29

Using OMG IDL to write OODCE applications

John Dilley <jad@hpl.hp.com>
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304, USA

Abstract

The Object Management Group's Interface Definition Language (OMG IDL) provides a standard for specifying object-oriented interfaces for distributed applications. This paper describes a compiler that allows specification of object interfaces using OMG IDL and implementation using the Open Software Foundation's Distributed Computing Environment (OSF DCE). Using this approach application developers are able to use OMG IDL to define their distributed object interfaces while using the facilities of OSF DCE for remote communications, including the DCE Security Service and DCE's object location service (CDS). This provides a very low-cost infrastructure for the development and deployment of (CORBA-based) object-oriented distributed applications.

Keywords

DistributedComputing(C.2.4);NetworkProtocols(C.2.2);Object-OrientedProgramming(D.1.5)

1. INTRODUCTION

OMG IDL is an object-oriented interface specification language which is rapidly becoming the industry standard for distributed object definition. An IDL interface identifies a set of types and operations that define a contract between a client and server, similar to how a C++ class defines types and methods for local object-oriented application developers to use. IDL interfaces can use the data types and operations from one or more other interfaces through (interface) inheritance. Once an interface is defined, clients communicate with object implementations using a remote method invocation (effectively a remote procedure call) protocol.

OMG IDL is one component of the Common Object Request Broker Architecture (CORBA) Object Management Architecture (OMA) [OMG91-12-1]. CORBA and the OMA also specify an architecture for how an Object Request Broker facilitates communication between client and server, and the interfaces for a set of facilities (Object Services) required to make the distributed environment viable.

A. Schill et al. (eds.), *Distributed Platforms*

The Open Software Foundation's Distributed Computing Environment (OSF DCE) [OSF92] provides a distributed computing platform with a core set of services that supports a Remote Procedure Call (RPC) model for client/server communication. The RPC mechanism has been developed and improved over a span of more than 15 years (since NCS RPC, the progenitor of DCE RPC was released), and today provides a robust platform for distributed application development. DCE also contains a set of services, built using RPC, to provide global object location (the Cell Directory Service), secure communications providing authentication of the client and server as well as client access control on a per-object basis (the Security Service), a global Distributed File System (DFS), and a Distributed Time (synchronization) Service (DTS).

DCE provides essential components for building and deploying enterprise-wide distributed applications. The intent of our work is to leverage and reuse the DCE components as efficiently and naturally as possible to support CORBA-based application development.

1.1. CORBA and DCE – similarities and differences

The basic model for communication between entities in both the CORBA and DCE distributed environments is one of remote procedure calls: the caller locates a remote object instance it wishes to communicate with, sets up an association with that object (binds to the object), and makes a procedure or method call to a local proxy object, which transmits zero or more input parameters to the remote process, causes the remote procedure to execute given those parameters, and when it completes returns the results to the user in output parameters and a return value. The intent behind the RPC paradigm is to provide the developer with the same model for communicating with local and remote procedures or object instances. Since both CORBA and DCE are based upon a synchronous RPC model, supporting one atop the other seems natural.

One difference between the environments is evident when one examines the product definition and deployment models of the OMG and the OSF. The OMG provides the specifications of an object-oriented distributed software system architecture, and defines standard interfaces for the distributed services in that environment. This allows for flexibility in implementation, and for the CORBA architecture to be applied at various levels and types of (distributed and even non-distributed) systems. By contrast, the OSF defines a set of services and provides a software reference implementation; OSF DCE defines interfaces as well as a default implementation. (Other DCE implementations exist, most notably the independently developed MS-RPC from Microsoft which follows the DCE specification but does not use OSF's code base). The difference in models leads to a greater diversity among CORBA offerings, and a more homogeneous DCE offering.

OMG also has focused more on the object-oriented quality of application development, where OSF has focused more on providing facilities for application distribution. The focus on object-orientation is evidenced in the Object Management Architecture, and by OMG IDL's inclusion of interface inheritance, and its syntactic similarity to C++. By contrast, DCE's focus on distribution is evidenced by its required use of global unique identifiers for all interfaces and objects, interface versions, control over operation semantics and data marshaling in IDL, and data types and marshaling facilities tuned for distribution (e.g., pipes, control over transmit format from the IDL).

