Developing Ada Applications in a Distributed Computing Environment

Eugen N. Vasilescu
Sabah Salih
Reinaldo Perez

Northrop Grumman Data Systems
2411 Dulles Corner Park
Herndon, VA 22071

Abstract

Seamless integration of applications in existing MIS environments is difficult because of specific differences in hardware and software characteristics. We investigate issues involved in deploying Ada based applications in standardized distributed environments. We focus mainly on transparent database access and transfer of complex objects between Ada programs across several hardware platforms utilizing multi-vendor DBMS products within the framework of the Open Software Foundation (OSF) / Distributed Computing Environment(DCE).

Keywords -- Ada, MIS, OSF DCE, Remote Procedure Call (RPC), Complex Objects.

1.0 Introduction

The OSF/DCE environment is fast emerging as a de-facto standard for distributed applications. The importance of OSF/DCE has been demonstrated by the use of OSF/DCE as a platform of choice for middleware such as Common Object Request Broker Architecture (CORBA) and Transarc's Encina distributed transaction system while the OSF/DCE RPC specification is used in Windows NT environments.

OSF/DCE is a set of services organized in a comprehensive framework and architecture that supports the creation, use and maintenance of distributed applications in a heterogenous computing environment. OSF/DCE commercial applications are now being deployed by Fortune 100 companies, with brokerage houses and large banks (Charles Schwab, Merrill Lynch, Citibank) making massive investments and adopting aggressive development schedules.

The developing of Ada applications for OSF/DCE revolves around interactions between Ada (and its run-time) and OSF/DCE components.

The OSF/DCE components are([DCE91], [DCE92]): Threads Service (standardized on POSIX 1003.4a, Draft 4), RPC, Naming Service (X.500 based), Time Service, Security Service (Kerberos), Distributed File Service (DFS), Diskless Support Service. Most OSF/DCE

©1994 ACM 0-89791-684-0/94/0006 3.50

vendors support at a minimum the threads service, RPC, time service, name service and the security service, and these components cover substantially all of the functionality required for distributed applications. Threads and RPC constitute the foundation services for the other components and are affecting the most Ada applications.

An essential aspect of OSF/DCE is its explicit focus on the client/server model and its main supporting mechanism: RPC. Because of its layered, standardized, and comprehensive nature, OSF/DCE can isolate applications from specific or proprietary traits of operating systems and products, such as UNIX, OS/2, VMS, and databases. Database Management Systems operating in a client/server environment can utilize the services provided by OSF/DCE to transparently make their services available to processes on different machines.

The establishment of an environment in which several client and server machines may operate is a critical step in permitting communications between different machines. RPCs can be used to help solve the problems of data transfer between client and servers residing on different platforms by providing a transparent means of communicating potentially complex data types. Additional flexibility and design independence can be gained by leveraging OSF/DCE name services.

The OSF/DCE RPC implementation uses the DCE threads component which in turn overlaps in functionality with the Ada compiler run-time. As we will explain further in the paper, this will most likely conflict with the tasking mechanism and its run-time support, making the use of Ada tasking in DCE a thorny issue.

The RPC service, while essential to OSF/DCE, is greatly enhanced and complemented by the other DCE services. Making judicious use of all core DCE services in developing Ada applications allows for conceptually cleaner and more robust distributed systems.

In deploying our Ada prototypes over DCE we started with existing Sun ONC distributed applications developed using the methodology of [VASI91] and [VASI92]. This methodology fits the layered architecture of OSF/DCE because it can target in an orthogonal manner (persistent) resources, communication services or security services. In addition the methodology is supported by automatic code generators allowing for quick testing of targeted features.

The paper first describes issues arising from the interaction of Ada with some core DCE components. Next, we give a quick overview of our methodology and its application to DCE prototypes. We continue with lessons learned during prototype development covering mainly OSF RPC (which is contrasted with Sun ONC RPC), security and name services. The appendix lists sample Ada code describing complex objects and their automatic translation into DCE Interface Description Language (IDL).

