Software Tools for Automating the Migration From DCE to CORBA

Aniruddha Gokhale and Douglas C. Schmidt
{gokhale,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, Missouri, 63130

Stanley Moyer
stanm@bellcore.com
Bellcore
445 South Street
Morristown, NJ, 07960

A subset of this paper appeared in the proceedings of ISS 97: World Telecommunications Congress, IEEE Toronto, Canada, September, 1997.

Abstract

Next-generation telecommunication software must be flexible and reusable. These requirements motivate the use of object-oriented (OO) middleware like the Common Object Request Broker Architecture (CORBA). However, many existing telecommunication software products have already been written using the Distributed Computing Environment (DCE) RPC toolkit. To reduce porting effort and to minimize unnecessary rework, it is essential to provide a smooth migration path from DCE to CORBA.

This paper provides two contributions to the study of migration strategies from DCE to CORBA. First, we describe a migration tool we developed that provides source-level interoperability between DCE RPC and CORBA. Our tool automatically translates DCE Interface Definition Language (IDL) into CORBA IDL and generates code that integrates existing DCE-based code with stubs and skeletons generated by CORBA IDL compilers. Second, we present our lessons learned applying this migration tool to a project at Bellcore.

Our experience using the tool on existing applications at Bellcore indicates that source-level interoperability provides a low-cost, yet powerful solution. In addition, the time required to develop the tool (about 9 person months) was substantially less compared to developing a full scale DCE-CIOP protocol. We also found out that minimal human intervention was necessary to achieve interoperability.

The disadvantages of using source-level interoperability arise from the fact that not all constructs in the source domain can be mapped onto the target domain. Therefore, source-level interoperability may provide solutions to only a subset of constructs of the source domain (DCE in our case). In practice, this was not a problem as long as programmers followed certain development guidelines.

Keywords: CORBA, DCE, Interoperability, Software migration tools.

1 Motivation

CORBA is an emerging standard for distributed object computing [10]. CORBA enhances application flexibility and portability by automating many common development tasks such as object location, parameter marshaling, and object activation. In general, CORBA improves upon conventional procedural RPC middleware (such as OSF DCE RPC and ONC RPC) in the following ways:

• **Support for OO language features:** such as encapsulation, interface inheritance, parameterized types, and exception handling;

• **More flexible communication models:** such as object references, which are essentially "network pointers" that support a range of communication models like peer-to-peer and distributed callbacks;

• **Dynamic invocation capabilities:** which allow applications dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in to the application.

These features enable complex applications to be developed rapidly and correctly.

An increasingly large number of distributed applications are being developed using the CORBA technology. However, a significant number of complex, distributed applications that predate CORBA use DCE. It is generally not cost-effective to reimplement these applications from scratch using CORBA. Therefore, it is essential to develop automated techniques and tools that can enable existing DCE applications to interoperate with, and incrementally migrate to, CORBA.

There are two general models for achieving interoperability between DCE and CORBA:

• **Protocol-level interoperability:** The CORBA specification defines a protocol-level interoperability standard called Environment Specific Inter-ORB Protocols (ESIOP)s. ESIOP's provide a means for CORBA to interoperate with non-CORBA applications and toolkits. The primary example of an ESIOP is the DCE Common Inter-ORB Protocol (DCE-CIOP). This protocol uses the DCE remote procedure call (RPC) mechanism to transport messages between DCE and

CORBA, whereas message formatting, marshaling of data, and operation dispatching can be performed by a CORBA Object Request Broker (ORB).

Although DCE-CIOP offers a solution that allows DCE and CORBA to interoperate seamlessly, it has the following disadvantages:

- *High cost solution* – requires an expensive third-party CIOP component, which is currently not widely available;
- *Format conversion* – requires converting between CORBA Common Data Representation (CDR) and DCE Network Data Representation (NDR);
- *DCE as transport mechanism* – uses the DCE RPC mechanism to transport of messages rather than using the ORB.

• **Source-level interoperability:** Another way to interoperate between DCE and CORBA is to use source-level interoperability. Source-level interoperability has the following benefits:

- *Low cost solution* – provides a low cost solution that is based on publicly available components;
- *Less format conversion* – the only conversions required are between the native representation and CORBA CDR (which is often a "null" conversion since CORBA CDR is a bi-canonical format that supports both big-endian and little-endian byte-orders);
- *ORB as transport mechanism* – uses the ORB for message transport. This allows interoperability with other ORBs without the need for an ESIOP such as DCE-CIOP.

The work presented in this paper uses the source-level interoperability model. We have developed a software tool that translates DCE IDL into the corresponding CORBA IDL, plus extensions for DCE IDL types that do not map directly into CORBA IDL. In addition, our tool generates code that integrates existing DCE-based code with the stubs and skeletons generated by CORBA IDL compilers.

Our DCE→CORBA IDL translation tool is based on the SunSoft CORBA IDL front-end. The "front-end" of our tool extends the SunSoft IDL compiler to accept DCE IDL syntax. The "back-end" of our tool generates CORBA IDL that corresponds to the DCE IDL provided as input. In addition, our back-end generates code that integrates existing DCE-based code with the stubs and skeletons generated by the CORBA IDL compiler.

This paper is organized as follows: Section 2 describes key similarities and differences between DCE and CORBA and summarizes the advantages and disadvantages of migrating from DCE to CORBA; Section 3 describes the design methodology of our DCE→CORBA IDL translation tool; Section 4 describes our experience using the tool; Section 5 provides concluding remarks; and Appendix A outlines how we modified the SunSoft CORBA IDL compiler to develop our migration tool.

2 Comparing and Contrasting DCE and CORBA

Both DCE and CORBA support the development and integration of applications in heterogeneous distributed environments. This section summarizes the main features of DCE and CORBA, comparing and contrasting their key similarities and differences [3].

2.1 Key DCE/CORBA Similarities

The key similarities between DCE and CORBA are outlined below:

• **Simplify common network programming tasks:** DCE and CORBA simplify common tasks of building distributed applications such as service registration, location, and activation, demultiplexing, framing and error-handling, parameter (de)marshaling, and operation dispatching.

• **Support for heterogeneous environments:** DCE and CORBA shield application developers from differences in programming languages, operating systems, computer hardware (particularly instruction byte ordering), and networking protocols.

