Managing data-types : the CORBA Approach and AutoMap/AutoLink, an MPI Solution

Martial MICHEL , André SCHAFF , Judith Ellen DEVANEY

Abstract— **There are many ways to create a distributed system, such as with Parallel Virtual Machine (PVM), PARMACS, p4, Message Passing Interface (MPI) and the Common Object Request Broker Architecture (CORBA).**

This article concentrates on MPI, CORBA and the interface for information (data-type) transfer. We discuss the transfer of complex data-types, that are compositions of basic predefined data-types, and also present methods for transferring dynamic data-types, which are data-types that link to other basic, complex or dynamic data-types (i.e. contain pointers).

The article will present CORBA, MPI, their default way of handling such data-types, and our MPI solutions : AutoMap and AutoLink.

Keywords— **CORBA, MPI, data-types, AutoMap, AutoLink.**

I. Introduction

THE Common Object Request Broker Architecture (CORBA) was introduced in 1991, and it is in wide use; the Message Passing Interface (MPI) is more recent (version 1.1 in 1995) and its use is primarily in High Performance Computing (HPC). Both aim at supporting distributed computing. In distributed computing, two issues need addressing : the model for communication, and the interface presented to the user for transmitting information. Commonly, the information to be transfered consists of collections of pre-defined data-types contained in user defined structures.

In CORBA, the communication model is client/server. The client and server are usually distinct processes, but they may be contained in a single process. In MPI, the communication model is at the discretion of the user and thus must be constructed by the user.

This paper is concerned with the second issue : the interface available to the user for information transfer, specifically the transfer of varieties of data types in CORBA and MPI. The motivation for this is to see if the algorithm developed for MPI can easily be transfered to the CORBA environment. We address two items :
1. the transfer of multi element or complex data-types. These correspond to ordinary C structs.
2. the transfer of groups of complex types of (1) where the individual elements are connected by pointers. These correspond to dynamic data-types such as trees, graphs, . . . , with the additional requirement of support for heterogeneous nodes in the dynamic data-types case.

The rest of the paper is organized as follow. In section II, we present CORBA and show how the Common Object Request Broker Architecture handles composed data-types. Then, in section III, we show how MPI allows the creation of complex data-types (composition of basic data-types), in a simple but repetitive process. In section IV, we present AutoMap, a tool designed to create complex MPI data-type (composition of data-types) directly from the user's code. In section V, we introduce AutoLink, an MPI based library, which given the root node of a dynamic data-type (like a tree or a graph) will flatten it, transfer it via MPI, and reconstruct it and its links on the receiving MPI process. The conclusion to this article is section VI.

II. CORBA

CORBA is a conceptual "software bus" that makes communication possible between applications, regardless of computer language, and platform. CORBA is designed to work transparently with objects; an object is a data-type plus a behavior, i.e. procedures internal to the type. Parts of the data may be private; i.e. opaque to the user.

CORBA specifies (as can be seen in [1]) the components :
• *Object Request Broker*, which in a distributed environment, makes it possible to work with objects.
• *Object Services*, which provide basic functions for using and implementing objects.
• *Common Facilities*, which provide a collection of services that many applications may share.
• *Applications Objects*, which control object interfaces.

In this article, we will to concentrate on the way CORBA handles data-types.

A. Objects in CORBA

CORBA uses *concrete* object models, which are models that work with data and methods to be applied to it.

The Object Management Architecture (OMA) specifies that a CORBA object adheres to a classical object model[1].

An object possess a public interface that is defined through the CORBA Interface Definition Language (IDL). This interface hides the concrete object implementation.

To work with the object, the Object Request Broker (ORB) component implements the CORBA "bus", and manages the client requests while hiding location and implementation details[2] from the client.

