float)	arbitrary size arrays containing integers or floats
record{integer, ?}	records of two fields, the first of which is integer
prog(**val** integer or float,*)	routines with one or more parameters, the first of which is integer or float passed by value

A signature that uses these symbols is called *underspecified* since it might not map to a single host language type. Unification involving underspecified signatures proceeds exactly as described in Section II-E for fully specified signatures: unification succeeds if the set of types denoted by the import signature is a subset of the set denoted by the export signature.

The definition of a type language in which a parameter's signature does not map directly into a single host language data type allows for more general-purpose routines, but it also raises questions about implementation. Specifically, it causes a problem since it may be impossible to supply a single host language parameter that can accept all possible argument values. For example, what host language type should be used if the specification is "?", indicating that any type can be accepted? This problem is solved by delaying the decoding of the UTS value and placing that process under programmer control. Specifically, a *representative* for the UTS value is supplied as the argument value. Then, routines in the MLP library can be invoked with a representative as argument to allow the programmer to inquire as to the UTS type of the corresponding value and to perform explicit conversion of values from the UTS language format into the format of the host language. Thus, the representative of a value can be viewed as a ticket that is presented to an agent to gain access to the value. The value itself is maintained by the agent, which also assigns the representative.

The use of a representative can be viewed as a natural consequence of constructing a language binding. As noted in the previous section, a language binding specifies how each UTS type translates into the types supplied by the host language. However, to be precise, the language binding must be constructed so that a mapping is given for each possible type *expression* (i.e., signature) derivable from the grammar, not simply each individual type. That is, it must be determined how, for example, "integer or float" maps into the host language and not just how "integer" and "float" translate individually. Of course, some of these expressions cannot be translated directly in a meaningful way; in such cases, the mapping is to a host language type, usually integer, that is used to hold the representative for that value instead of the value itself.

Note, however, that operations that are valid for the host language type to which a representative translates should not be applied to the representative itself ("+" makes no sense, for example). To reinforce this view, the variable to contain the representative is declared in the program using a UTS declaration, not a host language declaration. In particular, such a variable *r* is declared by using the keyword **representative**. The MLP translator then translates this into a host language variable declaration, the type of which is determined by the language binding. In addition to emphasizing that *r* is not to be manipulated by the programmer, such a declaration also allows the actual host language type utilized to be determined by the MLP system implementor.

B. Utility Routines

To motivate the functionality that the conversion and inquiry procedures must supply, consider the two reasons why the UTS specification for a particular parameter may not map into a host language data type: the parameter may be underspecified or it may be a constructed type not supported by the host language. The former would occur, for example, if a parameter is specified as "integer or float," while an example of the latter is a Fortran procedure that has a parameter specified as UTS type **record**{. . .}.

The routine uts_inspect is intended to resolve the first problem by supplying a means for testing the UTS type of a datum. Invoked with a representative *r* and a signature *sig*, uts_inspect returns **true** if the value whose representative is *r* is of a UTS type that unifies with *sig* and **false** otherwise. For example, to determine the type of an argument supplied to a parameter whose signature is "integer or float," one could use uts_inspect in a Fortran 77 program as follows.

```
      SUBROUTINE example2(r)
        REPRESENTATIVE r
        EXPORT "example2" prog(val "r" integer or float)
        INTEGER rint
        REAL rfloat
C
C
        . . .
        IF(uts_inspect(r,"integer"))
          THEN
            fetch integer value into rint
            continue computation, using integer value of r in rint
        ELSE IF(uts_inspect(r,"float"))
          THEN
            fetch float value into rfloat
            continue computation, using real value of r in rfloat
```

Having determined the type of an argument, some way to covert it into a host language value is needed. This is provided by the routine uts_decode, which takes a representative *r* and a host language variable *x*, and converts the UTS value represented by *r* into host language format and assigns it to *x*. So, for example, obtaining the value of the argument in the above program after having determined it was an integer could be done by

```
CALL uts_decode(r,rint)
```

Note that uts_decode is only defined for UTS types that map directly to host language types.