1.2. OODCE

OODCE [Dill95] is a C++ programming environment for DCE that provides greatly enhanced ease of use through encapsulation of DCE's programming complexity, and provision of a set of pre-packaged capabilities. It supports the existing DCE object model, rather than adding a new notion of object-based computation to DCE. To use CORBA terminology, OODCE essentially provides a C++ language binding for OSF DCE. The OODCE object interaction model is very similar to that of CORBA–where clients make local method calls on proxy objects in order to invoke a remote implementation.

OODCE provides the structure for DCE-based distributed applications in C++. The OODCE framework defines a common interface and default methods for binding and object location, specification of security preferences, registration of implementation objects with the environment, etc. These are some of the same tasks an Object Request Broker (the ORB in CORBA) and Object Adaptor must provide. The framework classes interact with DCE facilities such as the Security Service or Cell Directory Service in order to register objects in the environment, allow location and binding, specify security preferences, and so on. OODCE does not provide any of its own distribution services, rather it makes full use of the services DCE provides. OODCE does augment the DCE object model in the area of object references, by defining a location independent object reference format, and by specifying a protocol for object activation and deactivation. The activation protocol allows server developers control over the lifetime of the (memory for) C++ objects that implement the behavior of DCE objects, independent of the client's view of the object's lifetime. OODCE also provides an object factory interface, which gives clients direct control over the lifetime of the DCE objects they wish to access.

2. THE OMG-OODCE COMPILER

The OMG-OODCE compiler translates an OMG IDL interface definition into a set of corresponding DCE IDL interface definitions and C++ classes. The DCE interfaces allow the DCE IDL compiler to generate the communication stubs (client and server) for the application. Communication stubs provide distribution transparency for distributed environments (CORBA and DCE). Distribution transparency gives application developers a higher-level view of distributed computing; communication stubs provide for low-level communication fault handling, hide potentially different internal data formats on the communicating hosts, and support object binding (object location and relocation). The C++ classes provide the developer with convenient access to the objects defined in OMG IDL. These generated classes inherit some of their interface and behavior from OODCE framework classes.

One promise of object technology is software reuse. This effort attempts to deliver on that promise by reusing a full-service distributed computing environment to provide a CORBA-based environment. The philosophy behind this work has been to create an OMG IDL compiler as efficiently as possible through software reuse. As a result the compiler relies heavily upon the facilities of DCE and OODCE, rather than defining and building a separate (and quite likely non-interoperable) set of distributed services. Our approach provides CORBA compliance by using OMG IDL to define application interfaces, while making very direct and efficient use of the underlying distributed communication platform.

2.1. Advantages of this approach

Use of OMG IDL has some benefits over DCE IDL: OMG IDL is a more object-oriented interface specification language, providing the ability to express interface inheritance, define parameterized exceptions, and to pass objects by reference in RPCs more conveniently. However, OMG IDL does not fully address the distribution of objects. For instance, OMG IDL does not provide a standard way to identify interfaces or interface versions; nor does it provide a way to provide for context between a client and server. While some of these issues are being addressed in the CORBA 2.0 specifications, using OSF DCE as the communication substrate can compensate for these shortcomings today.

Furthermore, every distributed environment requires some (coordinated) administrative and management services. Developing these distributed operational services is expensive and non-trivial to say the least. By using DCE directly only one set of administrative and management tools will needed to maintain the entire distributed environment. The alternative in a mixed DCE/CORBA environment would be to have a separate set of DCE and CORBA tools which administrators must learn and use. Equally significant, only one global directory name space, and one security registry database are needed. Having multiple disjoint directory services and security databases would cause obvious problems for application developers, administrators, and end users.

Finally, OODCE is a second-generation DCE technology, which provides a high productivity development environment. The first release of DCE, like the first release of most powerful new technologies, was difficult to learn and use. With the OODCE framework, many of the tasks in developing a distributed application are supplied by the default behavior, simplifying the task of application development.