[1]all methods are contained within a class
[2]an Object Managment Group (OMG) standard language mapping

Martial MICHEL is the AutoMap/AutoLink Project Contact, a Guest Researcher at NIST, and a PhD student in the RÉSÉDAS group in France
André SCHAFF is the RÉSÉDAS scientific leader
Judith Ellen DEVANEY is the Scientific Applications Support Project Leader at NIST, as well as the AutoMap/AutoLink Project Leader

Thanks are here given to Edward J. BARKMEYER and Kevin G. BRADY for their help on the CORBA part

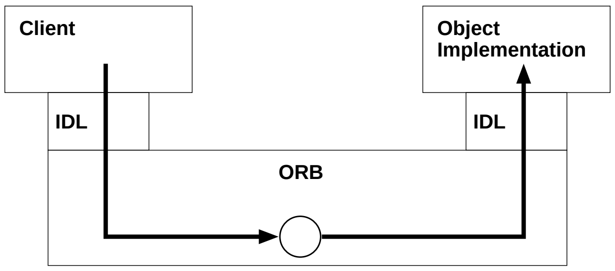

Fig. 1. CORBA request example

Figure 1 is a simple example of how a request from the client applies data to an object; the request goes through the ORB, and is interpreted for the requesting applications through the IDL, so that the client sees only the object interface; not the details of the implementation. The client and the object requested may not be in the same process but available through network connection; however, the concept of requesting an object remains the same.

CORBA gives the user two modes of object invocation : *static* and *dynamic*. In the first mode, IDL object definition is compiled in the target program. In the second mode, everything is handled at runtime, by the client using ORB functions. This last mode gives more flexibility for the use of the objects, but it requires more user coding.

The CORBA object model works to some extent with the concept of *object references*, which defines an object instance (locally or remotely)

B. Data-types in CORBA

CORBA recognizes types of two kind : object and non object. A non object is a basic or constructed type, as shown in figure 2 (on page 2).

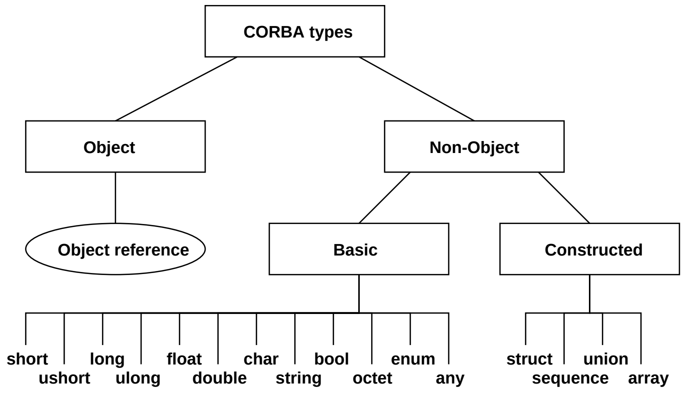

Fig. 2. CORBA types

As can be seen in figure 2, among the basic data-types CORBA recognizes, is an *any* type. The *any* type can represent any possible basic or constructed type, where constructed types are complex data-types made out of basic data-types.

is defined for some programming language, so that users can have access to OMG IDL types and methods

An *any* value is a type definition that can be passed to a program without any static information for the type, so the object does not have a portable method of using it. An *any* value can be dynamically interpreted and constructed using *DynAny* objects.

Using its IDL, CORBA can work with complex and dynamic data-types. Complex data-types are easily handled by CORBA for they are easily created through IDL (an example of such a "composed" data-type for IDL can be found in entry 1 (on page 2)); in contrast with MPI (cf Section III-A.1) it is not necessary to give details of memory offsets and displacements to create a CORBA data-type. Dynamic data-types on the other hand, are not handled by default by CORBA, and are not easily created. Although it is syntactically possible to generate recursive type specifications in IDL, such recursion is semantically constrained; the only permissible form of recursive type specification is through the use of the *sequence* template type.

Entry 1 IDL entry example

```
struct FormattedData {
  string          representation;
  sequence<octet> value;
};
```

In CORBA, in a sense, the object reference replaces the pointer; but only some vendors have supersets of IDL that include the notion of pointer. Still, a pointer does not reference across process bounds. Pointers from one process to another lose their meaning : to avoid that, either what it references must travel with it, or an adjustment is needed relative to its target on arrival.