2.0 DCE Threads/RPC Interaction with Ada

The development of Ada applications using OSF DCE RPC and Ada tasking is difficult because two runtimes unaware of each other are involved in the management of the same resources. The Ada runtime is required whenever tasks are used. RPC requires threads services which in turn bring in their own runtime. Within OSF DCE the need to interface Ada with non-Ada libraries is pervasive. This usually represents additional problems because some libraries (such as Motif or X11 R5) are not thread-safe.

The difficulties arise when the Ada runtime system captures the process and, among other things sets up signal (interrupt) handlers for all Ada important signals. The Ada runtime needs control over the process to manage tasking, exceptions, exception handling, and Ada I/O. However, it is not aware of signals meant for the threads runtime, and does not know how to handle these signals. The consequence is that Ada

programs will crash unexpectedly when these signals are encountered.

On the other hand, the use of Ada libraries in a threaded environment is made possible by some compilers provided no use of the Ada runtime is made (no use of tasking, I/O, exceptions). As specified in the Ada reference manual all Ada subprograms are reentrant and therefore these subprograms can be made thread-safe by following a few simple guidelines (such as the restricted use of global objects).

There are common techniques (such as the use of jacket routines) that can be used for the integration of non-threaded libraries from other languages not developed for a concurrent environment. These techniques are used for instance by DCE threads packages to provide thread-safe OS libraries, and were employed when needed by our Ada prototypes.

The Ada tasking interaction with threads varies somewhat on whether it deals with user or kernel level threads. A user-level thread implementation has a runtime system that is layered over the OS kernel while kernel threads implement the required functionality in kernel space. In a threaded environment threads share open files, timers, and signals, etc. Because the thread concept is in fact very close to the tasking concept, integrating user-level thread runtimes and Ada runtimes amounts to integrating two related concepts having entirely separate and conflicting implementations while the environment outside the process supporting these implementations is not able to really distinguish each implementation. In this context application developers are normally forced to drop the use of Ada tasking and the development of reliable Ada tasking/DCE applications is virtually impossible.

A better chance of integrating Ada runtimes and thread runtimes arises if OS native threads are available and used by both runtime implementations. If both the DCE and Ada compiler vendors choose to map their appropriate runtime specifications onto kernel level threads they may also delegate to the OS much of their functionality. For instance the OS might be able to manage the signals, scheduling, and dispatching of threads on behalf of the two runtimes.

In this context the use of Ada tasking with OSF DCE becomes at least feasible, even though one expects to give up some flexibility on the Ada side. When the management of tasks is delegated to OS one might expect that Ada tasking behavior is dictated by the OS. Ada applications might not have the flexibility of modifying the scheduling algorithms, or might not control all threads of execution in the process, and might be required to map the priority levels to those of the operating system.

Newer Operating systems (Solaris 2.X, OS/2 2.1, Windows NT) support native (kernel level) threads and vendors for these platforms are now moving rapidly to allow for the reliable coexistence of Ada and thread runtimes.

3.0 Prototyping Methodology

Our prototyping methodology assumes a modular definition in Ada of high level data structures able to model complex objects. Implicit semantics are associated with modules of interest. The functionality expressed by these semantics is made available as subprogram specifications to client applications. The methodology calls for choosing semantics that can be supported through well-defined mechanical transformations.

For instance, the appendix lists Part_C recursive data structure as a record type with variants. The complex object in this case consists of the parts-subparts hierarchy together with the suppliers of each base part. Implicit semantics for persistent resources are associated with Insert, Retrieve, Delete specifications. For communication the primitives might be Send/Receive. For security the semantics might call for setting of protection level attributes during communication.

Considerable care and analysis needs to be invested in choosing semantics that can be mapped onto standardized layers of interest in a computationally efficient manner. For instance, in describing high level objects suitable for database manipulation, one needs to follow a tight discipline in order to avoid endless recursion in part-

subpart hierarchies. Or, in shipping high level objects over networks, the semantics should be chosen so that execution of deep copy of an object is possible.