2.2. Interoperability

Applications developed using the OMG-OODCE compiler will be able to act as servers for standard DCE or OODCE applications, since all objects use the same underlying (DCE) communication infrastructure. An object specified using OMG IDL can be accessed via either its OMG-OODCE generated stubs, or the corresponding DCE IDL communication stubs. This allows DCE-only platforms (such as the PC or MVS) to access OMG IDL-defined applications built using this compiler.

However, it is not possible in general to write an OMG IDL definition such that the compiler-generated DCE IDL file will be compatible with a preexisting DCE IDL description, since the preexisting DCE IDL may have used language facilities that are not accessible through this compiler, such as pointers, pipes, context handles, or the **`transmit_as`** attribute.

Interoperation with CORBA 2.0 applications using a protocol called Universal Networked Objects (UNO) [OMG94-9-32] should be possible using a half-bridge (protocol translator, or gateway) from the UNO protocol to the DCE protocol (we have not verified this, however).

2.3. Application portability to a CORBA environment

Applications built using OMG IDL and employing this compiler will be able to migrate to other CORBA environments much more easily than regular DCE applications for a number of reasons.

- These applications will already be using OMG IDL to specify their interfaces, so there is no IDL port required. Since IDL expresses the application design at a high level, this reduces the likelihood of an expensive redesign phase during porting.
- Applications will be designed and written using object-oriented techniques, which means the programming model will not have to change from procedure-oriented to object-oriented. This again reduces the likelihood of application redesign.
- Application objects will already have C++ implementations. There would be little reason to modify them (other than altering the API to accommodate possibly different data types).
- Finally, an OODCE language binding will provide ease of use for developers (and the compiler writer!) by supplying a class library encapsulating the interface with DCE.

These factors increase the portability of applications using this approach to a CORBA platform, but let there be no question: the migration will not be trivial. In particular, use of DCE facilities such as naming or security will have to be modified to use the equivalent CORBA object service (if available). Any code interacting directly with the RPC runtime facilities (such as for object registration) must change; and since the C++ language mapping is different there will be some issues regarding parameter structure in the different C++ implementations.

3. Technical Overview

The basic approach of this work has been to implement a backend for the public domain SunSoft OMG IDL Compiler FrontEnd (CFE_1.3) [Sun94]. The OMG-OODCE compiler parses OMG IDL and creates one or more corresponding DCE IDL files (one per interface defined in the OMG IDL file). Each DCE IDL file defines data types and remote operations as necessary to support the data types and methods defined in the OMG IDL. The remote operations are generated as a set of C++ classes. Each interface defines a *pure abstract class*, a *client proxy class*, and a *server implementation skeleton*. The pure abstract class defines the signature of the interface in C++–each remote operation defined in IDL becomes a pure virtual member function in this class. The client proxy class inherits from the pure abstract class and provides implementations for accessing the remote operation methods. The implementation provides the distribution transparency mentioned earlier: object location and binding, security, and some enhanced fault handling. The server implementation skeleton defines a class the server developer can implement to provide the remote object's behavior for each of the remote operation methods. In addition to these three classes, a C++ stub is generated to link incoming DCE RPC calls with the appropriate C++ implementation.

3.1. Language mapping

The creation of the DCE IDL and C++ files necessitates a language mapping from OMG IDL to each of the two languages. In most cases the mapping is straightforward, as CORBA and DCE specify very similar RPC-based systems. They each define a set of data types similar to those found in C or C++, and remote operations (basically procedures) that use those data types.

The mapping to C++ is currently based upon the OODCE `idl++` compiler. DCE defines the mapping from its IDL data types to C; `idl++` uses this same data type mapping, since C++ is a proper superset of C. A future version of this compiler could implement the official OMG IDL C++ Language Mapping–but since simplicity in development and use of OMG IDL was a key

goal of this project, that (more complex) standard language mapping has not been implemented to date.

