DCE(Distributed Computing Environment) Porting Tool [1]

Sridhar Muppidi, Neal Krawatz, Ganesha Beedubail, Rakesh Burudgunte, Anish Karmarkar, Anil Gurijala, Paul Brutch, Kyle Walzel, Vance Swaggerty, Ganesh Jayadevan, Ramesh Narayanan, Charlie Richardson, Sunil Santha, Willis Marti, Udo Pooch.

Department of Computer Science, Texas A&M University,
College Station, TX - 77843.

Technical Report (TR_95-009)
February 1995[2].

Abstract

This paper describes the design and implementation of a conversion tool. This tool, called Distributed Computing Environment (DCE) Porting Tool, converts an Open Network Computing (ONC) Remote Procedure Call (RPC) application to a DCE RPC application. The manual intervention needed during the conversion process in minimal. The paper briefly describes RPC programming and highlights the difference between ONC RPC and DCE RPC. The design decisions of the Porting Tool is explained. The implementation uses LEX and YACC (language processing tools supported by Unix) for conversion logic and OSF/MOTIF for Graphical User Interface (GUI). The paper concludes with current limitations of the Tool and the possible future work.

Keywords: Remote Procedure Call (RPC), Distributed Computing Environment (DCE), Open Network Computing (ONC), Interface Definition Language (IDL), Porting Tool.

1 Introduction

In simple terms, **Distributed Processing** means that the (individual) components of an application run (execute) on different nodes in a network. The foremost reason for the current popularity of distributed processing is the modular nature of the applications that are being computed. Many of today's high-end applications are inherently distributed. Reasons for the popularity of distributed processing are: high availability of (low cost) computing facilities (such as hardware, software and network components), ability to share the expensive resources over a network, the availability of large computing power of small computers, and inherent fault tolerance associated with the distribution of computing resources.

[1]This project was supported by IBM Austin under contract number 32525-42640-CS.
[2]This Paper was revised in October 1995. The revised version appears in the Proceedings of the IFIP/IEEE International Conference on Distributed Platforms (ICDP'96), Dresden, Germany, February 1996.

From the programming point of view, distributed processing can be categorized into various models (Lockhart, 1994). Here we explain the **client-server** model, used in this paper. For a discussion of other models refer to (Lockhart, 1994).

In the client server model, each interaction is characterized by an asymmetrical relationship between two software processes. One, the client, makes requests for a service, and the other, the server, provides the service for the request. The processing flow is characteristically synchronous (blocking) in nature. That is, once the client makes a request, it waits until the service has been performed. The communication is from client to server, and then from server to client.

For client server programming, one can use messages for the interaction between client and server. But, message passing may be tedious and error prone. To ease the task of client server programming, RPC programming model was developed. RPC relieves the user from the tedious task of low level message passing by hiding the complex details of message passing from the programmer.

Two prominent commercially available RPC programming systems are ONC RPC and DCE RPC. ONC RPC was developed by Sun Micro Systems, while DCE RPC is supported by Open Software Foundation (OSF).

This paper details the design and implementation of a DCE Porting Tool which automatically convert ONC RPC programs to DCE RPC programs. The motivation for such a tool is derived from the fact that, DCE is a full fledged distributed programming environment supported by the OSF consortium. An organization may want to switch its computing platform to OSF DCE for various reasons. But if it has a substantial base of ONC RPC applications, it needs to migrate those applications to the DCE platform. Otherwise it may have to maintain two different computing environments which may be expensive in terms of infrastructures required. Porting the ONC applications manually may be prohibitive in terms of time and money. An automated Porting Tool would be very convenient for this task.

The rest of the paper is organized as follows. Section 2 briefly explains the RPC programming, including ONC RPC and DCE RPC. Section 3 presents an overview of Porting Tool. Section 4 highlights the design and implementation of the tool. Section 5 gives the summary and limitations of the current tool.