• **Use of Interface Definition Languages (IDLs):** DCE and CORBA support the definition of service components, using high-level interface definition languages. The main purpose of an IDL is to separate interface from implementation. This separation of concerns makes it possible to (1) improve the modularity and specification of software components, (2) transparently distribute implementation across process and host boundaries, (3) write language-independent applications, and (4) remove common sources of network programming errors.

• **Automatically generated stubs and skeletons:** Implementations of DCE and CORBA provide IDL compilers that automatically translate IDL definitions into client-side stubs and server-side skeletons. Stubs are *proxies* [5] that interact with the underlying run-time systems to allow clients to access services defined by servers. Skeletons integrate application-specific code with automatically generated code that performs demarshaling, demultiplexing, and dispatching of client requests to target object implementations.

• **Synchronous request/response communication:** Both DCE and CORBA support synchronous request/response communication. In this approach, the client calls an operation on the server. The client blocks until the server completes the operation, at which point `out` or `inout` parameters and/or a return value is passed back to the client. In theory, synchronous request/response communication helps shield client applications from knowledge of whether the target object implementation is local or remote. In practice, it is often difficult to completely hide the use of distribution from clients due to differences in performance and reliability [2].

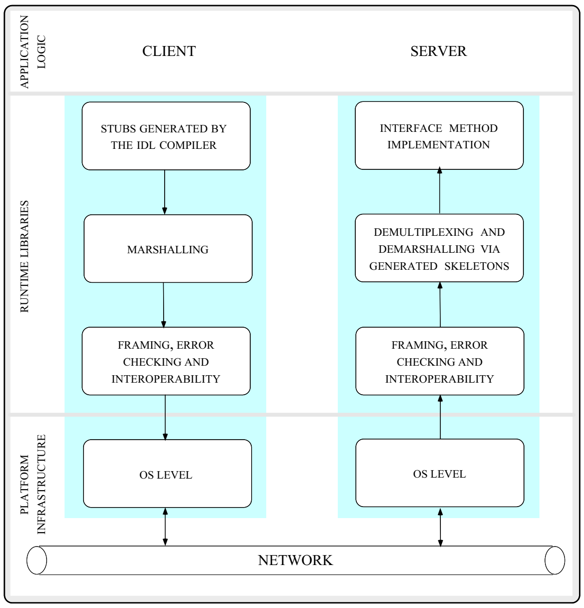

Figure 1: General Path of CORBA and DCE Requests

• **Oneway communication:** CORBA supports "oneway" (send-only) calls, where the server does not return any information to the client (*e.g.,* as part of the operation's return value or inout/out parameters). In DCE, oneway operations can be achieved using "maybe" semantics, which are a special case of DCE "idempotent" operations.

• **Similar request path:** Figure 1 shows the general path that CORBA and DCE implementations use to transmit requests from client to server for remote operation invocations. The client code invokes the IDL compiler-generated stubs to access the services of the server. After the client invokes the stub, it blocks until it receives a response from the server.

The presentation layer encodes the request data into a common data representation. The run-time system then packetizes the encoded data by adding framing information. This may include packet headers and trailers, checksums for data integrity, encrypted data for security and information for interoperability. The run-time system uses the underlying network software provided by the operating system and device drivers to send the packets to the destination.

On arrival at the server, the network software passes the request to the run-time system. The run-time system removes the framing information and passes the request to the presentation conversion layer. This layer converts the encoded data into the native format of the host machine (if necessary) and passes it over to the message demultiplexer. The demultiplexer dispatches the request to the appropriate server stub generated by the IDL compiler.

The response traces the reverse path of the incoming request message through the server and client. When the response reaches the client, it unblocks the client stub that is waiting for the reply.

• **Higher-level services:** Both DCE and CORBA build upon their core communication infrastructure (called the "executive" in DCE and the "ORB" in CORBA) to provide higher-level distributed services. Common services provided by both CORBA and DCE include a time service, event service, and naming and directory services.

2.2 Key DCE/CORBA Differences

The following describes the key differences between DCE and CORBA:

• **Programming model:** An important difference between DCE and CORBA is that DCE was designed to provide a *procedural* programming model, whereas CORBA was designed to provide an *object-oriented* (OO) programming model. This difference is analogous to the difference between the C and C++/Java programming models.

For instance, it is possible to implement OO programs using C by manually creating virtual tables and other OO language features. However, the effort required to do so is high and the results are often error-prone and complex. In contrast, the effort required to implement OO programs with C++/Java is typically much lower since these languages support OO features directly.

Note that the difference between DCE's procedural programming model and CORBA's OO programming model is often overstated. In particular, there are extensions to DCE that provide an OO veneer (such as OODCE [4] and DCE Objects [1]). There are, however, a number of restrictions inherent in using DCE in an OO manner. To illustrate these, consider the following ways in which the DCE and CORBA programming models differ:

- *Support for multiple inheritance of interfaces and polymorphism* – CORBA provides these features to support the specialization and reuse of existing interfaces. Developers can use inheritance to form new composite interfaces, which can be implemented flexibly using polymorphism.

  In contrast, DCE does not directly support interface inheritance or polymorphism of implementation, which requires tedious manual recoding of common interfaces and operations.

- *Accessing distributed resources via Object References* – In CORBA, Object References are "first class" entities that can be passed to clients throughout a network and used to flexibly access server objects. DCE does not provide this degree of flexibility without additional programming effort.

- *Object-style vs. RPC-style communication* – Figure 2 illustrates the difference between RPC-style communication (supported by DCE) and Object-style communication (supported by CORBA) [15].[1] There are several benefits to using Object-style communication:

[1] Note that CORBA can also be used in a manner that supports RPC-style communication.