3.1.1 Trivial Data Types

The basic data types map directly from OMG IDL to DCE IDL and C++: **`boolean`**, **`float`**, **`double`**, **`char`**, **`unsigned`**, **`short`**, **`long`**. The CORBA **`octet`** type maps to the DCE **`byte`** type. Many of the other data types map obviously and directly. Minor syntactic modifications are sometimes required since OMG IDL is slightly different from DCE IDL and C++. The trivial mappings are these:

- OMG **`array`** maps to the DCE and C++ **`array`** type.
- OMG **`struct`** maps to the DCE and C++ **`struct`** type.
- OMG **`typedef`** maps to the DCE and C++ **`typedef`**.
- OMG **`const`** maps to the DCE and C++ **`const`**.
- OMG **`enum`** maps to the DCE and C++ **`enum`** type.
- The **`union`** type maps to the DCE and C++ discriminated **`union`**. There is a slight syntactic difference between the two union types so syntactic rearrangement is required.
- OMG **`string`** maps to the DCE **`[string] char *`** type and the C++ (null terminated) **`char *`**.

3.1.2 Restrictions

There are some differences between OMG IDL and DCE IDL which require certain restrictions in the mapping or in the use of OMG IDL. The restrictions are:

- OMG allows unlimited length identifiers; DCE identifiers are limited to 31 characters. When using our OMG-OODCE compiler identifiers must be limited to 31 characters.
- OMG identifiers are case insignificant; DCE is case significant. Our compiler requires identifiers not to differ only in case.
- An OMG **`enum`** allows 2^32 identifiers, while DCE allows only 2^15. Only 2^15 identifiers are allowed by our compiler.
- OMG IDL has name space scoping (using the module and interface constructs); DCE has none. When using this compiler, OMG identifiers cannot rely on the module scoping, but must be unique within the global name space. (Method names from different interfaces can be identical, provided there is no inheritance relationship between them.)

3.1.3 Constructed Data Types

The mapping for other constructed data types follows.

3.1.3.1 sequence

An OMG IDL **`sequence`** is mapped to a DCE IDL conformant array. C++ developers access sequences using the DCE conformant array directly. A sequence template delivered with the compiler provides more convenient access to sequences from C++.

3.1.3.2 attribute

An OMG IDL **attribute** is mapped to a DCE IDL **`get_*`** operation (and a **`set_*`** operation if not declared **`readonly`**) to retrieve (and optionally update) the attribute's value. These operations are also present in the corresponding C++ classes.

3.1.3.3 exception
The OMG parameterized **exception** data type is mapped to a DCE and C++ **struct** type (at declaration time only; see below for the mapping of **raises** in operation declarations). Each data member of the **exception** becomes a member of the **struct**.

3.1.3.4 any
The OMG **any** type is mapped to a conformant array of DCE **byte** data type. The C++ mapping is simply the conformant array. The user is responsible for filling the array with the appropriate data (marshaling and unmarshaling). A mapping to TypeCodes, such as is described in [OMG94-9-14] or in [Vogel95] would make **any** more convenient. However, we feel the overuse of the **any** type can be considered harmful, and discourage its abundant use.

3.1.4 Module
An OMG IDL module may contain an arbitrary number of OMG IDL interfaces. OMG IDL **module** declarations are allowed but basically ignored since DCE IDL (and many C++ implementations) do not yet have nested name spaces.

3.1.5 Interface
An OMG IDL **interface** maps to a set of DCE interfaces and C++ classes; this mapping is described in detail below. An OMG interface may inherit from any number of other OMG interfaces. Interfaces can be declared within the current file or in an included file.

3.1.5.1 DCE Interface Mapping
Each OMG **interface** maps to a single DCE interface. If multiple interfaces are declared within a single OMG IDL file, multiple DCE IDL files will be generated (each DCE IDL file must contain at most one interface). Each OMG interface definition must specify a **#pragma** for the DCE interface UUID and may specify the version number of that interface.

3.1.5.2 C++ Interface Mapping
In C++ an OMG **interface** maps to the following set of classes as discussed earlier:

- A pure abstract class declaring the interface signature in C++. This class is inherited (either directly or indirectly) by all other classes corresponding to this OMG interface.
- A client proxy class. This is the class through which users access remote objects.
- An implementation skeleton class. This class must be implemented by the server developer; it provides the behavior of the object defined in IDL.

3.1.5.3 Interface Inheritance
If an OMG interface *inherits* from other OMG interfaces, the corresponding DCE interface *imports* all of its parent interfaces. The interface signature of the new DCE interface consists only of the new methods, but has access to all the data types defined in the imported interfaces. Note that OMG IDL interfaces cannot redefine (overload) inherited operations.