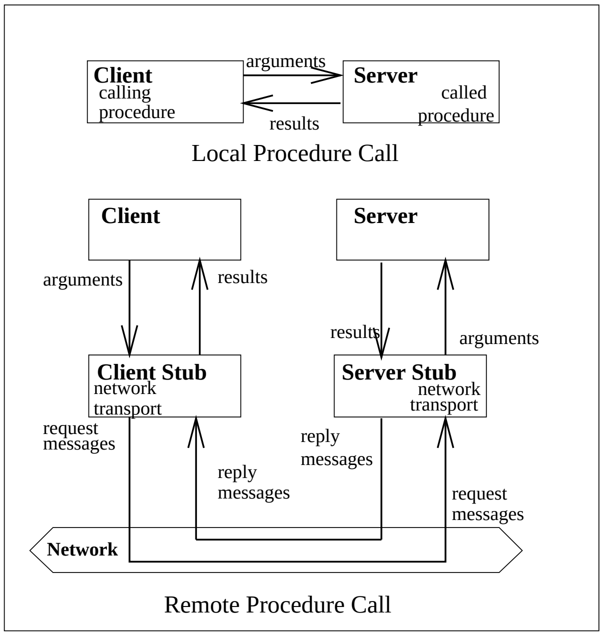

Figure 1: Difference between the Local and Remote Procedure Calls.

2 RPC Programming

In a traditional program, a procedure performs a specific task. The procedure is a part of the whole program and it is statically linked to the program load module. A procedure call, in this case, causes a branch to a different part of the program (and a branch back to the calling point after the completion of the called procedure). The parameters for the call are passed back and forth through the local stack or a parameter list.

RPC uses a request-and-reply communication model. The client program sends a **request** message to the server procedure (process); the server sends back the **reply** message. The client and server communicate by means of two **stubs**, one for the client and one for the server. A **stub** is a communication interface that implements the RPC protocol and specifies how messages are constructed and exchanged. The stub performs the **marshaling** of arguments (the process of converting the arguments to a form, that can be transmitted over a network) and **unmarshaling** (the process of reconstructing the arguments from the marshaled form).
Figure 1 illustrates the difference between local and remote procedure calls (Bloomer, 1992). RPC extends the familiar local procedure call semantics into distributed programming. In the case of RPC, the called procedure is not a part of the program. The called procedure executes in a different process (address space) and not necessarily on the same machine. The invocation

of the procedure and the argument passing are handled through some type of inter process communication (IPC). In a client-server model, the servers may be implemented as a remote procedure, and the client may request a service of the server through remote procedure calls. For the client application, the RPC just looks like a local procedure call.

2.1 Developing an RPC Application

Normally an RPC application is developed in two steps as explained below.

- Specify the protocol (or interface) for the client server communication.
- Develop the client and server programs.

The first step, specifying the interface/protocol precisely, states the interface of the procedure, i.e. it states the procedure name, type and the number of input arguments and the type and number of the output arguments (returned results). Normally a vendor provides a standard specification language for specifying this information. Generally these specification languages are referred to as **IDL** (interface definition language) or **RPCL** (RPC language). The vendor also provides compilers for these interfaces. The compilers generate the necessary stub and header files.

The second step involves the development of client and server programs. This is the implementation of the application logic. The client and server code should be compiled and linked with the respective stub files to get the client and server load modules.

Although RPC programming is conceptually the same on different vendor supplied platforms, syntactically and semantically they are not the same. Below we will explain RPC programming in two common platforms (ONC and DCE).

2.2 ONC RPC Programming

In the ONC environment, the interface (or protocol) is specified in the External Data Representation (XDR) language. The detailed description of XDR is given in (Bloomer, 1992). The protocol compiler is called **RPCGEN**.

Figure 2 shows the general (normal) steps involved in the application development with RPCGEN (Bloomer, 1992). It also shows the various conventions that are used. Note that in the ONC environment, an RPC application can be developed without the use of RPCGEN and a Protocol specification. In fact it is possible to develop an RPC application with various levels of details (Bloomer, 1992). But in this paper, and for the DCE porting tool, only ONC RPC programming using RPCGEN is considered.