Two solutions —at least— are possible for handling such dynamic data-type transmission with CORBA :

1. Based on the *object concept*, it is possible to give each object a behavior so that it knows how to traverse and recreate itself if asked through the ORB. This method requires some coding from the user, but insures proper —and efficient, depending on the coding— traversal of such data-types. It may require the object to contain methods that need to be given to the other objects. An example, consider the transfer of a dynamic data-type such as a tree (or a graph); the user would have to define a node indexing process, as well as a tree (graph) reconstruction process, in order to send and receive the object content a proper way.
2. In IDL, there is also a transfer mode called "objects by value". This uses a *get* and *put* to copy the data as they are. Pre- and post- processing may need to be done by the user. In the case of pointers, new valid references would need to be created.

To have a higher level of understanding in the process of transmission of CORBA, one has to understand that through some interoperability protocols, basic data-types are sent as byte streams using the *Common Data Representation*, as explained here after.

C. Interoperability

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs.

Two *Inter-ORB Protocols* exist (the second a complement of the first) :
1. The *General Inter-ORB Protocol* (GIOP) element specifies a standard transfer syntax and a set of message formats for communications between ORBs.
2. The *Internet Inter-ORB Protocol* (IIOP) element specifies how GIOP messages are exchanged using TCP/IP connections.

C.1 GIOP

The GIOP possesses the following elements :
- The *GIOP Message Formats*, which specify the contents of messages regarding object (request, implementation, ...).
- The *GIOP Transport Assumptions*, which describe general assumptions concerning any network transport layer that may be used to transfer GIOP messages.
- The *Common Data Representation* (CDR), which is a transfer syntax, mapping OMG IDL data-types so that they can be transfered between ORBs. This particular element is described more in detail in section II-D.

C.2 IIOP

The IIOP specification adds *Internet IOP Message Transport* to the GIOP specification; it describes how agents open TCP/IP connections and use them to transfer GIOP messages.

D. Common Data Representation (CDR)

The *Common Data Representation* (CDR) is a transfer syntax[3] (a data-type encapsulation), that has the following features :
Variable byte ordering Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.
Aligned primitive types Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.
Complete OMG IDL Mapping CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as *TypeCodes*. Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

[3]the format in which the GIOP represents OMG IDL data types in an octet stream

Encapsulations are octet streams into which OMG IDL data structures may be *marshaled* independently, apart from any particular message context. Once a data structure has been encapsulated, the octet stream can be represented as the OMG IDL opaque data type `sequence<octet>`, which can subsequently marshaled into a message or another encapsulation.

D.1 Basic data-types

Each data-type has a special *alignment* in order to enable primitive data to be moved into and out of octet streams, all primitive data types must be aligned on their natural boundaries.

D.2 Complex data-types

Also called "OMG IDL Constructed Types" for they are built from OMG IDL's data types using facilities defined by the OMG IDL language.

Constructed type have no alignment restrictions beyond those of their primitive components; the alignment of those primitive types is not intended to support use of marshaling buffers as equivalent to the implementation of constructed data-types within any particular language environment. GIOP assumes that agents will usually construct structured data types by copying primitive data between the marshaled buffer and the appropriate in-memory data-structure layout for the language mapping implementation involved.

E. Discussion

It is straightforward to transfer complex data types with CORBA using IDL to define the structures. Transferring structures with pointers requires additional work on the part of the coder. There needs to be either pre- and post-processing; or object encapsulation with behaviors that provide node indexing and traversal, for example.

III. MPI

MPI is a distributed communication library, working on the "Message Passing" concept. Like CORBA it knows a predefined set of basic data-types. It allows for the creation of new data-types. Since MPI is a library, we will discuss MPI functions using its bindings for the C language.

A. Data-Types

The MPI library can only transfer types that it knows about, and permits the creation of user defined MPI data-types (a complex and repetitive operation).

A.1 Complex Data-Type

Users may want to use a composition of basic data-types, which we will call a "complex data-type"(as long as the user doesn't use pointers to other structures or components inside the structure).