In general, the methodology limits the complexity of data structures for high level complex objects to a model capable of compact description of Directed Acyclic Graphs (DAG). This level of complexity (DAG-like structures) is fully adequate to support object-oriented modeling with minimum "impedance mismatch". In particular, it is adequate for expressing objects of roughly equivalent complexity and efficiency with those of CORBA IDL [CORB91] and ODMG-93 [ODMG93].

Crucial to this methodology is an environment where stable and standardized layers can be targeted. Standardized layers for persistent resources might be SQL-compliant or SAG-CLI compliant databases. For communication resources one may target Sun ONC RPC or DCE RPC.This methodology is designed to facilitate the creation of code generation toolkits that automatically bridge the gap between high level Ada-based objects (or other object-oriented host languages) and the targeted layers of interest. In fact, we developed toolkits supporting relational databases and communication resources following the general approach of Figure 1 on page 4.

The methodology is usable in a variety of contexts when rapid prototyping and proof-of-concept software is needed. It can be used for implementations of prototype APIs. It can be used as well in situations where APIs are not yet well defined but some desired semantics are. It is tailored to support a coherent and disciplined approach to interoperability in open environments where new and emerging standards require "glue" code generation, communication support, and thorough compatibility testing. Because implemented semantics are encapsulated and kept independent of semantics targeting different API, one achieves a high degree of modularity.

For instance, RPC support is not tied to DBMS support. One can have several application programs running on different platforms exchanging directly complex objects without any database interaction. When dealing with complex data structures like an inventory parts schema, data can be passed across systems at different levels. Data may be shipped at a very high level where the whole data structure is passed requiring very few RPCs in the server interface or at a very low level using more RPCs for passing each sub-component of the data structure. In fact the order in which the RPC semantics or database semantics are applied may be determined by the application programmer and guided by efficiency or practical considerations dependent on the platform in use.

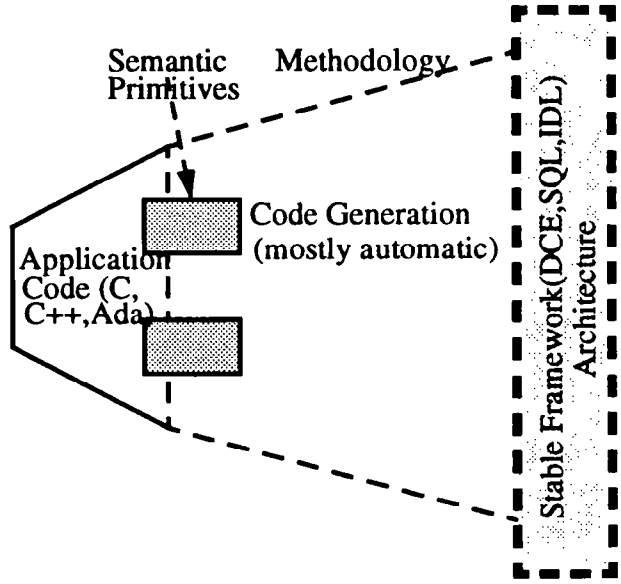

Figure 1. Methodology/Prototype Approach

An important by-product of this methodology is its ease of use. Application programmers need to concern themselves mainly with high-level types and object declarations, while the access to persistent or communication resources is non-procedural, automatic, and through a small number of semantically well-defined subprogram calls. Prototype implementations of this methodology over almost five years demonstrated full portability and high programmer productivity.

4.0 Prototype Implementation and Lessons Learned

OSF/DCE is a well balanced integrated environment. Its main services (RPC, name services, security services and thread services)

offer application developers unmatched flexibility and robustness.

Our Ada experience using OSF DCE can be classified into three general categories:

- The extent to which Ada can make full and unrestricted use of OSF DCE services.
- The extent to which our prototyping methodology facilitates deployment of Ada applications in OSF DCE.
- A comparison of DCE RPC and ONC RPC.