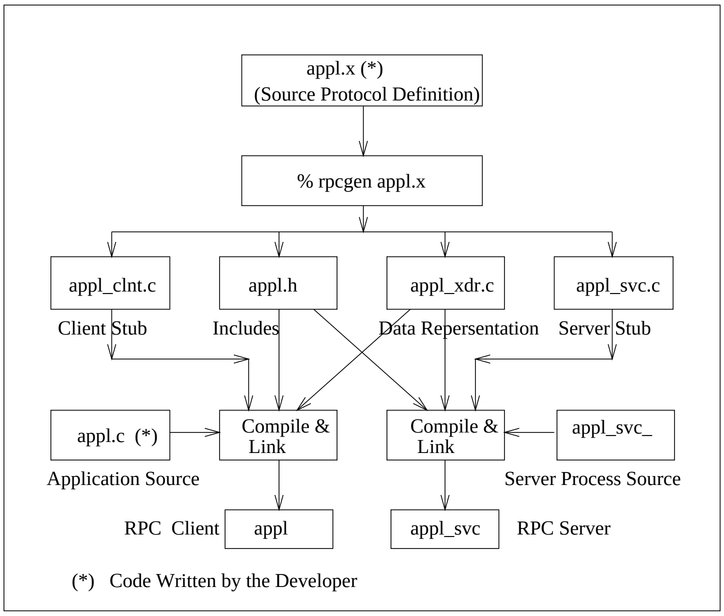

Figure 2: Steps involved in developing an ONC RPC application.

2.3 DCE RPC Programming

DCE RPC programming is slightly more involved than ONC RPC programming. This is because DCE provides more services (Threads, RPC, Security, etc.) to a distributed application developer.

An application developer can use one or more of the services to develop an application. Since the objective of this project is the conversion of applications from the ONC RPC platform to the DCE RPC platform, only DCE RPC programming is explained here. For a detailed description of programming in DCE see Lockhart (1992), Shirly (1993), OSF (1991a) and OSF (1991b).

The first step is to write an interface. In case of DCE, the interface specification language is IDL and the compiler is **idl compiler**. In DCE each interface is uniquely identified (in time and space) by a Universal Unique Identifier (**UUID**). This UUID is registered in the Name/Directory service along with the interface name. The UUID (in this context also known as Interface UUID) is used by the client to locate a server in the network. The DCE provides a utility called **uuidgen** to generate UUIDs.

After writing the interface (the UUID needed by the interface is generated by uuidgen), the

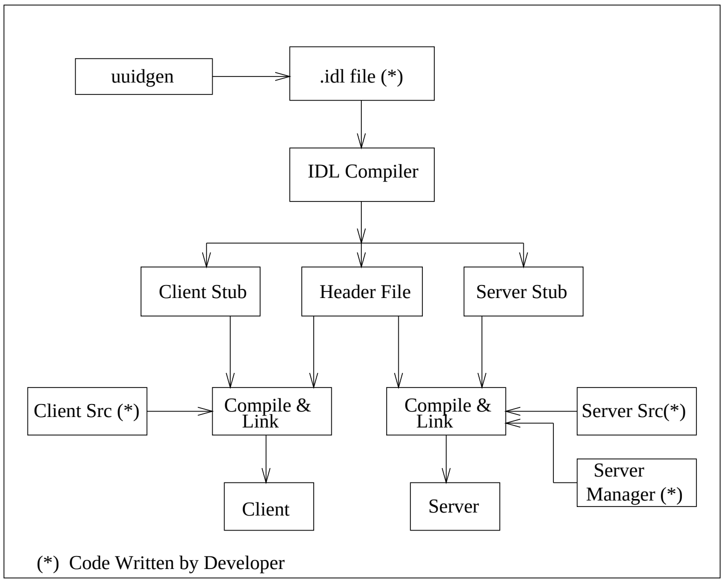

Figure 3: Steps Involved in Developing a DCE RPC Application.

client and server programs (application logic) should be developed. Thus, apart from the syntactic (and some semantic) differences, the program development procedure in DCE RPC and ONC RPC are similar. In DCE, it is also necessary to develop a manager program for the server component. The manager program registers the server in the name space. Note that this program is not present in case of ONC RPC (this functionality is taken care by RPCGEN in stub file). The **idl compiler** generates the required stub and header files. Figure 3 shows the development process and conventions involved in DCE RPC programming.

3 Porting Tool Overview

The goal of the Porting Tool is to automatically convert ONC RPC programs into DCE RPC programs. In doing so, the manual intervention should be minimal. This project only targeted converting the ONC RPC source code which includes the interface definition source (the .x file) and application logic (client and server - .c files). Conversion of Makefiles is not attempted. However given a standard template, the effort required in writing a new Makefile is minimal.

3.1 Design Overview

While designing such a Porting Tool the following factors should be considered:

- The syntax difference between ONC RPCL (RPC Language) and DCE IDL.