In OODCE, interface inheritance is supported through the DCEObj abstraction. An OODCE class exporting multiple DCE interfaces contains only a single instance of the DCEObj (virtual) base class. It is this class that identifies the C++ instance as a unique DCE object, and contains references to the (one or many) DCE interfaces supported by the object.

To support polymorphism (calling one of the base class operations on a derived class), the OODCE implementation registers each DCE interface in the inheritance hierarchy separately, but with the same object UUID and object underlying implementation (C++) object. To support this the DCE IDL file for derived interfaces explicitly contain all of the methods found in the

base (parent) interfaces. When a call comes in for an operation on an object, it is dispatched by the server stub to the correct method.

When an OMG IDL interface inherits from other interfaces, typically only the most derived interface is implemented by the developer. In C++, this is the implementation class that inherits from the other interface base classes. The OODCE implementation keeps track of which methods are associated with which interface.

3.1.6 Remote Operations

Remote operations in OMG IDL are mapped to remote operations in DCE IDL, and to function members in the generated classes. No DCE IDL attributes are attached to the operation (e.g., **idempotent**). Operations that contain **raises** or **context** are mapped as follows.

3.1.6.1 raises

The **raises** declaration on a remote operation causes a discriminated (tagged) union to be created in the DCE IDL file; the union has a data member for each exception type that can be thrown by the operation. The union is added as an extra [out] (full pointer) parameter to the DCE operation signature. The parameter is declared as a pointer so that in the normal case only a NULL pointer is transmitted. Only when an exception occurs is the union's value non-NULL.

The server implementation developer can **throw** any of the C++ structs corresponding to the exceptions defined in OMG IDL. The generated server stub does a C++ **catch** to receive the thrown data, and transmits it in the extra **[out] union** parameter added to the DCE IDL. On the client side, if the returned union is non-NULL its discriminant is examined to determine which exception was transmitted. The appropriate struct is extracted from the union and thrown as a C++ exception. The application developer can thus deal only with C++ exceptions, relying upon the underlying middleware to handle the stack cleanup and transmission details.

3.1.6.2 context

OMG **context** is transmitted in DCE RPC as an extra **[in] char *** parameter. There is no standard way yet to retrieve context in the implementation (i.e., the Context class is not supported). In a future version, the Context class will be added to the **DCEClientInfo** which is always available to the implementation.

DCE context handles, which are different from the OMG **context** type, are not supported by this mapping since there is no way to express the notion of context between client and server in OMG IDL.

3.1.7 Interface References

Interfaces can be specified as parameters to remote operations in OMG IDL. Interfaces are mapped to DCE object references for transmission (**DCEObjRefT***) and to the C++ **DCEObjectReference** OODCE class. In CORBA all objects are passed by reference.

When used as an **[in]** parameter, the object reference is extracted from the C++ object being sent (which can be either a proxy object or an implementation object), and transmitted as an object reference to the server (callee). On the server side, a client proxy object is constructed in the EPV using the binding information from the object reference; a pointer to the proxy object is given to the implementation.

When used as an **[out]** parameter, the object reference is again extracted from the C++ object returned by the implementation to the EPV. On the client (calling) side, a new client proxy object is constructed in the EPV using the binding information returned in the object reference and returned to the caller.