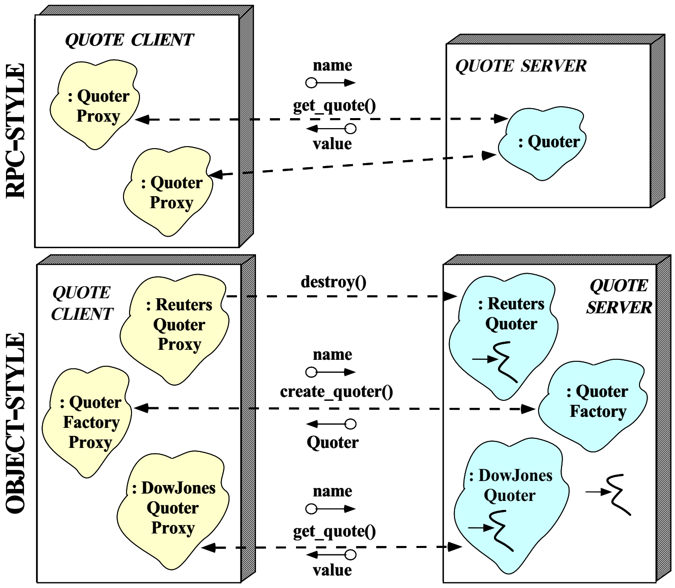

Figure 2: RPC-style vs. Object-style Communication

- *Customized quality of service* – Clients can use factories to create different types of product objects that support a range of functionality or performance characteristics (such as real-time quality of service or high-bandwidth) that are tailored to their individual needs.
- *Flexible lifecycle control* – Object-style communication gives clients more flexibility to control the lifecycle of object implementations, compared with RPC-style communication. For instance, servers accessed via RPC-style interfaces often must make inefficient or non-robust assumptions about the lifecycle of clients that access their services.

• *Ability to associate cohesive operations into modular and reusable components* – CORBA's programming model encourages the association of related operations to form modular and reusable components. Although it is possible to achieve much the same effect in DCE via developer conventions, the standard DCE programming model is not as conducive to supporting OO design and implementation.

• **Communication model:** CORBA supports the "deferred synchronous" communication, which separates a send operation from a server's reply. DCE does not support deferred synchronous operations, though it is possible to approximate this to some extent using multiple threads. However, DCE supports the notion of "idempotent" operations, which can be used to optimize duplicate detection on a server. CORBA does not support idempotent operations.

• **Interface Definition Languages (IDLs):** CORBA IDL is designed to allow interoperability between a range of target languages (such as C, C++, Smalltalk, Java, Ada, and COBOL). In contrast, DCE IDL is focused primarily on C (and C++ to the extent that the type system of C is a subset of C++).

• **Type systems:** An important consequence of CORBA's emphasis on language independence is that its type system is simpler (though inherently more restrictive) than DCE. In particular, there is no support in CORBA for passing pointers, arrays of varying sizes, and streaming data.

In particular, there is no direct support in CORBA for the following type system features:

- *Passing pointers or structures containing pointers*
- *Streaming data – e.g.,* via the DCE `pipe` mechanism.
- *Conformant arrays – i.e.,* arrays of varying sizes.

Unlike DCE, on the other hand, CORBA support the "any" type, which allows clients and servers to pass arbitrary data values whose type is determined at run-time.

• **Dynamic invocation:** CORBA supports the dynamic invocation of requests that can be created and called at runtime. The correctness of these requests can be checked at run-time using CORBA's Interface Repository. In contrast, DCE does not provide support for dynamic invocation or interface repositories.

• **Interface and Implementation Repository:** CORBA's Interface Repository stores information present in the IDL files. Applications can query an Interface Repository for information about interfaces they plan to utilize. This feature is useful for tools such as browsers and debuggers that have no prior knowledge of the interfaces offered by a server. By querying the Interface Repository, applications can dynamically access services offered by different servers and construct requests at run-time.

In addition, CORBA defines an Implementation Repository that ORBs use to map client requests to object implementations. Implementation Repositories contain information that ORBs use to locate and activate object implementations. DCE does not define an Interface or Implementation Repository explicitly in the standard specification.

• **Component identity:** In DCE, all components (*e.g.,* IDL definitions, IDL implementations, servers, etc.) are identified by "Universal Unique Identifiers" (UUIDs). CORBA has no notion of a UUID. Instead, components in CORBA are "identified" via Object References, which grant applications access to CORBA objects, but provide no guarantees of unique identity. For more information on the pros and cons of this issue see [11] and [8], respectively.

• **Infrastructure services:** DCE defines a multi-threading API that is part of its core "executive" infrastructure. In contrast, CORBA does not define a standard API for multithreading. Therefore, it is not possible to write portable CORBA multi-threaded applications. Likewise, DCE defines a security service (based on Access Control Lists) in its core infrastructure, whereas CORBA defines security as a higher-level service.[2]

• **Higher-level services:** Higher-level services defined by DCE and CORBA are different. For instance, DCE defines a distributed file service (DFS) in its *extended services* component, whereas CORBA does not provide this service. On the other hand, the OMG has completed specifications for a much wider range of distributed services, *e.g., Trading Service*. including *Concurrency Control*, *Event Service*, *Externalization Service*, *Life Cycle Services*, *Naming Service*, *Persistent Object Service*, *Query Service*, *Relationship Service*, *Transaction Service*, *Licensing Service*, *Property Service*, *Security Service*, *Time Service*, *Trading Service*.
In addition, the OMG is currently defining standards for even higher-level, application-specific services, known as *CORBA facilities*. These facilities will cover domains such as user-interface, compound documents, and task management. However, the CORBA services and facilities are not yet well defined, nor widely implemented.

• **Interoperability and portability:** The DCE specification and its various implementations were designed *a priori* to be interoperable and portable. In contrast, the original CORBA 1.x specification did not ensure interoperability and portability of CORBA implementations. Although CORBA 2.0 addresses this weakness, many CORBA implementations do not yet implement interoperability robustly [7].

In addition, there are a number of non-portable aspects of the CORBA server-side Object Adapter specification, including:

- *Non-portable mapping of skeletons onto implementations* – There is no standard way to map the automatically generated IDL skeletons onto application-specific target object operation implementations.
- *Incomplete Object Adapter Interface* – The existing interface for the Basic Object Adapter in the CORBA 2.0 standard is very incomplete. Therefore, each ORB vendor has added non-standard features to make it possible to utilize important OS platform resources (such as threads or dynamic linking).

This lack of specificity in the CORBA 2.0 specification makes it hard to develop completely portable server implementations. However, the client-side CORBA 2.0 specification does support the development of relative portable clients.

• **Context:** The notion of context in DCE and CORBA is different. Contexts in DCE are used to maintain server states during a series of logically related requests from a single client. The run-time system understands the information stored in the contexts. DCE contexts in a distributed application is analogous to a file handle in a local application. These contexts are maintained by the stubs and the RPC run-time libraries and not by the application code. In contrast, CORBA contexts are opaque to the run-time system. They are used to carry user information along with the request and are similar to UNIX "environment variables." Programmers responsible for managing and interpreting CORBA context information.