So, an example of a complex data-structure may be as in entry 2 (on page 4)

Entry 2 Complex data-type example

```
struct {
  char      display[50];
  int       maxiter;
  double    xmin, ymin;
  double    xmax, ymax;
  int       width;
  int       height;
} cmdline;
```

A.2 Dynamic Data-Type

Dynamic data-types are an extension of complex data-types, for they can handle structures containing pointer fields. Examples of such data-structures [2] are *linked lists*, *trees*, and *graphs*.

Like CORBA, there are at least two ways to transfer dynamic data-types. Similar to the CORBA "object by value", one can send data-types, such as trees, as is with MPI; but the pointer memory references will be invalid on the receiving processor. So creating valid dynamic data-types may require the user to pre- and post- process the data-types.

Alternatively, one can encapsulate the pre- and post-processing in the MPI library; this is the approach followed in the AutoLink design, detailed in Section V.

In the following, we describe the creation of a new data-type from a complex data-type. Then, we discuss AutoMap which automates this process.

B. Complex Data-Type Creation with MPI

B.1 C Structure

We will create the data-type defined in entry 2 (on page 4), which contains 50 chars, 3 integers (1 and then 2 more), and 4 doubles.

B.2 Creation of the MPI Data-Type

The process of creating an MPI data-type involves specifying the layout in memory of the data in the C structure [3]. It is done in six operations :

1. Set up an array defining the number of data of each kind that will be used (in the same order as the structure definition).

`int blockcounts[4]= {50,1,4,2};`

which corresponds to : 50 `char`, 1 `int`, 4 `double`, 2 `int`.

2. Set up an array that will contain the type specification for each element contained in the structure. There are four fields[4] in the `struct`, thus :

`MPI_Datatype types[4];`

Set the data-type for each element of the data-type to be created :

```
types[0]= MPI_CHAR;
types[1]= MPI_INT;
types[2]= MPI_DOUBLE;
types[3]= MPI_INT;
```

[4]even if there are only 3 different data-types, the `struct` type order needs to be followed so that a correct mapping may be done

3. Set up an internal displacement array containing the memory offset of each field in the `struct` :

`MPI_Aint displs[4];`

Map onto the displacement array, the MPI data-type on the C structure (by linking it to the very first memory element) :

```
MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin, &displs[2]);
MPI_Address(&cmdline.width, &displs[3]);
```

Adjust the displacement array so that the displacements are offsets from the beginning of the structure :

`for (i = 3; i >= 0; i - -) displs[i]-= displs[0];`

4. Give a name to the MPI data-type :

`MPI_Datatype cmdtype;`

5. Build the new MPI type :

```
MPI_Type_struct(4, blockcounts, displs, types,
&cmdtype);
```

6. Validate the type existence to be used with MPI :

`MPI_Type_commit( &cmdtype );`

IV. AutoMap

AutoMap is a tool designed to simplify the MPI user's task when willing to create complex data-types; the tool will, for the user's C type definition file extract used data-types and create output files containing C procedures to define the MPI user types.

A. Overview of AutoMap use

AutoMap is very simple to use, and in version 2.1 it comes with the optional building of AutoLink output files (see Section V on page 5).

To use it the process is simple :

1. Edit your type definition file, by adding 3 markers :

- `/*~ AM_Begin */` to mark the beginning of the AutoMap recognition process.
- `/*~ AM_End */` to mark the end of the recognition process.
- `/*~ AM */` to specify which types are to be processed.

In the entry 2 (on page 4), it will look like entry 3 (on page 5).

2. Run AutoMap on the type definition processed file (the options are explained in entry 4 (on page 5), so in the present case it was run —at least— with the option `-noAL`); it will output files such as :

- `mpitypes.inc` which defines a file to be included by the user code (after the user type definition file) fully defining a `Build_MPI_Types()` command, that does all of the MPI type creation process (as described in Section III-B).
- `mpitypes.h` is the prototype definition for the `mpitypes.inc` file.
- `logbook.txt` is an internal AutoMap log [5].