The second problem—that of accepting an argument of a constructed type not found in the host language—is solved by procedures uts_decompose and uts_length. The first routine, when invoked with a representative *r* and an integer selector *s*, returns a representative for the *s*th element of the UTS value represented by *r*. The second routine, when invoked with a representative *r*, returns the number of elements in the UTS value represented by *r*. uts_decompose is defined only for composite types constructed by **array** and **record**, while uts_length is defined for arrays, records, strings and bytes. Nested aggregates such as **record**{**record**{. . .}, . . .} can be decomposed using repeated calls to uts_decompose. Also, note that since the result of uts_decompose is a representative, the returned value cannot be used until it is decoded into a host language value.

The above description focuses on the problem of decomposing a composite argument into constituent parts so that it can be accepted into a procedure. The symmetric problem can arise at the point where the arguments originate, that is, in the procedure containing the cross-language invocation. For example, a Fortan program might invoke a Pascal procedure that has a parameter of type record. To solve this kind of problem, three routines for constructing UTS values are also provided. The first—uts_encode—is analogous to uts_decode, while the others—uts_create and uts_compose—are used to create and initialize constructed types.

Two additional library routines are provided that combine the encoding/decoding and composition/decomposi-

uts_inspect(r, signature)
: returns true if the type of the UTS value represented by unifies with *signature*; false otherwise.

uts_length(r)
: returns the number of fields of the UTS value represented by *r*. The type of the value represented by *r* must either be **array**, **record**, or string.

uts_encode(r, signature, var)
: converts the value of the variable *var* into the UTS value with signature *signature*, and assigns *r* as its representative.

uts_decode(r, var)
: converts the UTS value represented by *r* into *var*. The value is converted into the format of *var's* host language.

uts_create(r, signature)
: creates a UTS value whose signature is *signature*, initializes its fields to null, and assigns *r* as its representative.

uts_compose(r, field, s)
: assigns the UTS value represented by *s* to field number *field* of the UTS value represented by *r*. The UTS type of *r* must be either **array** or **record**.

uts_decompose(r, s, field)
: assigns *r* as the representative to field number *field* of the UTS value represented by *s*. The UTS type of *s* must be either **record** or **array**.

uts_insert(r, field, signature, var)
: converts the value of host language variable *var* to a UTS value with signature *signature*, and assigns it to field number *field* of the UTS value represented by *r*.

uts_extract(r, field, var)
: converts field number *field* of the UTS value represented by *r* into host language format and assigns it *var*.

Fig. 6. Inquiry and conversion routines from the MLP library.

tion aspects of value translation. The first, uts_insert, has the effect of a uts_encode followed by a uts_compose. This routine simplifies the task of, for example, piecing together an argument value of a constructed type prior to invoking an imported routine. Similarly, the routine uts_extract is equivalent to a uts_decompose followed by a uts_decode; it is used primarily for accepting the value from a single field of an argument into a program. The functionality supplied by these routines and the others in the MLP library is summarized in Fig. 6.

The flexible nature of the MLP library routines facilitates the programming of general-purpose utilities. For example, they can be used to create a print routine in C that accepts one value for any type and writes a printable representation on the standard output. This routine uses uts_inspect to test the argument type, then uses uts_decode to convert the argument into C format from which a printable representation can be output using the standard C routine printf. The code for this generic print routine appears in Fig. 7.

V. Related Work

Much of the previous work in mixed language programs has been motivated by the needs of numerical programmers desiring a flexible host language, yet anxious not to waste the large investment in Fortran-based routines. As a result, current facilities tend, in general, to be applicable within a narrow scope; typically, a particular language implementation supports cross-language calls to a small number of other languages, a specific collection of library routines is invokable from a small number of languages, or languages adhering to a machine-specific

```
/*  print — print the contents of one UTS value  */
print(r)
export "print" prog (val "r" ?);
representative r;
{
        representative q;
        int rint, len, i;  double rdouble;  char ch[256];
        /*  Determine the type of the UTS value and take appropriate action  */
        if (uts_inspect (r, "integer")) {
                printf("Integer: ");
                uts_decode(r, &rint);
                printf("d\n", rint);
        } else if (uts_inspect (r, "float")) {
                printf("Float:  ");
                uts_decode(r, &rdouble);
                printf("f\n", rdouble);
        } else if (uts_inspect (r, "bool")) {
                printf("Bool:  ");
                uts_decode (r, &rint);
                if (rint)  printf("True\n");  else  printf("False\n");
        } else if (uts_inspect (r, "string[?]") || uts_inspect(r, "byte[?]")) {
                printf("String/Byte:  ");
                len = uts_length(r);
                uts_decode(r, ch);
                for (i = 0; i < len; i++)  putc (ch[i], stderr);
                putc('\n', stderr);
        } else if (uts_inspect (r, "record[*]")) {
                printf("Record:\n");
                len = uts_length (r);
                for (i = 1; i <= len; i++) {
                        uts_decompose (&q, r, i);
                        print (q);
                }
        } else if (uts_inspect (r, "array[*] of ?")) {
                printf("Array:\n");
                len = uts_length (r);
                for (i = 1; i <= len; i ++) {
                        uts_decompose (&q, r, i);
                        print (q);
                }
        } else if (uts_inspect (r, "null"))
                printf("Null\n");
          else if (uts_inspect (r, "prog(*) returns (?)")) {
                uts_decode(r, ch);
                printf("Procedure or Function: %s\n", ch);
        } else  printf("UTS signature\n");
}
```