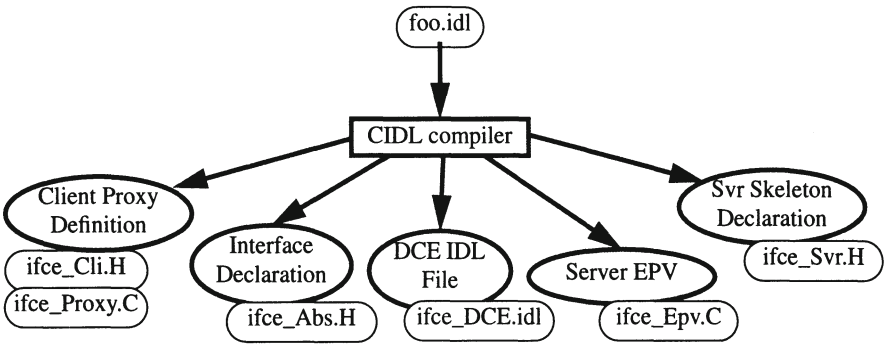

Figure 1. OMG CORBA IDL (CIDL) to OODCE compiler file generation

3.2. Files created by the compiler

The files created by the compiler are depicted in Figure 1. In the descriptions, **`ifce`** refers to the name of an interface defined in the IDL file. If multiple interfaces are defined in the OMG IDL file, multiple sets of these files will be generated.

- ifce_DCE.idl: specifies the interface to be used by the DCE IDL compiler to construct the communication stubs. The OODCE **`idl++`** compiler is then run on this file to create the interface header and stubs.
- ifce_Abs.H: the pure abstract class declaration corresponding to the OMG IDL interface. The data types and member functions of this class correspond to the types and remote operations defined in the OMG IDL file.
- ifce_Cli.H: the declaration of the proxy class used by clients to access the remote implementation. A client of a remote object invokes an operation on that object by making a member function call on this proxy object.
- ifce_Proxy.C: the definition (implementation) of the proxy object. The compiler generates an implementation class that makes an RPC into the DCE runtime to handle remote operation requests (member function calls). The proxy object implementation has code for object location and binding, setting of security preferences, object activation, and some experimental failure management and rebinding code.
- ifce_Svr.H: the declaration of the server implementation abstract class and of a default implementation class. The abstract class provides the linkage with OODCE and has pure virtual members for each of the remote operations. The default implementation class inherits from this and provides non-pure declarations for the methods the server developer must implement to provide the intended behavior of this interface. Instances of this class are registered with the DCE runtime to satisfy incoming client requests.
- ifce_Epv.C: the definition of an Entry Point Vector mapping incoming client call requests to the appropriate implementation object and method. Each client call identifies an object instance by its DCE object UUID. The functions in the EPV look for that object UUID in the OODCE object map, and make a method call on the C++ object pointer returned. The EPV supports security (the calling of a security reference monitor) and the OODCE object activation protocol.

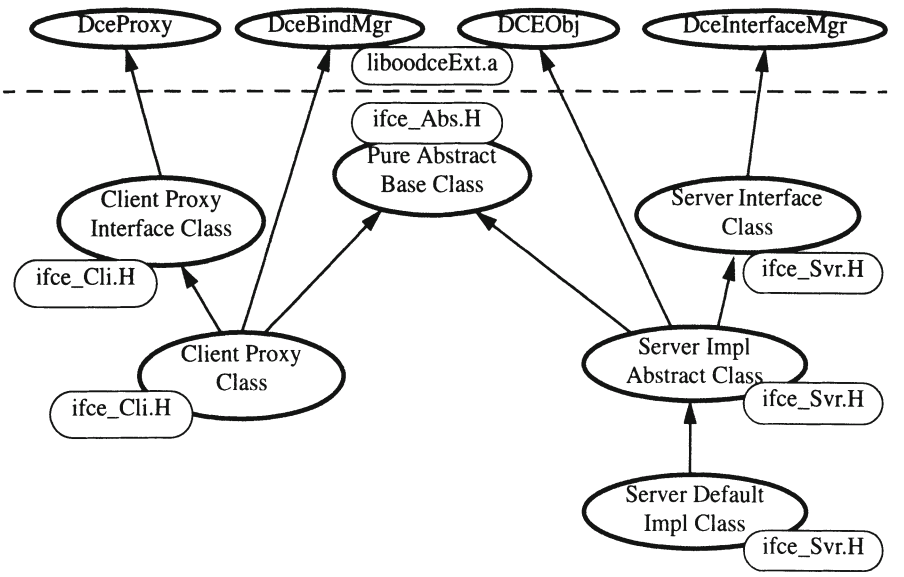

Figure 2. OMG IDL generated file inheritance hierarchy

3.3. Generated class inheritance hierarchy

The inheritance relationship among the generated classes described in the list above is shown in Figure 2. The classes on the top of the diagram (above the dashed line) are provided in the OODCE class library or in the OMG-OODCE extension library delivered with the compiler.