3. Include the generated files and other prototypes files, as shown in figure 3 (on page 5, in the following order :

- `mpi.h` because MPI is required.

[5]used mainly for debugging purposes, and/or user understanding of the low level engine

• `struct.h` so that the user data-types are known to C, and we can do some mapping to MPI.
• `mpitypes.inc` so that the created `Build_MPI_Types()` command definition is available for use.
4. In the user code, after having done the required `MPI_Init`, just execute the `Build_MPI_Types()` procedure so that the user AutoMap created data-types are available. One can the then use `AM_` prefixed user data-types with MPI [6].

Entry 3 AutoMap adapted code

```
/*~ AM_Begin */

struct {
  char     display[50];
  int      maxiter;
  double   xmin, ymin;
  double   xmax, ymax;
  int      width;
  int      height;
} cmdline /*~ AM */;

/*~ AM_End */
```

Entry 4 AutoMap options

```
AutoMap [-help | [-v] [-noAL] [-log] filename]
        -help : Will print this help menu
        -v    : Verbose mode
        -noAL : Will not generate the entries
                for use with AutoLink
        -log  : Will generate the "logbook.txt"
                for this run
        filename : name of the C typedef
                   definition file to analyze
```

B. *Design of AutoMap*

The design of a "source-to-source" compiler tool is with grammars that match C type definitions, and a core engine to process the recognized types. The tool Yacc++[4] is used as the parser and lexer for the grammars.

Two grammars are used in the creation process :
• The first grammar is used to recognize the AutoMap begin marker and then run the second grammar processing.
• The second grammar reads all the user created data-types' definitions and make it possible for the core engine to access them. The second grammar stops its processing when recognizing the AutoMap end marker.

The core engine works with the abstract simplified design algorithm described in entry 5 (on page 6), that makes sure that if a "sub-type" is used by a data-type, this sub-type definition is created. To do so, it has to work with an Abstract Syntax Tree (AST) generated from the user data-type definition.

[6]in our example, the data-type was `cmdline`; as the AutoMap data-type is `AM_cmdline`

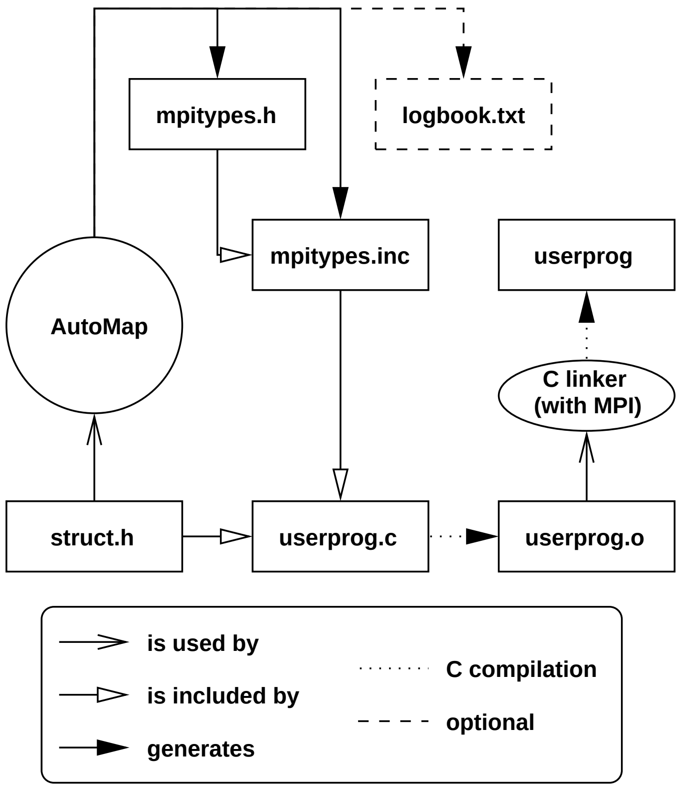

Fig. 3. AutoMap use process

C. *Availability*

AutoMap is available at `http://www.nist.gov/itl/div895/auto/`. It may be downloaded or run directly on its web page .