4.1 Using OSF/DCE Services

The name service provides a global addressing system for any kind of entity that permits multiple binding methods and profiling of servers. The security services provide secure RPC's through the exchange of secret passwords and encryption of data between distributed systems. The threads services permit multiple clients to simultaneously access a server. These are some of the services that provide the flexibility but also introduce the level of complexity encountered in DCE.

The use of these services from Ada is unrestricted. However, the caveat of section two holds: Ada tasking is not to be used. A server can be non-secure or be authenticated, can handle one or multiple calls at a time, and be stand-alone or replicated. A client can use automatic, explicit, or implicit binding, can talk to multiple servers simultaneously, and can rebind to a backup server when there is a failure.

DCE applications use the name services to help locate binding information to build a binding handle. Client-Server binding is a two step process that clients perform to find on what host the server is running (using the Name server) and to find what endpoint (using the endpoint mapper) the server is listening to (for a good introductory explanation of DCE RPC see [SHIR93]). Automatic bindings are used when transparency is desired and calls are limited to a single server. Simple LAN applications where the identity of the server does not matter are good candidates for automatic bindings. Implicit bindings are used when control of binding management is desired yet there is no requirement for a visible binding handle in the RPC and calls are mainly to one server. Elaborate applications use explicit binding where the decisions to bind to multiple arbitrary servers is required for greater program flexibility. Explicit binding applications make the name server calls (binding management) to obtain the binding handle for each server and pass the binding handle in each RPC.

The Name server can also provide the flexible lookup of object entries by using groups and profiles. Name server object entries can be grouped into container objects similar to directories in files systems or can be profiled to control the lookup of available servers to perform operations like load balancing. Application servers register themselves by exporting their interface into the server namespace and adding themselves to groups. Client applications can do lookups on a single container object entry to access the dynamically configured server binding information.

This environment permits the dynamic configuration of available servers. Whole name server hierarchies can be included into profiles without any client configuration. Client applications can access a single entry that will prioritize and choose the appropriate service according to the policies implemented.

For example, a number of print servers can be organized by class and be prioritized by proximity. When clients request to print, the binding returned will depend on the class of printer, and the availability of the closest printer.

In the development of a DCE application the method of binding and the grouping/profiling of server entries define its behavior and flexibility. There are numerous options in implementing DCE services which dramatically change the behavior of applications without client modification or even recompilation.

As a rule of thumb, we were able to exercise in our Ada applications an almost complete range of options in terms of bindings, grouping/ profiling, security. Calls available from C were duplicated in Ada with little effort. Yet it became clear quite soon that developing DCE applications is akin to developing low-level X applications because of the large number of features available in the DCE services APIs.

For instance, the authenticated RPC applications were developed using the DCE Security services based on Kerberos version 5. Integrating authentication into our applications required the development of an Ada wrapper package that would make the DCE calls through C. Authentication required little effort because it amounts to simple attribute setting and modifications. The implementation of authorization is a different matter. DCE authorization is implemented through DCE privilege service that provides access control lists (ACL). The implementation of ACL managers for application servers to control access to server objects is quite difficult and time consuming. The current state-of-the-art requires developers to develop ACL management services to conform to a predefined DCE RPC interface. There are no tools or standard approaches for developing ACL managers. This forces developers to conform to a complex predefined API and to handcraft its precise semantics as well.

We avoided to a large extent the uncontrolled proliferation of APIs through the use of our methodology. In our prototypes we targeted some standardized use of DCE services. When applications are generated the type of binding, profiling and security developed are automatically integrated.

4.2 Ada Prototypes over OSF DCE

Our prototype work started in 1989 and originally supported SunOS environments and Oracle. It has expanded to several other platforms (OS/2, Windows NT) and databases (SQL Server, Sybase, Informix, object-oriented databases such as ObjectStore and Versant), with automatic code generation for relational database support and Sun RPC. And starting late 1992 we extended its reach to OSF/DCE. The prototypes benefit from our methodology and supporting toolkits which now successfully generate IDL code supporting DCE RPC.