- The semantic difference between the similar language constructs on the two platforms.
- The service facilities provided by one that is not supported by the other and vice versa.

Only the syntax and semantic difference between the interface/protocol definition languages in considered here. It is assumed that the source (application logic) is written in 'C' language and both platforms support 'C'. It is also assumed that the application is working correctly on the ONC platform (as specified in the application requirement).

The design of the Porting Tool involves mainly the design of the following two sub components (modules).

- The design of a conversion module that converts the ONC interface definition file (the **.x** file) to the DCE interface file (the **.idl** file). This module is called **X2IDL** module.
- The design of a conversion module, that converts ONC application programs (client and server programs - **.c** files) into corresponding DCE application programs (**.c** files in DCE). This module is called the **Function Converter** module.

The following subsections discuss the design of these modules.

3.1.1 Design of the X2IDL Module

The following logical steps are involved in the design of the X2IDL module.

- Study the *grammar* of ONC RPCL and DCE IDL.
- Identify the language constructs in ONC RPCL that can be *one-to-one* mapped to the language constructs of DCE IDL (with similar syntax but with different keywords).
- Identify the language constructs in ONC RPCL that can be mapped to similar language constructs in DCE IDL but with different syntax and different keywords.
- Identify the language constructs in ONC RPCL that can be safely omitted when porting to DCE IDL.
- Identify the language constructs that are to be added to the **.idl** file that are (or equivalent of which) not present in ONC RPCL.
- Identify that part of the code, that may need re-organization (restructuring) when porting to DCE.

The following code is an example of ONC RPC .x file(interface specification for ONC RPC).

```
/* rdb.x: remote database access protocol */

/* preprocessor directives */

%#define DATABASE "personnel.dat" /* '%' passes it through */

/* constant definitions */
const MAX_STR = 256;

/* structure definitions, no enumerations needed */
struct record {
string firstName<MAX_STR>; /* <> defines the maximum */
string middleInitial<MAX_STR>; /* possible length */
string lastName<MAX_STR>;
int phone;
string location<MAX_STR>;
};

/* program definition, no union or typdef definitions needed */
program RDBPROG { /* could manage multiple servers */
    version RDBVERS {
        record FIRSTNAME_KEY(string) = 1;
        record LASTNAME_KEY(string) = 2;
        record PHONE_KEY(int) = 3;
        record LOCATION_KEY(string) = 4;
        int ADD_RECORD(record) = 5;
    } = 1;
} = 0x20000001; /* program number ranges established by ONC */
```

Figure 4: Example code for ONC RPC .x file.

In the following, some of the findings of the above analysis are discussed. A detailed discussion of the findings is outside the scope of this paper. Interested readers can refer to the System Design Document (SDD) for the DCE Porting Tool (SDD, 1994). Interested readers can also refer to the grammars of ONC RPCL and DCE IDL in (Bloomer, 1994 and OSF, 1991a). For brevity, the grammars are not presented here. However an example of the ONC RPCL interface file and the (the .x file) ported DCE IDL file (the .idl file) is given in the Figures 4 and 5. These will illustrate most of the points discussed here.

A new construct, the UUID along with the **interface name** is required by the DCE IDL. The interface name is derived from the '.x' **program name** (the '.x' file is the protocol definition

The following code is an example of converted ONC RPC .x file.
The converted file is .idl file (interface specification for DCE RPC).

```
[
/* Program Name RDBPROG and ID 0x20000001 */
uuid(002aa21a-8d1a-1e0c-9a0b-80c28e3eaa77),
version(1) /* version name RDBVERS*/
]
interface ONC_RDBPROG
{/* * rdb.x: remote database access protocol */
/* preprocessor directives */
/* constant definitions */
/* structure definitions, no enumerations needed */
typedef struct record {
    [string, ptr] char * firstName ; /* <> defines the maximum */
    [string, ptr] char * middleInitial ; /* possible length */
    [string, ptr] char * lastName ;
    long phone;
    [string, ptr] char * location ;
} record ;

/* program definition, no union or typdef definitions needed */
/* could manage multiple servers */

/* function 1 */ [idempotent]
record * firstname_key_1([in] handle_t cl, [in] char * arg);
/* function 2 */ [idempotent]
record * lastname_key_1([in] handle_t cl, [in] char * arg);
/* function 3 */ [idempotent]
record * phone_key_1([in] handle_t cl, [in] long * arg);
/* function 4 */ [idempotent]
record * location_key_1([in] handle_t cl, [in] char * arg);
/* function 5 */ [idempotent]
long * add_record_1([in] handle_t cl, [in] record * arg);

/* program number ranges established by ONC */
}
```