V. AutoLink

AutoLink is a tool for transmitting dynamic data-types (graphs, trees, ...) with MPI.

A. *Overview of AutoLink use*

To use AutoLink :
• Run AutoMap on a definition file containing the same tag as defined in Section IV-A. It will create the files :
– `mpitypes.inc` and `mpitypes.h` define AutoMap user's data-type generation process.
– `autolink.h` defines the prototype definition for AutoLink functions, as well as the naming of the user defined types for those functions (same concept as for AutoMap, but the `AM_` is replaced by `AL_`).
– `al_routines.inc` defines the specialized code for each user function on user data-types; it is only supposed to be used by `autolink.inc`, and defines the "@" entries in the send and receive algorithm of AutoLink shown in entry 6 (on page 7) and entry 7 (on page 7).
– `logbook.txt` is the internal AutoMap log.
• Include files as presented in figure 4 (on page 6), where :
– `al_common.h` contains all the common AutoLink definitions and debug mode settings.

Entry 5 AutoMap abstract simplified core engine algorithm

```
|Creation of the AST
|   |Processing Grammar1
|   |Processing Grammar2
|Updating the AST
|   |Copying Pointer information and
|   |  Assigning subTypes (number)
|   |Assigning Datatype Numbers
|Writing MPI data-types definitions files
|  (and the Build\_MPI\_Types procedure)
|   |First, the sub-types
|   |Then, the user defined types
|If AutoLink generation asked
|Then |Writing AutoLink internal
|     |  routines definitions
|     |Writing AutoLink types definition
|If run in verbose mode
|Then |Print run information
|Cleaning the AST
```

– `al_internals.h` defines the prototypes of AutoLink internal data-types (detailed in Section V-C).
– `al_internals.c` contains the definition of the AutoLink internal data-types; note that they define some fully operational specialized types and functions.
– `autolink.inc` is the AutoLink main code definition itself, it is the front end to the specialized functions generated by AutoMap for each data-types required.
• In the user code, after the required `MPI_Init`, execute the `AL_Init` procedure (it will automatically generate the MPI data-types, and set up the AutoLink engine). Now the AutoLink functions (detailed in Section V-B) are available for use. Before doing the `MPI_Finalize` command, do a `AL_Finalize` command to tell AutoLink to finish its processing.

B. AutoLink functions

B.1 `AL_Init()`

It initializes AutoLink internals.

B.2 `AL_SetPacketSize(size)`

Enables the change of the maximum size of the packet (by data-type) transmitted. `size` is a value in bytes. This need to be done for each AutoLink communicating processes, for a test is run at the initialization of the sending and receiving end, to check it.

B.3 `int AL_Send(buf, datatype, dest, tag, comm)`

Will transfer a dynamic data-type following every —first level— pointer in the data-type from `buf` (of AutoLink data-type `datatype`) to MPI rank `dest` with matching `tag` and `communicator`. Returns `MPI_SUCCESS` if no error is encountered.