Thus we can efficiently ship complex objects across heterogenous platforms taking advantage of DCE services for a variety of bindings, server profiling, secure RPC, multithreaded servers.

Our current testbed includes two DCE cells, one cell configured for the Sun environment running Transarc's DCE product, the other cell running IBM's DCE implementation on OS/2. We also use MS RPC, a Windows NT compatible implementation of DCE RPC, to communicate with OSF/DCE servers.

OSF DCE provides extensive APIs covering many services including threads, RPC, Name, Security services. The extent to which the API is used depends on the complexity of the application. For example, it is possible to develop an elementary DCE application that uses the RPC runtime with well-known endpoints. In this case no use is made of the name server, security server, or the end-point mapper that DCE provides.

As application complexity rises the developer is confronted with a multitude of calls to be mastered, making for a steep and long learning curve. In addition, the potential for subtle interactions among these calls is high, and these interactions easily translate into implementation pitfalls. This certainly calls for commercially available toolkits or facilities that application developers can use to target standardized OSF DCE components, and such efforts are currently under way.

Our prototypes benefited from the mechanisms supported by our tools. The application developer is not concerned about SQL data modeling, database access, interface development, and distributed server development. It is all available to him via high-level Ada-driven specifications.

Also, because the Ada support for the security component is independent of any particular security mechanisms and independent of the

communication and database support component, we gain the flexibility of swapping and integrating other security mechanisms as well.

Our methodology allowed us to test a variety of partition options in a client/server environment. When dealing with complex data structures like an inventory parts schema, data can be passed across systems at different levels. Data may be shipped at a very high level where the whole data structure is passed or at a very low level using more RPCs for passing each sub-component of the data structure. The type and size of data structures shipped across systems must be carefully considered in order to assess the overhead associated with each RPC.

Passing data at a high level required very few RPCs in the server interface. Developing low level RPC calls partitioned the application differently. The server interface contained large numbers of calls and required the clients to make multiple RPC calls for the equivalent of a high-level RPC call.

Passing data at a high-level proved efficient when deep hierarchies were involved. Passing data at a lower-level was acceptable when hierarchies were shallow and involved large elementary opaque objects. Of course, using high-level RPC provides better support when scalability and interoperability of the application is an issue. Other implementations options included packing and unpacking data structures for transfer through a DCE pipe (shipping of files of larger size -such as images- are well suited to the pipe mechanism.)

The prototype brings together many technologies and languages into a single environment. It addressed successfully many pragmatic issues like linkage (shared vs. static), libraries with thread support, non-thread safe libraries, and integration with DBMS server libraries.

A key element in developing distributed applications is the existence of adequate compilers and debugging tools. For example, the compiler we used did not handle shared libraries well, nor was it threads aware. An additional difficulty in debugging distributed applications is the handling of communication time-outs. Due to the present compiler and tool limitations, the development of Ada distributed applications can be difficult.

4.3 Comparison Between OSF/DCE RPC and Sun ONC RPC

While adding OSF/DCE services to our prototype a number of differences between OSF/DCE and Sun ONC distributed services were exposed. One of the prototype activities is to automatically generate a server interface file from the Ada source code (user schema and application files). This generation step, originally targeted to ONC RPC, was modified to accommodate DCE RPC. ONC and DCE have different approaches with respect to user defined types, parameter passing, RPC server registration, support for pointers, and DCE has additional enhancements like string or pipe types.

When developing client-server applications the most widespread communication facility is the Sun ONC RPC. The communication facility provided by SUN ONC has a solid market penetration due to its low cost and availability on many platforms. We feel however that OSF DCE has some outstanding technical features that make it a better distributed framework. Here are some differences between ONC RPC and DCE RPC.

1. When using ONC RPC one needs to specify the server name when making a call. In contrast, DCE allows for simple and transparent implementation of server location independence. Clients use a standard name server to locate servers in DCE, and then get the required binding.