Category	Sub Category	CORBA	DCE
Programming Aspect	Model	Object-oriented	Procedural
	Interface inheritance	supported (including mult. inheritance)	not supported
	IDL design	interoperability between many languages	only for C (C++ minimal)
Communication Aspect	Model	object style	RPC style
	Idempotent ops	not supported	supported
	Component Identification	object references	unique ids (UUID)
	Dynamic Invocation	supported	not supported
	Deferred Synchronous	supported	not supported
	Repository (impl and i/f)	supported	not supported
	Interoperability	optional (UNO specs)	required
	contexts	opaque to run-time system	used solely by run-time system
Services	Infrastructure	no thread API	built in thread API
	security	external	built-in
Market Force	Vendor support	active, large	restricted
	Customer base	large	restricted

Table 1: Summary of Key Differences between DCE and CORBA

2.3 Transition Strategies from DCE to CORBA

It is hard to port from DCE to CORBA since many features do not map directly. Therefore, to achieve some degree of portability to transition from DCE to CORBA it is necessary to avoid certain DCE features. In this context, portability focuses on writing DCE applications that may some day need to port to CORBA.[3] The following guidelines are intended to make it easier to port DCE code to CORBA:

- Avoid using DCE ACF attributes (*e.g.,* `comm_status` and `fault_status`). Instead, use DCE exceptions since they map better onto CORBA exceptions. Likewise, don't use the "implicit" form of binding handles since this is hard to port to CORBA (which utilizes explicit Object References).
- Do not rely on the fact that DCE is case significant (CORBA is case *insignificant*).

[2] Note that the DCE security model has been available for many years, whereas the CORBA security model is relatively not widely implemented yet.

[3] A different (and somewhat easier) set of guidelines are required to go from CORBA to DCE.

- Avoid the use of DCE pipes since CORBA doesn't provide an equivalent mechanism (yet).[4] To implement this feature in CORBA requires the use of either (1) external mechanisms (*e.g.,* ACE wrappers for socket streams) or (2) defining IDL classes for streaming, which incurs very high overhead on many existing ORBs [14, 12, 6].
- Avoid the use of pointers in DCE interfaces since CORBA doesn't provide a mechanism for specifying this in a portable and transparent manner. To pass "pointers" in CORBA requires additional explicit marshaling code and/or the use of CORBA IDL structures. If pointers are being used for optional values (as was the case in some DCE software at Bellcore) there are ways of working around this in CORBA (*e.g.,* defining an IDL union that has "valid" and "NULL" enum tags or such).
- Avoid the use of DCE contexts and context handles since CORBA doesn't support these transparently (in CORBA, the programmer is responsible for managing this type of context information explicitly).
- Be careful of using DCE "varying arrays," which only pass a part of the array from client to server. CORBA doesn't support this directly, though similar behavior can be obtained using CORBA sequences.
- In general, to achieve interoperability, developers should focus on defining service interfaces and semantics first. Only once this is defined should they attempt to implement that behavior, which should minimize reliance on non-portable DCE or CORBA features.

The key differences between DCE and CORBA are summarized in Table 1.

3 Achieving Source-level Interoperability between DCE and CORBA

This section describes the design of our DCE→CORBA migration tool. Our migration tool handles most DCE IDL constructs, including:

- Separate IDL files used for constants, struct declarations, and interfaces;
- Simple constant declarations;
- Simple DCE IDL types;
- `enums`;
- Varying arrays within a structure;
- Full pointers used in structures and in operations for optional parameters;
- Simple structures;
- Complex structures;
- Operations returning `error_status_t`;
- Operations using an explicit binding handle;
- Reference pointers used in `out` parameters;
- Operations having `in` parameters;
- Operations having `out` parameters of type array.

[4] The OMG Telecom SIG is currently working on standardizing a similar mechanism.

Our tool does not handle the following DCE IDL constructs because they have no equivalent in CORBA IDL:

- Pipes;
- Complex constant declarations involving relational and logical operators;
- Idempotent operations.

In addition, our tool did not handle the following DCE IDL constructs since they were not used by the domain of applications for which the tool was designed.

- unions;
- hyper integers;
- conformant arrays.

3.1 Mapping DCE IDL to CORBA IDL

[16, 17] describe techniques to map a number of DCE IDL constructs to CORBA IDL. Our tool uses many of these techniques to map DCE IDL to CORBA IDL. In addition, we devised other techniques for our tool to handle a wider range of cases than described in [16, 17]. All these techniques are outlined below.

- **Import Statements:** A DCE `import` statement is shown below:

```
interface example{
    import "default_constants.idl";
}
```

The corresponding CORBA IDL declaration uses the `#include` preprocessor primitive to include the IDL files as shown below:

```
#include <default_constants.idl>
interface example {
}
```

The `#include` statements are left outside the `interface` definition since the included file may contain an interface definition. Including such a file inside an interface causes nested interface definitions, which are not supported in CORBA. [16, 17] describe a technique where the IDL files are included using interface inheritance.

• **Constants:** The DCE to CORBA mapping for `constants` is straightforward. A DCE `constant` declaration is shown below:

```
interface default_constants{
   const unsigned short FIX_DATE=0;
}
```

This declaration maps to the following CORBA declaration:

```
interface default_constants{
   const unsigned short FIX_DATE=0;
}
```

DCE allows constant declarations with expressions involving logical and relational operators. There is no equivalent CORBA IDL support for these cases. An example involving these operators in DCE constant declaration is shown below:

```
/* assume x, y, z are predefined */
const unsigned short XYZ=x < y ? x : y;
const long PQR= ((x < y) && (y > 5));
```

Whenever the mapping tool encounters such non portable constructs, it flags an error. In this respect, the tool behaves similar to the `lint` C program checker.

• **Enumerations:** An enum declaration in DCE is shown below:

```
typedef enum {
        TRACE_0,
        TRACE_L,
        TRACE_M,
        TRACE_H,
        TRACE_A
} TraceLevel_e;
```

This is transformed to the following declaration in CORBA:

```
enum _TraceLevel_e{
        TRACE_0,
        TRACE_L,
        TRACE_M,
        TRACE_H,
        TRACE_A
};
typedef _TraceLevel_e TraceLevel_e;
```

• **Fixed Length Strings:** In DCE, the `string` attribute is applicable to the `char` and `byte` data types. In addition, the `string` attribute can be applied to a `struct` only if all its members are of type `byte`. In DCE IDL, all characters in an array including the `NUL` character will be transmitted, unless the `string` attribute is specified. The `NUL` character helps in terminating the array.