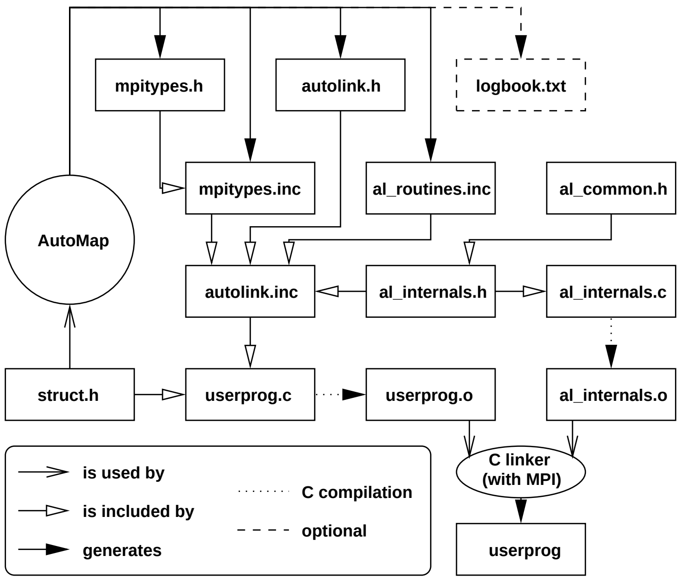

Fig. 4. AutoLink use process

B.4 `int AL_Recv(buf, datatype, source, tag, comm, status)`

Will receive and reconstruct a data-type sent by AutoLink through the command `AL_Send`. The function stores the address of the root node of the recreated dynamic data-type into `buf`, its data-type in `datatype`. This for an element sent from `source` with matching `tag` and `comm`. If given, will store an `MPI_status` into `status`. Returns `MPI_SUCCESS` if no error is encountered.

B.5 `AL_Finalize()`

Finalize AutoLink internals.

C. Internal data-types

For AutoLink data-types and operations on those data-types were designed for efficiency and practical use. We will describe them by the names that are used in the algorithm shown in entry 6 (on page 7) and entry 7 (on page 7).

C.1 `NEXT`

This is a Queue of elements. It allows AutoLink to know the number of elements still to be traversed.

C.2 `''mark''`

The "mark" is defined as a Hash table where each level is made of a Linked List with add in head. It is used by AutoLink to "mark" that an element has already been traversed [7].

C.3 `ADDRESS`

ADDRESS is a dynamically expanded two dimensional array. AutoLink is designed to handle dynamic data-types with *heterogeneous* nodes, thus one dimension is for distinct

[7]the dynamic data-type is to be traversed by passing only one through every possible element

data-types, the second for indexing the data-type elements for such node types. This array is used by AutoLink to store addresses of traversed elements (they are added in it on the recreation end in the same order) for quick access to those elements.

C.4 `LINKS`

Uses C casting facility, and replaces the content of a pointer by the index in ADDRESS (same as "mark") of the pointed element. Since AutoLink works using strong data-type typing, by using the data-type and the index in the data-type storage in ADDRESS, it is possible to find the place in memory of the element.

C.5 `PACKET`

It constitute the PACKET to be sent through MPI by AutoLink. It is a one dimensional array, where each element is a memory space for some elements of each data-types.

D. AutoLink simplified algorithm

The "@" lines in the algorithms are code specified in the data-types specialized code file `al_routines.inc`.

Entry 6 AutoLink simplified sending algorithm

```
Initialization :
|Check coherency of PACKET size with receiver
|  or stop error.
|Add entry node's address in ADDRESS.
|Add entry node's informations in NEXT.
|Mark entry node.

Main loop:
|While there are elements in NEXT
|    |Reach current element in NEXT.
|    |Add current element in PACKET.
|    |@For each son of current node (copy in
|    |  PACKET)
|    |    |@If son is not marked
|    |    |@Then |@If son does not exist
|    |    |      |@Then |@Add ``non existent''
|    |    |      |  in LINKS.
|    |    |      |@Else |@Add son's address
|    |    |      |      |  in ADDRESS.
|    |    |      |      |@Add son's informations
|    |    |      |      |  in NEXT.
|    |    |      |      |@Mark son.
|    |    |      |      |@Add son's mark in LINKS.
|    |    |@Else |@Add son's mark in LINKS.
|    |If PACKET is full, send.
|    |Next element from NEXT.
|Send last PACKET.
|Send initial element index.
```

D.1 Sending

See entry 6 (on page 7) for a simplified version of the sending algorithm.

Entry 7 AutoLink simplified reception algorithm

```
Initialization :
|Check coherency of PACKET size with sender
|  or stop error.

Main loop :
|While there are PACKETs to receive
|    |Receive PACKET.
|    |For each element from PACKET
|    |    |@Create element in memory.
|    |    |Add created element's address in ADDRESS.
|Receive initial element index.
|@For each element in LINKS
|    |@If element is ``non existent''
|    |@Then |@Set son's value to non existent.
|    |@Else |@Set son's value to referred element
|    |      |  in ADDRESS.
|Result is initial element with recreated links.
```

D.2 Receiving

See entry 7 (on page 7) for a simplified version of the receiving algorithm.