2. One needs to specify the XDR filter for function parameters when making client calls in a Sun ONC environment. Also ONC client calls are much more complicated. This means when

developing servers whose data format changes, major code changes occur in all clients. In contrast, DCE function calls look just like local calls, and little client modification is required when data format changes.

3. ONC RPC is more restricted about the size of data being transmitted, and requires low-level calls for managing larger size data. When shipping large amounts of data, the programmer must manage the connection between client and server to guarantee data delivery. In DCE RPC large user defined data structures are transparently supported. In addition, DCE provides support for a pipe type at the interface compiler level.

4. Sun ONC servers cannot easily be made multi-threaded. In contrast, automatic support of multi-client servers is provided by DCE.

5. When using Sun ONC one needs to register every server call on the server side. When using OSF DCE, the server development is simplified by registering the service just once, with no per call requirement.

6. ONC clients must pass single pointers to all objects instead of the actual objects. DCE has no restrictions as to what data and how many parameters are passed. The data to be transferred and the direction is specified at the protocol level in DCE.

7. ONC has no support in the protocol compiler for callback. DCE provides the support for callbacks thus allowing the development of asynchronous RPC. This allows background execution of long task.

8. When using DCE the selection of transport protocol is dynamic and can be made transparent to the application. Thus DCE provides the framework to access other network types without major changes to source code.

5.0 Conclusions

Ada can be used right now to develop applications for OSF DCE. If the tasking feature of Ada is avoided, applications can take full advantage of the superior technical features and the powerful framework of OSF DCE.

The use of Ada tasking with DCE in a portable way might be achieved if a Posix/Ada standard is adopted and supported by compiler vendors. In the meantime, the use of Ada tasking is possible in a proprietary manner to the extent that compiler vendors integrate Ada and DCE threads runtimes.

There is a fairly steep learning curve associated with OSF DCE due to the extensive functionality of each DCE component and the complexity of their APIs. The application development is fairly tedious and there is a need for production strength middleware supporting tools.

While we could make qualitative assessments on OSF DCE (in particular by contrasting Sun ONC RPC and DCE RPC), more experiments are needed for careful quantitative analyses of OSF DCE performance in large distributed systems.

Our prototypes avoided to a large extent the tediousness associated with DCE development because we applied our methodology and associated tools to automatically support target DCE components. We successfully generated support code for complex objects and independently for SQL-based database servers and security services. This capability is unique in both Ada and non-Ada environments.

These prototypes demonstrate now interoperability at complex objects level in a heterogenous DCE cell (OS/2 PCs and Sun workstations) with different persistent resource managers (relational and object-oriented databases). This is seen as a prerequisite for high-level support of distributed transaction processing on OSF DCE, which is the focus of our future efforts.

6.0 Bibliography

[VASI91] Vasilescu, E., "Using Ada for Rapid Prototyping of Database Applications", Proceedings of the Eighth Washington Ada Symposium, 1991.

[VASI92] Vasilescu, E., Salih, S., Skinner, J., "Maintaining Transparency of Database Objects over networks in Ada Applications", 10th Annual National Conference on ADA Technology, 1992.

[DCE91] Open Software Foundation, "Distributed Computing Environment: An Overview", OSF White Paper, April 1991, Cambridge, Mass.

[SHIR93] Shirley, J., "Guide to Writing DCE Applications", O'Reilly & Associates, In.,1993.

[DCE92] Open Software Foundation, "Application Development Guide", OSF DCE 1.0.1. , 1992, Cambridge, mass.

[CORB91] Object Management Group , "OMG CORBA" - OMG Document Number 91.12.1 Revision 1.1

[ODMG93] "The Object Database Standard: ODMG-93", Morgan Kaufmann Publishers, San Mateo, Ca.