In contrast, CORBA provides a `string` data type, which is represented as a sequence of `chars`. Hence, we provide the following two different mappings for `string` of `chars` and `string` of `bytes`:

- *Strings of characters* – A DCE declaration for a `string` of `chars` is shown below:

  ```
  typedef [string] char MsgType_str[MAX_MSG_TYPE+1];
  ```

  The corresponding CORBA mapping is shown below:[5]

  ```
  typedef string <MAX_MSG_TYPE+1 - 1> MsgType_str;
  ```

- *Strings of bytes* – A DCE declaration for a `string` of `bytes` is shown below:

  ```
  typedef [string] byte MsgType_str[MAX_MSG_TYPE+1];
  ```

  The corresponding CORBA mapping is as shown below:

  ```
  typedef sequence <octet, MAX_MSG_TYPE+1 - 1>
          MsgType_str;
  ```

  The CORBA mapping subtracts a 1 from the length specified in the corresponding DCE definition. This is based on the assumption that the DCE declaration uses one extra space to accommodate the `NUL` character.

  The CORBA `octet` and DCE `byte` data types are equivalent. They represent untyped or opaque data. They do not undergo any marshaling/unmarshaling.

• **Unbounded Strings:** Unbounded strings of characters are mapped to CORBA unbounded strings. An example is shown below:

```
/* DCE unbounded string */
typedef [string] char *v_string;

// CORBA Mapping
typedef string v_string;
```

• **Simple and Complex Structures:** The CORBA mapping for simple and complex DCE structures is straightforward. A DCE `struct` declaration is shown below:

```
typedef struct MsgOutput {
     MsgType_str type;
     MsgCode_str code;
     MsgDest_str dest;
     MsgText_str text;
} MsgOutput_t;
```

The equivalent CORBA mapping is shown below:

```
struct MsgOutput {
     MsgType_str type;
     MsgCode_str code;
     MsgDest_str dest;
     MsgText_str text;
};
typedef MsgOutput MsgOutput_t;
```

• **Varying Arrays within a Structure:** A DCE declaration for varying arrays handled by our tool is shown below:

```
typedef struct infoList {
    short first;
    short len;
    [first_is(first), length_is(len)] info_t
      info_arr[MAX_ARRAY_LEN];
} info_array;
```

[5] The assumption here is that the DCE `string` declaration always allocates one extra byte to hold the `NUL` character.

The CORBA IDL does not directly support the `varying array` concept. Hence, our CORBA mapping uses `bounded sequences`, along with some additional information that keeps track of the lower and upper bounds of the array being transmitted. [16, 17] suggest a similar approach, but their prototype uses CORBA constant arrays.

```
typedef sequence <info_t, MAX_ARRAY_LEN> seq_info_t;
typedef struct info_array{
    short first;
    short len;
    struct varying_array{
        short lower;
        short upper;
        long length;
        seq_info_t var_arr;
    } info_arr;
};
```

The `seq_info_t` sequence in the above declaration defines a bounded sequence of `info_t` with a maximum size of `MAX_ARRAY_LEN` as specified in the DCE declaration. The lower and upper bound of the part of the array being transmitted is stored in the variables info_array::varying_array::lower and info_array::varying_array::upper. Our translation tool generates external helper functions to copy the incoming sequence into a pre-allocated array starting at the location indicated by the lower bound value.

• **IN, OUT and IN,OUT Parameters:** The DCE IDL to CORBA IDL mapping for the `in`, `out` and `in,out` attributes is straightforward. An example for the mapping of `in`, `out` and `in,out` is given below.

```
/* DCE declaration */
error_status_t getInfo(
        [in] handle_t handle,
        ...,
        [out,ref] ReqStat_i *reqStatus,
        ...);

// CORBA mapping
error_status_t getInfo(
        in handle_t handle,
        ...,
        out ReqStat_i reqStatus,
        ...);

/* The DCE [in,out] attribute will be
   mapped to the CORBA inout attribute */
```

• **Full Pointers:** The domain of DCE applications we considered uses DCE "full pointers" in structures and in operations to specify optional parameters. Since these pointers can take `NULL` values and do not represent linked lists, full pointers can be mapped to a CORBA `sequence` of size one [16, 17].

A DCE full pointer used in a structure is shown below:

```
typedef struct info {
   ID_str   id;
   long              *quantity;
   OrderNum_str      *orderNum;
   Number            number;
}info_t;
```

The corresponding CORBA mapping is shown below.

```
typedef sequence<long,1> long_seq;
typedef sequence<OrderNum_str,1> OrderNum_str_seq;

typedef struct info {
   ID_str   id;
   long_seq         quantity;
   OrderNum_str_seq orderNum;
   Number           number;
};

typedef info info_t;
```

In the CORBA mapping shown above, the sequence's length will be zero if the pointer value is NULL. A sequence length of one indicates presence of data. In CORBA IDL, if any member of a `struct` is of a sequence type, then that sequence type must be typedef'd before the struct definition.

• **Reference Pointers:** Since DCE reference pointers cannot assume a NULL value, a DCE declaration of a reference pointer can be mapped to a CORBA type declaration [16, 17]. An example of this mapping is shown below.

```
/* DCE Ref Ptr declaration */
typedef [ref] short *xyz;

// CORBA mapping
typedef short xyz;
```

• **error_status_t:** The DCE methods returning `error_status_t` are mapped to CORBA methods returning `unsigned long` [16, 17]. Thus, the CORBA mapping can `typedef error_status_t` to `unsigned long` and use it as the return types for all operations that return `error_status_t`.

• **comm_status and fault_status Status Messages:** The DCE `comm_status` and `fault_status` codes can be mapped to the CORBA user-defined exceptions. We cannot rely on the CORBA::Status return value since according to the CORBA specification, ORB implementors are free to typedef CORBA::Status to void. User defined exceptions can be defined for all the different communication failures and can be raised when one occurs.

• **Binding handle mapping:** Binding handles in DCE are mapped to CORBA object references and client stub proxies.