E. Discussion

It is straightforward to transfer complex data-types with MPI using AutoMap to create the MPI structures. We have enabled the transfer of types with pointers through the use of the MPI AutoLink library. AutoLink flattens the data-types, sends them, and reconstructs them on the receiving process. All of this occurs "behind the scene"; of the user interface of AutoLink, through the two routines `AL_Send` and `AL_Recv`.

VI. Conclusion

In this article, we have compared two ways for CORBA and MPI to send and receive complex and dynamic data-types, and shown an implementation for MPI of one way of doing so (AutoMap/AutoLink).

This method could also be created in CORBA by encapsulating a similar algorithm in an object definition so that the objects are given a behavior for traversing and reconstructing themselves. Thus the AutoLink algorithm developed for MPI can be used in a CORBA environment.

VII. HTTP references

- MPI data-type tools :
`http://www.nist.gov/itl/div895/auto/`
- NIST :
`http://www.nist.gov/`
- RÉSÉDAS :
`http://www.loria.fr/equipes/resedas/`
- SASP :
`http://www.nist.gov/itl/div895/sasg/`

References

[1] Object Management Group, "The Common Object Request Broker : Architecture and Specification," Tech. Rep., Object Management Group, 1998, `http://www.omg.org/`.

[2] Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein, *Data Structures Using C*, Prentice Hall, 1990.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[4] Compiler Resources, Hopkinton, MA, *Yacc++ and the Language Objects Library Reference Guide*, 1996.

[5] Alan Pope, *The CORBA Reference Guide*, Addison Wesley, 1997.

[6] Robert Orfali, Dan Harkey, and Jeri Edwards, *Instant CORBA*, Wiley Computer Publishing, 1997.

[7] Thomas Mowbray and Ron Zahavi, *The Essential CORBA*, Wiley Computer Publishing, 1995.

[8] Message Passing Interface Forum, *MPI : A Message-Passing Interface Standard.*

[9] "MPI: A Message Passing Interface Standard," HTML document, 1994, `http://www.mcs.anl.gov/Projects/mpi/index.html`.

[10] Ian Foster, *Designing and Building Parallel Programs*, Addison Wesley, 1995, `http://www.mcs.anl.gov/dbpp/`.

[11] Peter Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers Inc., 1997, `http://www.usfca.edu/mpi/`.

[12] K. H. J. Vrielink, E. C. Baland, and J. E. Devaney, "AutoLink: An MPI Library for Sending and Receiving Dynamic Data Structures," in *International Conference on Parallel Computing.* University of Minnesota Supercomputer Institute, october 3-4, 1996.

[13] Judith Ellen Devaney, Martial Michel, Jasper Peeters, and Koen Vrielink, "AutoLink: An MPI C Library For Sending and Receiving Dynamic Data Structures," Tech. Rep., NIST, April 1997, `http://www.itl.nist.gov/div895/sasg/parallel/`.

[14] Judith Ellen Devaney, Martial Michel, Jasper Peeters, and Eric Baland, "AutoMap: A Software Tool for the Automatic Creation of MPI Data Structures From User Code," Tech. Rep., NIST, April 1997, `http://www.itl.nist.gov/div895/sasg/parallel/`.

[15] Delphine Stéphanie Goujon, Martial Michel, Jasper Peeters, and Judith Ellen Devaney, "Automap and autolink : Tools for communicating complex and dynamic data-structures using mpi," *Lectures Notes in Computer Science*, vol. 1362, 1998, Presented at CANPC'98.

[16] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, second edition*, Prentice Hall PTR, Englewood Cliffs, NJ, 1988.

[17] Bjarne Stroustrup, *The C++ Programming Language, second edition*, Addison-Wesley, 1991.

[18] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988.

Disclaimer