4. Results and Experiences

Our results from the development of this compiler indicate that it is possible to use the DCE and OODCE environment to provide an application development environment for CORBA. While the environment does not provide "100% CORBA compliance" (a term which is still not strictly defined), it does provide the ability to use OMG IDL in order to develop object-oriented distributed applications on top of an existing DCE environment. It allows use of the object-oriented features of OMG IDL–however it does not allow use of the distribution-oriented features of DCE IDL (other than interface and version identification, necessary to provide even a basic DCE application).

Since this work is based upon DCE it reduces the administrative overhead in environments where DCE is already deployed, and is a potential benefit where OMG CORBA is being considered as a future direction or where interface inheritance is beneficial. The platform described here reuses in its entirety an already developed, supported, and tested environment providing access to RPC, naming, threads, and security for the application developer. This work provides distributed application developers with OODCE's ease of use and access to C++.

4.1. Code size comparison

The following table compares code and object sizes for equivalent applications created using this compiler and using the base OODCE technology. The first set of values are for the sleeper sample application in OODCE, and the corresponding OMG-OODCE application. The second set are for the multi_if OODCE sample application, and for the equivalent OMG-OODCE application. Note that the OMG-OODCE version is using (multiple) interface inheritance, which the OODCE application can only simulate.

Table 1. Code size for OODCE and OMG-OODCE applications

OMG IDL	*DCE IDL*	*CIDL-gen'd file size*	*IDL-gen'd file size*	*Application code size*	*Total object file sizes*
—	sleep.idl	—	284	32	286Kb
Sleep.cdl	Sleep.idl	129	297	30	339Kb
	Growth (%)	n/a	5%	-6%	19%
—	multi.idl	—	1006	82	496Kb
Multi.cdl	Multi.idl	554	1284	98	602Kb
	Growth (%)	n/a	27%	20%	21%

From this we observe the following:

- The developer-written code size to implement the application is roughly the same. Since both applications are developed to the OODCE API this was expected. The difference in the multi case is due to multiple interface inheritance; in the OODCE case inheritance is simulated outside the IDL file. The OODCE code is smaller, but not as easy to understand.
- The OMG-OODCE compiler generates a lot of code. Compiling a DCE IDL file generates five files for each DCE interface. Compiling an OMG IDL file with one OMG interface similarly creates six files, including one DCE IDL file (for a total of 11 generated files). The size of the corresponding DCE IDL-generated code is also somewhat larger.
- The object files and executables are somewhat larger when using this compiler as well. This bloat is due to the extra translation layer. However, in the extra layer we are dealing with user-defined exceptions declared in OMG IDL, and have added enhanced failure recovery code.

While these results are mildly interesting they are not necessarily representative of more full-featured distributed applications. In particular, the equivalent OODCE code required to handle interface inheritance or user-defined exceptions provided by the OMG-OODCE compiler would likely bring these numbers back to parity. Alternatively, the code to make up for OMG IDL's lack of client-server context or bulk data transfer would make the OMG-OODCE case much less attractive. As always, experienced application designers will choose the tool best suited to the task at hand–be it OMG IDL, DCE IDL, Distributed Smalltalk, or Modula-3.

4.2. Related efforts

There are a number of commercial and prototype CORBA environments available currently. Some of the current CORBA implementations use a DCE infrastructure to support inter-ORB, communication. However, we are not aware of any implementations that make direct use of the DCE development model and services. While reuse of the DCE RPC marshaling engine and state machine are valuable, framework-based reuse offers greater cost savings for developers as well as for administrators.

Other efforts have focused on adding more object-oriented concepts directly to DCE IDL, most notably Digital's DCE++ [XIDL]. There are several advantages to this approach; most notably it lessens the semantic gap between the application's IDL and the distribution and implementation environments. With XIDL, anything expressible in IDL can be more naturally implemented, with fewer compile steps; and XIDL preserves application access to all of DCE's underlying facilities, including pointers, pipes, and context handles. The XIDL work is still under development and undergoing standardization. By contrast, this work uses the existing OMG IDL, but is also under development–perhaps "under developed" would be more accurate. (There are no plans in place to enhance this prototype technology or release it as a product.)