7.0 Appendix

```
************************************
Sample schema file sch_parts.a
************************************
-- Complex object data structures
-- Implicit semantics:
-- Insert, Retrieve, Delete, Send, Receive
-- Set, Get authentication attributes
with Parts_Types; use Parts_Types;
package Parts_Schema is
use Parts_Types.Ada_Sql;
type Made_From_C;
type Access_Made_From_C is access
  Made_From_C;
type Suppliers_C;
type Access_Suppliers_C is access Suppliers_C;
--Part_C is your only root
type Part_C (Part_Kind : Part_Class :=Base_Part)
  is
record
  No_Ref        : No_Type;
  Name          : Part_Name_Type(1..10);
  --Next_Part_C     : Access_Part_C;
case Part_Kind is
  when Base_Part =>
    Cost       : Cost_Type;
    Mass       : Mass_Type;
    Supplied_By : Access_Suppliers_C;
  when Composite_Part =>
    Assembly_Cost: Cost_Type;
    Mass_Increment: Mass_Type;
    Made_From   : Access_Made_From_C;
end case;
end record;
type Access_Part_C is access Part_C;
type Made_From_C is
record
  How_Many       : How_Many_Type;
  Component      : Access_Part_C;
  Next_Made_From_C : Access_Made_From_C;
end record;
type Suppliers_C is
record
  Name_Ref       : Supplier_Name_Type(1..10);
  Next_Supplier_C  : Access_Suppliers_C;
end record;
end Parts_Schema;

************************************
Sample Application file parts.a
************************************
procedure Parts is
use Parts_Types.Ada_Sql;
The_Result_List : Access_Part_C_Join;
The_Part : Part_C;
The_Part2 : Part_C(Composite_Part);
```

```
begin
Parts_Utilities.Construct_Bike;
Open_Database("Johns","Bass",dion);
Put_Line("Insert the bike");
Insert(Parts_Utilities.PC3,dion);
Put_Line("Insert the wheel");
Insert(Parts_Utilities.Pc1,dion);
Put_line("Retrieve(The_Part,
R_A_1 (The_Part.Name = Bike)); (Bike)");
The_Result_List := Retrieve(The_Part,
  R_A_1(The_Part.Name = "bike     "),dion);
--Check what we get back
while The_Result_List  /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
end loop;
Put_Line("Delete the bike");
Delete(The_Part,R_A_1(The_Part.Name =
"bike     "),dion);
Put_Line("Delete The Bike");
Delete(The_Part,R_A_1(The_Part.Name =
"bike     "),dion);
The_Result_List := Retrieve(The_Part,
  R_A_1(The_Part.Name = "bike     "),dion);
-- Check what we get back
while The_Result_List /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
  end loop;
Put_Line("Retrieve(The_Part,
R_A_2(The_Part.No_Ref = 33333));
(WHEEL)");
The_Result_List := Retrieve(The_Part,
  R_A_2(The_Part.No_Ref = 33333),dion);
-- Check what we get back
while The_Result_List /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
  The_Result_List.Part_C_Join;
  end loop;
Put_Line("Insert The Bike");
Insert(Parts_Utilities.PC3,dion);
Put_Line("Retrieve(The_Part,
R_A_3(The_Part.No_Ref = USER_VAL));
(SPOKE)");
The_Result_List :=
  Retrieve(The_Part,R_A_3(The_Part.No_Ref =
  USER_VALUE),dion);
-- Check what we get back
while The_Result_List /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
 end loop;
Put_Line("Retrieve(The_Part,
R_A_4(The_Part.Mass = 2");
Put_Line("OR The_Part2.Mass_Increment =
25");
Put_Line("OR The_Part.No_Ref = 55591 OR
The_Part.Name = bike);");
The_Result_List := Retrieve(The_Part,
  R_A_4(The_Part.Mass = 2 OR
  The_Part2.Mass_Increment = 25 OR
  The_Part.No_Ref = 55591 OR The_Part.Name
  = "bike     "),dion);
-- Check what we get back
while The_Result_List /= null
  loop
```

```
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