Fig. 7. Generic print routine written in C.

calling convention can be combined. For example, in UNIX, it is straightforward to invoke C procedures from Fortran and vice versa [8], while the IBM P1/1 compilers contain provisions for invoking routines written in Fortran or Cobol [12]. Most operating systems provide analogous-facilities.

Due to the ad hoc nature of many of the provisions for interlanguage programming, the amount of work required of a programmer is often considerable. This and other deficiencies have prompted several attempts at improving the situation. For example, the designers of Ada® anticipated this problem and included in their specifications a facility to invoke procedures in other languages [6]. This is accomplished by including in the calling program a pragma of the form

pragma INTERFACE(language,routine_name)

The type of the parameters to be passed to the named routine and the type of the return value are specified in a regular function declaration. The actual implementation of such a cross-language call, however, remains unspecified; it is left to the designer of each particular Ada implementation to decide how this goal is to be achieved. Moreover, it is even possible for versions of Ada to omit the feature altogether since it is not one of the items required by the Ada standard. A more general solution along these same lines has also been proposed in [7].

A slightly different approach has been proposed by Darondeau *et al.* in [4]. Basically, their approach consists of defining a small set of universal types (e.g., stdreal, stdint) in which all interlanguage communication can be conducted. These universal types are added to each communicating language by modifying the compilers. Since such a collection of basic types is not rich enough to provide flexibility in all situations, the concept of a *foreign type* has also been defined. A foreign type is a type that is imported into one routine from another routine written (potentially) in a different language. A foreign type may only be used to specify parameters to be passed in procedure invocations. Since the importing routine cannot declare or manipulate variables of the foreign type, such a concept is best suited for abstract data types, where an initial call to the implementing routines returns a value to be passed as a parameter in subsequent invocations.

Related work involving distributed systems has concentrated on the implementation of remote procedure calls within specific languages [3], [11], [15], [21]. Since these systems typically define a canonical data representation and a translation mechanism that performs a function similar to our agents, they could be used as a basis for developing a scheme comparable to ours. However, modifications would still be necessary since these systems tend to be tied closely to a single language rather than being language-independent in the manner of the MLP system.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

A more closely related remote procedure call mechanism is that recently proposed for Berkeley UNIX by Sun Microsystems, Inc. [19]. Like the MLP system, their scheme facilitates distributed mixed language programs by supplying a machine independent type system (called XDR) and a generic remote procedure call facility. However, there are several significant differences:

1) The MLP system has the ability to do runtime type discrimination; this allows underspecification of parameter types, runtime type checking, and separate compilation of program components without having to rely on a shared database of interface specifications.

2) The UTS data description language is more complete; for example, there is no notion analogous to the UTS type prog in the Sun scheme.

3) MLP places less burden on the user; for example, in the Sun system, the user must code the procedure as a process and explicitly initiate the process before the call.

It seems clear from the proposal that, unlike MLP, their facility is intended primarily for implementing system services rather than for allowing users to construct distributed mixed language programs.

VI. Discussion

Our philosophy in developing the MLP system was to design a system that would facilitate the development of most kinds of distributed mixed language programs. However, it has never been our intent to handle all possible situations that would arise in multilingual programs and so our approach has certain limitations. One class of limitations stem from the fact that the UTS language is not complete. Because of this, it cannot be used to represent an arbitrary data structure nor describe an arbitrary program interface. It does not allow the creation of types with a semantic component nor the creation of abstract data types. To do so would require the maintenance of a central database of type definitions and associated routines, substantially increasing the complexity of the system with only a marginal improvement in utility.