Figure 5: Example code of converted (DCE RPC interface) file.

file in ONC RPCL and program name is the name given for the interface). The (interface) UUID is generated using *uuidgen*. The "version number" is mapped *one-to-one*. Note that in case of the .idl file the interface name and the version number appear at the beginning of the source file, but in the case of the .x file it appears at the end of file. Because of this, the conversion task is carried out as a two pass process (more about the implementation detail is given in the next section).

The following are some of the mapping decisions made in the design:

- *string* type in the .x file is converted to *char pointer (char *)* in the .idl file.
- *int* type in the .x file is converted to *long int* in the .idl file. This is because IDL does not support an *int* type.
- *struct* type of the .x file is converted to *struct* type of the .idl file. Here it is necessary to add some additional keywords to the *struct construct* of the .idl file so that it retains the same meaning in the .idl file as in the .x file. All of the structure members are converted accordingly.
- *discriminated union* type of the .x file is converted to a similar structure in the .idl file but with different syntax.
- The procedure numbers in the .x file are omitted in the .idl file. IDL does not have procedure numbers.
- The procedure has only one parameter in the .x file (basically the input parameter to the procedure). In the .idl file one more parameter is added to the procedure. The added parameter is the *binding handler* parameter. Binding handler (Shirly, 1993) is used by DCE RPC call when explicit binding method is used. In the converted DCE RPC program, explicit binding method (to retain the same semantic of ONC RPC call) is used.
- Some constructs of RPCL (the .x file) are not supported in IDL. RPCGEN also acts as a preprocessor for the .x file. It puts some constructs like macro definitions in an include file (similar to .h file in 'C'). Examples are *CONST* definitions and some conditional compilation statements. While porting to DCE, such statements are written as macro definitions (*#define's* in 'C') in a special file named as foo_M.h (where foo is the base file name of the .x file). This file will be included in all the ported source files (the client, server and the manager files).

- A file, foo_svc_DCE.c (where foo is the same as above) is generated during the conversion process. This file contains the code for registering the DCE RPC interface with the Name Service and DCE runtime. Later we will give more details about this file.

3.1.2 Design of Function Converter Module

For the conversion of source files (.c files) a different approach is taken. Since the same language syntax and semantics are used in both the environments (ONC RPC and DCE RPC), there is no need for any syntactic changes while converting application source (logic) files from the ONC environment to DCE (the programming language 'C' is used in both the environments). However, in the ONC RPC environment, the .c files contain the function calls that are supported and meaningful only in ONC. Many of these functions are not supported in DCE (or not necessary in DCE). The main design tasks in the conversion of source files are the following.

- Identify the ONC function calls that can be directly mapped with the same semantics to DCE function calls.
- Identify the ONC function calls that have similar (may be with slightly different semantics) DCE function calls.
- Identify the ONC functions that have no equivalent DCE function calls. For some of these functions it is necessary to develop equivalents, but some of these functions can be safely omitted.
- Identify ONC specific *"#include files"* and replace them with DCE specific *"#include files"*.

A detailed discussion of the result of the above study is given in the appendices of SDD (1994). Some of the salient design decisions are discussed here. In the converted application, the *explicit binding* method of the DCE RPC is used. This is because in ONC RPC, the client knows the server location and directs the RPC call to a specific server. We wanted to retain this feature of the original program in the DCE environment. For this purpose, the *clnt_create()* function of ONC RPC is replaced by a new function called *DCE_clnt_create()*. This new function searches the DCE Name Space (cell directory) and obtains the (explicit) binding handle. This binding handle is used as the first parameter in all the RPC calls. Note that all of the ONC RPC calls are converted to DCE RPC calls just by including this binding handle. The foo_M.h header file has the required macros to effect this change. Also, all ONC specific function calls (not ONC RPC calls) are prefixed by *'DCE_'* (for example, onc_function() is replaced by DCE_onc_function()). A separate file called *map.c* implements all these functions with the necessary logic. The *map.o* file is linked with respective .o files of server and client. In DCE, it is necessary to register the server in the DCE Name Space (DCE Cell Directory Service) and with the DCE runtime. This