  end loop;
Put_Line("Retrieve(The_Part,
R_A_5(The_Part.Name /= tire AND");
Put_Line("(The_Part.Mass = 5 OR The_Part.Cost
= 1)");
The_Result_List := Retrieve(The_Part,R_A_5
  (The_Part.Name /= "tire     " AND
  (The_Part.Cost = 1 OR The_Part.Mass
  =5)),dion);

-- Check what we get back
while The_Result_List /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
  end loop;
Put_Line("Retrieve(The_Part);    get all parts");
The_Result_List := Retrieve(The_Part,dion);
- Check what we get back
while The_Result_List /= null
  loop
  Parts_Utilities.Display(The_Result_List.Part_C
  _Elem);
  The_Result_List :=
    The_Result_List.Part_C_Join;
  end loop;
Put_Line("Delete(The_Part);    delete all parts");
Delete(The_Part,dion);
Exit_Database(dion); -- generated automatically
end Parts;
```

```
************************************
Sample idl  file ap.idl
************************************
[
uuid(00972d5e-71b3-1d90-98fc-84e45e38aa77),
version(1.0),
pointer_default(ptr)
]
interface Parts
{
 typedef   struct made_from_c
*access_made_from_c;
 typedef   struct suppliers_c
*access_suppliers_c;
 typedef enum  {
       BASE_PART_PART_CLASS,
       COMPOSITE_PART_PART_CLASS,
       END_PART_CLASS} part_class;
typedef struct base_part_part_c{
   long no_ref;
   char name[12];
   long cost;
   long mass;
    access_suppliers_c supplied_by;
} base_part_part_c;
typedef struct composite_part_part_c{
   long no_ref;
   char name[12];
   long assembly_cost;
   long mass_increment;
    access_made_from_c made_from;
} composite_part_part_c;
typedef union part_c switch
   (part_class part_class) {
 case BASE_PART_PART_CLASS:
   struct base_part_part_c base_part;
 case COMPOSITE_PART_PART_CLASS:
```

```
   struct composite_part_part_c composite_part;
  default: ;
  } part_c;
typedef    union part_c *access_part_c;
 typedef struct made_from_c{
   long how_many;
     access_part_c component;
     access_made_from_c next_made_from_c;
} made_from_c;
typedef struct suppliers_c{
   char name_ref[12];
     access_suppliers_c next_supplier_c;
} suppliers_c;
typedef    struct part_c_join *access_part_c_join;
typedef struct part_c_join{
      part_c part_c_elem;
     access_part_c_join part_c_join;
} part_c_join;
typedef    union part_c_variant_join
*access_part_c_variant_join;
typedef struct base_part_part_c_variant_join{
     access_part_c_variant_join part_c_join;
      part_c part_c_base_part_elem;
} base_part_part_c_variant_join;
typedef struct
composite_part_part_c_variant_join{
     access_part_c_variant_join part_c_join;
      part_c part_c_composite_part_elem;
} composite_part_part_c_variant_join;
typedef union part_c_variant_join switch
   (part_class part_class) {
  case BASE_PART_PART_CLASS:
   struct base_part_part_c_variant_join base_part;
  case COMPOSITE_PART_PART_CLASS:
   struct composite_part_part_c_variant_join
composite_part;
  default: ;
  } part_c_variant_join;
typedef struct params_1
{ [string] char *p_1;
  [string] char *p_2;
} params_1;
void OPEN_DATABASEX_1_1([in] params_1*
p);
void INSERTX_1_1([in] part_c* p);
access_part_c_variant_join
      RETRIEVEX_1_1(void);
void DELETEX_1_1(void);
access_part_c_variant_join
RETRIEVEX_2_1(void);
access_part_c_variant_join
RETRIEVEX_3_1(void);
access_part_c_variant_join
RETRIEVEX_4_1(void);
access_part_c_variant_join
RETRIEVEX_5_1(void);
access_part_c_variant_join
RETRIEVEX_6_1(void);
void DELETEX_2_1(void);
void EXIT_DATABASEX_1_1(void);
}
```