• **Linked Lists:** A full pointer in DCE could be used to point to a linked list of nodes. The CORBA mapping for linked lists or binary trees is provided in [16, 17].

The CORBA mapping for a DCE IDL linked list is shown below.

```
/* DCE declaration of a node used in a linked list */
typedef struct ObjectList {
    short               objectID;
    ObjectClass         objectClass;
    struct ObjectList*  next;
} ObjectList;

// CORBA Mapping
struct ObjectList {
    short               objectID;
    ObjectClass         objectClass;
    sequence <ObjectList, 1> next;
};
```

• **Unions:** The DCE `union` construct is mapped to the CORBA `union` construct. An example of the DCE `union` from the MediaVantage products is shown below:

```
typedef union switch(ObjectClass retType)
   PtrUnion {
      case NODE: NodeType *nodep;
      case LINK: LinkType *linkp;
      default:   xbbMsgOutput_t *error;
   } AnyPtr;
```

The CORBA mapping for DCE `union` is given below:

```
typedef sequence<NodeType,1> NodeType_seq;
typedef sequence<Link,Type,1> LinkType_seq;
typedef sequence<xbbMsgOutput_t,1>
        xbbMsgOutput_t_seq;

union PtrUnion
  switch (ObjectClass)
  {
     case NODE: NodeType_seq nodep;
     case LINK: Linktype_seq linkp;
     default:   xbbMsgOutput_t_seq error;
  };

typedef PtrUnion AnyPtr;
```

• **Conformant Arrays:** DCE conformant arrays are mapped to CORBA unbounded sequences [16, 17]. An example of a conformant array declaration in DCE and its CORBA mapping is shown below:

```
/* DCE Conformant array */
typedef unsigned long perf_data_t[0..*];

// CORBA Mapping
typedef sequence<unsigned long> perf_data_t;
```

• **Maybe Operations:** DCE `maybe` operations are mapped to CORBA `oneway` methods.

• **Hyper Integers:** Hyper integers in DCE range from $-2^{63} to + 2^{63} - 1$. CORBA does not support hyper or unsigned hyper types. The CORBA mapping for hyper is shown below:

```
struct hyper{
    long high;
    long low;
};
```

• **Character Sets:** DCE permits the use of three different character sets:

- ISO Latin-1;
- ISO Multilingual;
- ISO UCS.

CORBA only supports the ISO Latin-1 character set. Using a different character set will require external marshaling/demarshaling routines.

• **Pipes:** CORBA does not support the notion of `pipe` operations. To support pipes, a mechanism (such as "Blob" Streaming framework in [12]) that integrates CORBA with TCP stream sockets could be used.

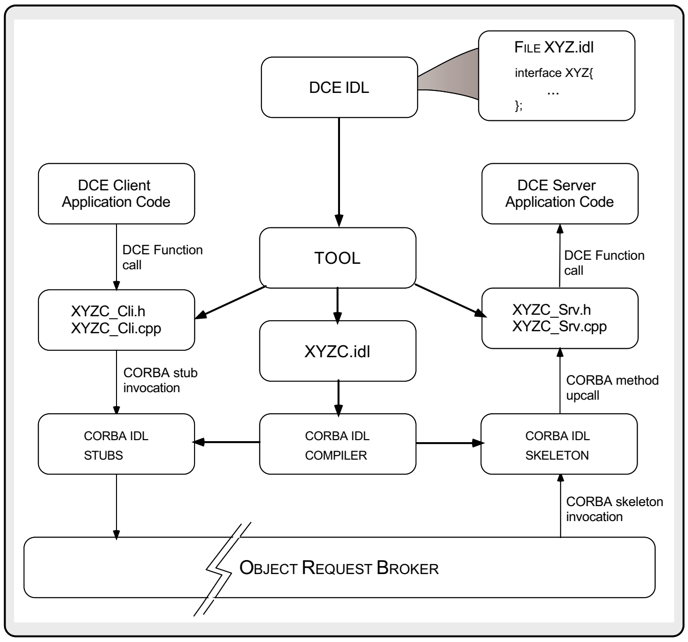

Figure 3: Interfacing Generated Code with DCE Application Code

3.2 Client-side and Server-side Adapters

This section describes the client-side and server-side interface code generated by our migration tool. Figure 3 illustrates the various files generated by the tool and shows how they are used to integrate existing DCE application code with CORBA stubs and skeletons.

For a given DCE IDL file, *e.g.,* "customer.idl," defining an interface called "customer," the following files are generated by our migration tool:

• **customerC.idl:** This file is the CORBA IDL mapping of the input DCE IDL that uses the rules discussed in Section 3.1. A "C" is appended to the file name to distinguish it from the original DCE IDL file.

The tool generates code that uses fully scoped names for all types defined in the CORBA IDL file. Fully scoped names for types are required as these types may be defined in other included CORBA IDL files. In addition, all types and definitions have a "C" appended to them to avoid name clashes with existing DCE code.

Each imported DCE file is parsed and included. If any imported file itself imports another file, the tool recursively parses the imported file and generates fully scoped names.

The tool generates "#ifndef" preprocessor commands around each included file that serve to prevent multiple inclusions of included files by the CORBA IDL compiler. A special predefined header file, "dce_corba.idl", is always included to predefine certain DCE types (such as `error_status_t` and `comm_status_t`).

• **customerC_Cli.h:** This header file defines a C++ class ("class customer" in this example) used by a DCE client application.

The C++ class maintains a private data member that stores the object reference to the server object. The public member functions have the same name and parameters as the original DCE IDL.

• **customerC_Cli.cpp:** Excerpts from the implementation of "class customer" generated by the tool are shown below.

```
// These files are also generated
// by the migration tool.
#include "common_structsC_Cli.h"
#include "exampleC_Cli.h"
#include ``customerC_Cli.h"
// Constructor
customer::customer()
{
  // Obtain object reference to server
  this->customerCvar = customerC__bind
    (/* obj name, host */);
}
// destructor releases the object reference
// (not shown).

// method getInfo
error_status_t customer::getInfo
  (handle_t handle, OrderNum_str orderNum, ...)
// other parameters not shown
{
  handle_tC handleC =
    CORBA_handle_t_adapter(handle);
  exampleC_OrderNum_strC orderNumC =
    CORBA_OrderNum_str_adapter(orderNum);
  // Other parameters translated in
  // the same manner.
  return this->customerCvar->getInfoC
               (handleC, orderNumC, ...);
}
```

The implementation file first includes the generated header files for all imported files. Next, it defines the constructor and the destructor. The constructor is responsible for acquiring an object reference to a server object. Within this constructor, manual intervention is required to set the "object name" and "host" parameters to the `_bind` call.[6]

The implementation of each method converts its parameters passed by the DCE application code into a form suitable for passing to the CORBA stubs. Finally, each method calls the CORBA stub corresponding to that method.