Our recommendation for distributed application developers are therefore multiple choice: use OMG IDL today with a current CORBA product; use OSF DCE today (with OODCE if possible) and XIDL tomorrow; use a framework like ACE/ACX to abstract away programming with TCP sockets and message queues; or like Conduits+ for programming atop ATM; or one of several products this author is not familiar with. The choice depends upon the project's needs and the capabilities provided by the various platforms. There is no one right answer.

5. Future Work

This compiler is currently being used in a project building a global data and service distribution application. The purpose of the application is to experiment on a large scale (over 100 sites, with several hundred users, worldwide) with system robustness, availability, and application management. The interfaces for the system are being developed in OMG IDL so that they are "future proof"–it is widely believed that OMG IDL will be a customer requirement in the future, so using it today allows us to architect and implement object interfaces that will be usable in future CORBA-based projects.

This project is experimenting with improving the reliability of distributed client/server systems through enhancements of the communication stubs. In particular, providing enhanced distribution transparency in the presence of failures in distributed components.

6. Acknowledgments

This compiler is based upon the OMG IDL CFE (compiler frontend) 1.3, provided in the public domain by SunSoft. Suggestions from Jeff Morgan drove the creation of this project, and his help in the development of the compiler is greatly appreciated. Assistance and consulting on the use of the compiler frontend, as well as source code examples from Steve Vinoski and Keith Moore have been greatly appreciated. Thanks also to Bob Fraley, Marta Kosarchyn, and Andreas Vogel for their comments, and to Peter deJong for his exuberant critiques of this work.

7. REFERENCES

[Dill95] *OODCE: A C++ Framework for the OSF Distributed Computing Environment.* John Dilley, Hewlett-Packard Laboratories. NSA-94-038. Winter '95 USENIX, January, 1995.

[OMG91-12-1]Object Management Group: *Common Object Request Broker Architecture and Specification.* Document Number 91.12.1, Revision 1.1.

[OMG94-9-32]*Universal Networked Objects.* J. Nichol, D. Curtis, D. Vines, N. Holt, O. Hurley, G. Lewis. OMG Document 94-9-32; Object Management Group, Inc. September, 1994.

[OMG94-9-14]*IDL C++ Language Mapping.* Digital Equipment Corporation, Expersoft Corporation, Hewlett-Packard Company, IONA Technologies, Ltd., International Business Machines Corporation, Novell Inc., SunSoft, Inc. OMG RFP Submission 94-9-14. September 12, 1994.

[OMG94-3-5]*Joint Submission on Interoperability and Initialization (JSII).* Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, International Business Machines Corporation, NEC Corporation, Open Software Foundation. OMG TC Document 94-3-5. March, 1994.

[OSF92] Open Software Foundation: *OSF DCE Application Environment Specification.* Open Software Foundation, 1992.

[Sun94] *README file* from the public domain Interface Definition Language Compiler Front End. Available via anonymous FTP as /ftp@omg.org:/pub/OMG_IDL_CFE_1.3/OMG_IDL_CFE_1.3.tar.Z.

[Vogel95] *Overcoming the Heterogeneity of Middleware Platforms.* Andreas Vogel, Brett Gray, Keith Duddy, CRC for Distributed Systems Technology, DSTC Pty Ltd. International Conference for Distributed Processing. February, 1996. (ICDP'96).

[XIDL] *DCE IDL With C++ Support–Functional Specification.* R. Viveney. OSF DCE RFC 48.2. October 1994.

8. BIOGRAPHY

John Dilley is a distributed systems architect with Hewlett-Packard Laboratories. His research interests include architecture and design of distributed systems and applications, object location (naming) and distributed directory services, and object-oriented design and development. He is one of the original architects of the HP OODCE product. Mr. Dilley's current research focuses on the construction and deployment of large-scale widely-distributed systems to explore issues of application availability and management on a global scale.

Mr. Dilley received Bachelor of Science degrees in Mathematics and in Computer Science from Purdue University in 1984, and a Master of Science degree in Computer Science from Purdue University in 1985.