A second aspect of MLP where this philosophy surfaces is in our decision to use separate address spaces and interprocess communication to implement a cross-language invocation. This approach was dictated primarily by our desire to allow programs to be distributed, but it also allows significant simplification in the implementation of mixed-language programs even on a single processor. It does result in a certain loss of flexibility, however. For example, this means the only value/result parameter-passing semantics are supported and that references to variables in another address space either directly or through pointers are impossible. Again, though, it is our opinion that the use of multiple processes is of sufficient generality to support the vast majority of situations in which such multilingual programs would be required.

It should also be noted that many of the efficiency concerns about the use of multiple processes in this way have been investigated in the context of remote procedure calls

[15], [17]. The conclusion of these studies with respect to efficiency is that this interprocess communications primitive can be implemented efficiently enough for practical use in a local area network. This argues that programs written following our approach will also be of acceptable execution efficiency, even though each cross-language invocation will incur more overhead than a normal procedure call.

VII. Conclusions

In this paper, we have described an approach for constructing distributed mixed language programs. This approach, embodied in the MLP system, uses a separate process for each program component and employs the UTS data description language to specify system interfaces and type data. Each cross-language invocation then becomes a matter of translating the arguments into UTS format, invoking the desired procedure by transmitting the arguments to the appropriate process, and then translating the information into the types of the language in which the procedure is implemented. Straightforward data translations can be handled with little user intervention, while provisions for more complicated situations are supplied in the form of library routines that enable explicit programmer control of conversions.

A prototype implementation of the MLP system is nearing completion [13]. This prototype currently allows construction of mixed-languages programs using the programming languages C, Pascal, and Icon, with the addition of other languages being planned. The system runs on a network of Vaxes and Sun workstations executing Berkeley UNIX.

Appendix
Grammar for UTS Types and Type Expressions

In the following grammar, nonterminals are surrounded by angle brackets (< >), and optional items are surrounded by brackets ([ ]); literal instances of these characters are surrounded by quotes (' ').

Most of the types should be self-evident. The base type "byte" is intended to allow programs to accept uninterpreted sequences of bits, while an instance of the "signature" type is a data value that represents a type expression; this type allows a program to receive, for example, the signature associated with a procedure as an argument. The error type is intended to allow error or out-of-channel values in the same spirit as the "Not-a-Number" values in the IEEE floating point standard [18].

The special symbols used in type expressions are interpreted as follows. "?" is the set of all types, while "*" is used to denote an arbitrary number of arguments of any type, or an arbitrary number of arbitrary size dimensions. The length of a string or the size of a dimension may be specified as a single number, or it may be specified using "–" to allow a range of lengths. Omitting the number at either end of the range indicates that the length is unbounded above or bounded below by 0. Thus, **array**(*) specifies neither number nor size of dimensions, while **array**(–,–,–) specifies 3 arbitrarily-sized dimensions. string[10–20] specifies a string of allowed length between 10 and 20 (inclusive).

Acknowledgment

L. Henderson provided assistance in locating the Eispack routine *cg* and the IEEE Floating Point Standard. Also, D. Hanson and especially G. Andrews made many helpful comments on earlier drafts of this paper. S. Manweiler deserves special thanks for his part in constructing the prototype implementation of the MLP system, and for providing valuable feedback on the ideas elaborated in this paper and on UTS in general.

References

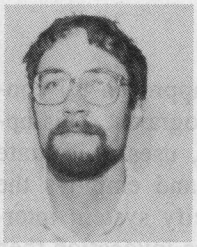
[1] G. R. Andrews, "The distributed programming language SR—Mechanisms, design, and implementation," *Software—Practice and Experience*, vol. 12, pp. 719–754, Aug. 1982.

[2] Amer. Nat. Standards Inst., "Draft ANSI Standard X3H3/83-25r3: Graphical kernel system," *ACM SIGGRAPH*, Feb. 1984.

[3] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, Feb. 1984.