In this way, "class customer" serves as an interface for DCE application code to use the ORB to transport requests to the server.

• **customerC_Srv.h:** This is the server-side header file generated by the tool. It defines a C++ class that implements the interface defined in the generated CORBA IDL. Thus, in our example, the tool generates a class called "customerC_impl". The header file includes all the server-side header files corresponding to files included by the CORBA IDL file.

• **customerC_Srv.cpp:** On the server-side, the tool generates the implementation of each method defined by the CORBA IDL interface. Excerpts from the file "customerC_Srv.cpp" are shown below.

[6] `_bind` is not a standard CORBA feature. It is used here to obtain an object reference.

```
#include "common_structsC_Srv.h"
#include "exampleC_Srv.h"
#include "customerC_Srv.h"

error_status_tC
customerC_impl::getInfoC
  (handle_tC handleC,
   const char *orderNumC, ...)
{
  handle_t handle =
    DCE_handle_t_adapter(handleC);
  OrderNum_str orderNum =
    DCE_OrderNum_str_adapter(orderNumC);
  // similar transformation for other parameters
  return getInfo(handle, orderNum, id,
                 nums, trace, reqStatus,
                 msgOutput, info);
}
```

The implementation of each method converts all its parameters into a form understood by the DCE application code on the server. After each parameter is converted, the method invokes the appropriate DCE method in the server. Thus, the server-side generated class serves as an interface for the ORB to delegate incoming requests to the DCE application code.

4 Experience

The DCE→CORBA migration tool has been used at Bellcore on a large distributed software application.[7] The server-side of the application ran on an HP/UX workstation and the client ran on a Windows NT-based PC. The server offered several DCE interfaces that the client uses to request services. The files that define these interfaces utilize the following features of DCE IDL:

- import
- enums
- constants
- fixed-length strings of characters
- simple and complex structures
- full and reference pointers
- linked lists
- conformant arrays

As evident by the list of features used, these DCE IDL files were very complex. The DCE→CORBA migration tool was run on all the DCE IDL files to generate CORBA IDL files and the appropriate DCE to CORBA (and CORBA to DCE) stubs. The migration tool was able to generate valid CORBA IDL (and stubs) for all of the DCE IDL files.

However, the CORBA IDL compiler that we used to compile the resultant CORBA IDL (into C++ files) did not generate valid code – the code generated C++ compilation errors.[8] We manually modified the CORBA IDL files gen-

[7] The DCE→CORBA migration was done as a research experiment and does not necessarily mean that Bellcore is (or is not) moving their products from DCE to CORBA.

[8] We tried a different CORBA IDL compiler and it generated valid C++ code.

erated by the DCE→CORBA migration tool so that the CORBA IDL compiler generated valid C++ code. This resulted in a client/server DCE application that communicated via CORBA.

The migration tool was invaluable since it significantly reduced the amount of code that had to be modified manually. The primary modification that we made was to change the calls to the DCE memory allocation routines. The main cost in using the tool (vs. overhauling the application to use CORBA instead of DCE) is that a run-time performance penalty is incurred by the adapter methods in the stubs and skeletons that converted DCE to CORBA and CORBA to DCE. While we did not measure this delay, we believe that it should be insignificant when compared to the cost of a remote procedure call or invoking a remote method.

Another drawback in moving from DCE to CORBA was that we were not able to duplicate the security of our DCE-based application in CORBA. This is not a fault of the tool, per se, but rather a shortcoming of current CORBA products.

5 Concluding Remarks

Currently, there is significant momentum in the telecommunication industry to move to CORBA-based middleware solutions [13, 9]. However, many existing distributed applications are developed with DCE. It is not cost-effective to reimplement these applications from scratch using CORBA. Therefore, it is essential to develop automated techniques and tools that can enable existing DCE applications to interoperate with, and incrementally migrate to, CORBA.

This paper describes a tool we developed to automate the migration from DCE to CORBA. The design of this tool is based on the principle of achieving source-level interoperability, as opposed to protocol-level interoperability (*e.g.,* as specified by DCE-CIOP). Our experience using the tool on existing applications at Bellcore indicated that tools based on source-level interoperability provide a low-cost, yet powerful solution. In addition, the time required to develop the tool (about 9 person months) was substantially less compared to developing a full scale DCE-CIOP protocol.

The disadvantages of using source-level interoperability arise from the fact that not all constructs in the source domain can be mapped onto the target domain. For the DCE to CORBA, these include `pipes`, complex constant declarations involving relational and logical operators, and `idempotent` operations. Therefore, source-level interoperability may provide solutions to only a subset of constructs of the source domain (DCE in our case). Fortunately, this subset was sufficient to satisfy the requirements of the application base at Bellcore.

References

[1] Bellcore. *dceObjects Developer's Guide*, Bellcore Document BD-DCEO-DG-R140-001 edition, November 1995.

[2] Kenneth Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, 1994.

[3] Thomas J. Brando. Comparing DCE and CORBA. Technical Report MP 95B-93, MITRE, March 1995. URL : http://www.mitre.org/research/domis/reports/DCEvCORBA.html.

[4] John Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. In *Proceedings of the Winter Usenix Conference*. USENIX Association, January 1995.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[7] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997. IEEE.

[8] William Harrison. *The Importance of Using Object References as Identifiers of Objects: Comparison of CORBA Object*. IBM, OMG Document 94-06-12 edition, June 1994.

[9] Silvano Maffeis and Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[11] Michael L. Powell. *Objects, References, Identifiers, and Equality White Paper*. SunSoft, Inc., OMG Document 93-07-05 edition, July 1993.

[12] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. In *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996. USENIX.

[13] Douglas C. Schmidt, Aniruddha Gokhale, Tim Harrison, and Guru Parulkar. A High-Performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[14] Douglas C. Schmidt, Timothy H. Harrison, and Ehab Al-Shaer. Object-Oriented Components for High-speed Network Programming. In *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995. USENIX.