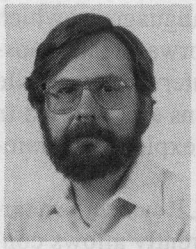
<typeexpr> ::= <type> | <type> or <typeexpr>
<type> ::= <single> | <composite> | (<typeexpr>) | ?
<single> ::= <basetype> | signature | error | null
<composite> ::= <record> | <array> | <prog>
<basetype> ::= integer | float | string '[' <length> ']' | byte '[' <length> ']' | bool
<record> ::= **record** { <typeexprs> }
<typeexprs> ::= <typeexpr> | <typeexpr>, <typeexprs>
<array> ::= **array** '[' <dims> ']' **of** <type>
<dims> ::= <length> | <length>, <dims> | *
<length> ::= <no> | [<no>] – [<no>]
<prog> ::= "name" **prog**(<args>) [**returns** (<typeexpr>)]
<args> ::= <arg> | <arg> , <args> | *
<arg> ::= [<direction>] ["name"] <type>
<direction> ::= **val** | **res** | **var**
<no> ::= <digit> | <digit><no>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

[4] P. Darondeau, P. le Guernic, and M. Raynal, "Types in a mixed language system," *BIT*, vol. 21, pp. 246–254, 1981.
[5] A. Demers and J. Donahue, "Data types, parameters, and type checking," in *Proc. Seventh Ann. Symp. Principles of Programming Languages*, Las Vegas, NV, 1980, pp. 12–23.
[6] *Reference Manual for the Ada Programming Language*, U.S. Dep. Defense, July 1982.
[7] B. Einarsson and W. Gentleman, "Mixed language programming," *Software—Practice and Experience*, vol. 14, pp. 383–395, Apr. 1984.
[8] S. Feldman and P. Weinberger, "A portable Fortran 77 compiler," in *UNIX Programmer's Manual*, vol. 2B, Bell Telephone Laboratories, Murray Hill, NJ; 1979.
[9] J. Foderaro, K. Sklower, and K. Layer, "The FRANZ LISP manual," in *UNIX Programmer's Manual*, vol. 2C, Comput. Sci. Division, Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, CA, 1983.
[10] R. Griswold and M. Griswold, *The Icon Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[11] M. Herlihy and B. Liskov, "A value transmission method for abstract data types," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 527–551, Oct. 1982.
[12] *OS360 PL/I Checkout and Optimizing Compilers: Language Reference Manual* (publication GC33-0009-4), IBM Corp., San Jose, CA 1976.
[13] S. Manweiler, R. Hayes, and R. Schlichting, "The MLP System user's guide," Dep. Comput. Sci., Univ. Arizona, Tucson, Tech. Rep. 86-4, Feb. 1986.
[14] *Macsyma Reference Manual (Version 10)*, Mathlab Group, Lab. Comput. Sci., Massachusetts Inst. of Technol., Cambridge, MA: 1983.
[15] B. J. Nelson, "Remote procedure call," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Rep. CMU-CS-81-119, May 1981.
[16] B. Smith *et al*, *Matrix Eigensystem Routines—EISPACK GUIDE*. New York: Springer-Verlag, 1976.
[17] A. Spector, "Performing remote operations efficiently on a local computer network," *Commun. ACM*, vol. 25, pp. 246–260, Apr. 1982.
[18] D. Stevenson, *A Proposed Standard for Binary Floating-Point Arithmetic, Draft 10.0*, IEEE Floating-Point Subcommittee Working Document P754/82-8.6, 1982.
[19] *Remote Procedure Call Reference Manual*, Sun Microsystems, Inc., Part Number 800-1177-01, Nov. 1984.
[20] N. Wirth, "Modula: A language for modular multiprogramming," *Software—Practice and Experience*, vol. 7, pp. 3–35, Jan.–Feb. 1977.
[21] *Courier: The Remote Procedure Call Protocol*, Xerox Corp., Xerox System Integration Standard XSIS 038112, Stamford, CT, Dec. 1981.



Roger Hayes received the B.S. degree in computer science from Portland State University, Portland, OR, and the Master's degree in computer science from the University of Arizona, Tucson.

He is currently working toward the Doctorate degree at the University of Arizona. His research interests include distributed systems, user interface paradigms, and graphics. His dissertation research, done under the auspices of the Saguaro Distributed Operating System Project, concerns the communication of data across subsystem boundaries.



Richard D. Schlichting (S'81–M'82) received the B.A. degree in mathematics from the College of William and Mary, Williamsburg, VA, in 1977, and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1979 and 1982, respectively.

He is currently an Assistant Professor in the Department of Computer Science at the University of Arizona in Tucson. His research interests include distributed systems, fault-tolerant computing, and programming logics.

Dr. Schlichting is a member of the Association for Computing Machinery.