[15] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[16] Andreas Vogel and Brett Gray. Translating DCE IDL in OMG IDL and vice versa. Technical Report 22, CRC for Distributed Systems Technology, 1995.

[17] Andreas Vogel, Brett Gray, and Keith Duddy. Understanding any IDL-Lesson one: DCE and CORBA. In P. Honeyman, editor, *Proceedings of Second International Workshop on Services in Distributed and Networked Environments*, Los Alamitos, CA, 1996. IEEE Computer Society Press. In Press.

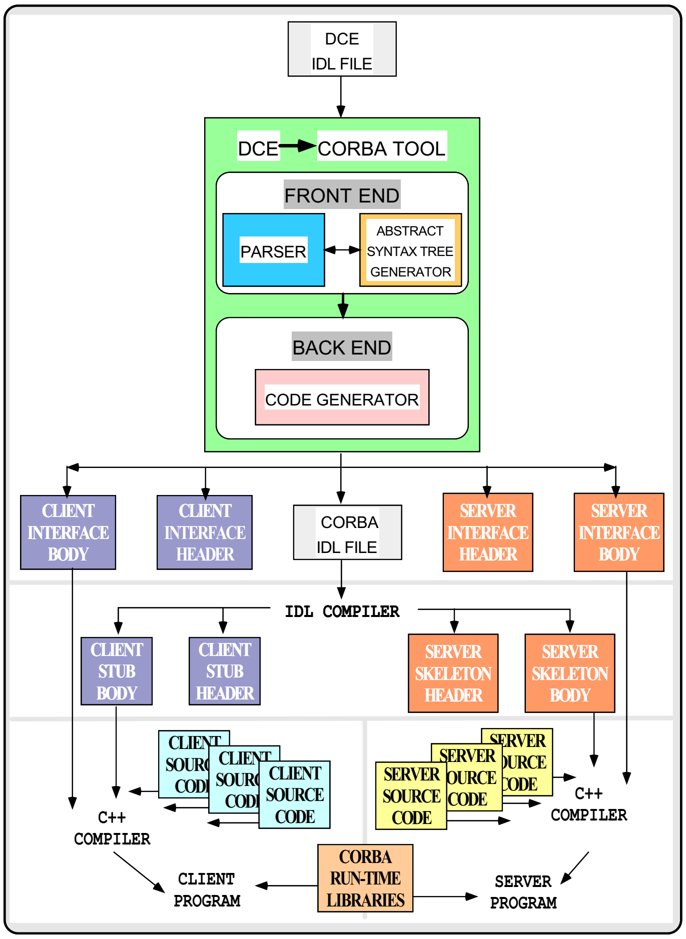

Figure 4: Design and Use of the DCE→CORBA Migration Tool

A Components in the DCE→CORBA Translation Tool

This section describes the architecture of the tool. First, we describe the architecture of the public domain CORBA IDL compiler front-end. Next, we describe our modifications to the front-end and the back-end that is responsible for the code generation.

A.1 SunSoft's CORBA IDL Compiler Front-end

The SunSoft's CORBA IDL compiler is available at `ftp://ftp.omg.org/pub/OMG_IDL_CFE_1.3`. This CORBA IDL compiler is simply a "front-end" that uses a `yacc` parser to generate an in-memory abstract syntax tree (AST) from CORBA IDL input. Developers must provide customized back-ends that can read the AST and generate source code (such as C++, C, or DCE IDL) appropriate for the target platform. Figure 4 depicts the components of the tool and the way it is used.

The SunSoft IDL compiler front-end contains the following components:

• **CORBA IDL Parser:** The parser comprises a `yacc` specification of the CORBA IDL grammar. The action for each grammar rule invokes methods of the AST node classes to build the AST.

• **Abstract Syntax Tree Generator:** Different nodes of the AST correspond to the different constructs of CORBA IDL. The front-end defines a base class called `AST_Decl` that maintains information common to all AST node types. Specialized AST node classes (such as `AST_Interface`) inherit from this base class.

The SunSoft IDL compiler also defines a class called `UTL_Scope`, which maintains scoping information. All AST nodes representing CORBA IDL constructs that can define scopes (such as `structs` and `interfaces`) also inherit from the `UTL_Scope` class.

• **Driver:** The driver component directs the parsing and AST generation process. It reads an input CORBA IDL file and invokes the parser and the AST generator.

A.2 Customizing the SunSoft CORBA IDL Compiler for DCE IDL

Our migration tool takes a DCE IDL file as input. Since we used the CORBA IDL compiler as our basic tool, we customized it as described below.

• **Customizing the front-end:** We modified the CORBA IDL grammar by augmenting it with rules that recognize DCE IDL syntax. This introduced several new AST node classes to the existing repertoire of AST node classes.

• **Providing a back-end:** We defined several specialized C++ classes that derived from the AST node classes. These specialized classes defined additional methods responsible for code generation.

The parser parses the input DCE IDL and creates an AST that comprises the specialized nodes described above. The back-end comprising the code generator walks through the AST and invokes the different code generation methods of the specialized AST node classes.

A.3 Tool Statistics

In this section, we provide statistics in terms of the number of classes defined by the original CORBA IDL compiler, the lines of code in the original CORBA IDL compiler and in the tool, additional classes defined by the tool, and the effective reuse of the original code.

• **Statistics for the Original CORBA IDL compiler:** The original CORBA IDL compiler defines roughly 27 AST node class and 16 utility classes. It provides a "null" back-end that prints out the input source by traversing the abstract syntax tree.

Category	Original	Tool
AST	7,892	7,910
Driver	1,398	1,402
Back end	1,778	4,005
Parser	3,727	5,346
Include	7,766	7,958
Utilities	4,382	4,643
Narrowing	163	163
TOTAL	27,086	31,427

Table 2: Comparison of Lines of Code

• **Statistics for the Tool:** The tool provides a back-end that defines roughly 20 C++ classes that derive from the various AST classes. In addition, each class provides methods for code generation. Table 2 provides statistics in terms of the lines of source code in the original CORBA IDL compiler and the tool.

Table 2 reveals that the additional effort to build the tool was roughly 5,000 lines of source code. The tool reused most of the class library provided by the original compiler.