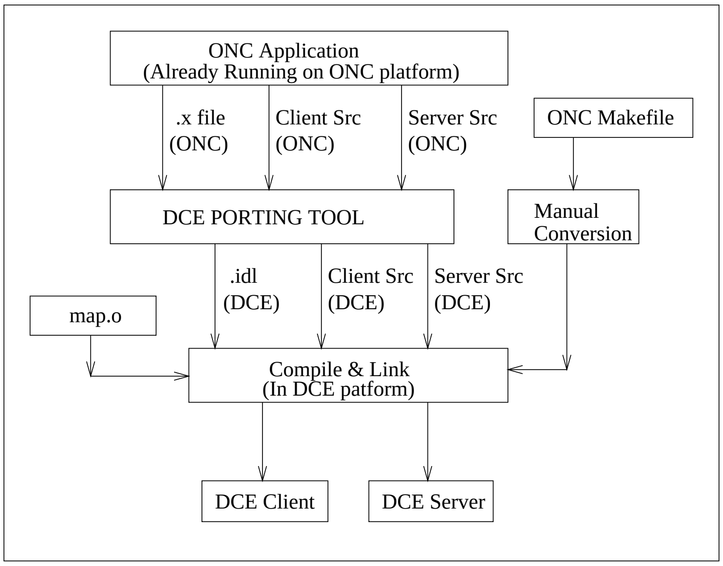

Figure 6: Expected Usage of the DCE Porting Tool.

is similar to ONC, where it is necessary to register the server with the ONC runtime, called *portmapper*. But, this is done by the stub files transparently when the .x file and RPCGEN are used in the application development. To implement this functionality in DCE a separate file (with file name foo_svc_DCE.c) is generated during the conversion process. The code in this file implements the logic for registering the server in the DCE Name Space and DCE runtime. The content of this file varies little for any RPC program. The only changes are the interface related names. These names are obtained from the .x file. Figure 6 illustrates the conversion process from ONC RPC using the Porting Tool.

3.2 Implementation Overview

The implementation involves analyzing the syntax (and recognizing the language tokens) for the .x file (ONC RPCL). It is also necessary to parse the .x file, so that the logically equivalent .idl file can be generated. This is similar to interpreting the .x file. Language processing tools **LEX** and **YACC** are ideal for such a job. Thus LEX and YACC, (Levine, 1992) are used in the implementation of the Porting Tool.

The implementation of the Porting Tool contains three main modules.

X2IDL This module converts the ONC RPC .x file into DCE RPC .idl file. This module also generates **_M.h**(macro header file) and **_svc_DCE.c**(Server registration logic) files. The logic of this module is implemented using LEX and YACC.

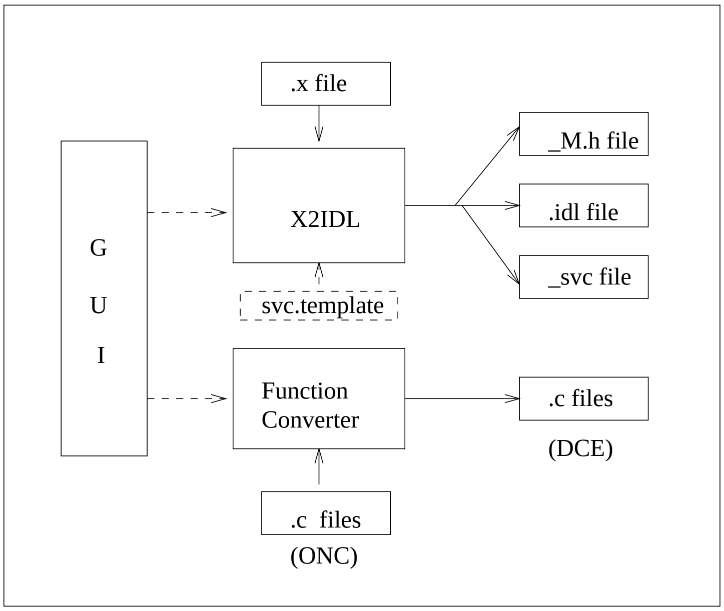

Figure 7: Overview of the Porting Tool.

Function Converter This module converts the source files (.c files - both client and server source files) of ONC RPC application into the source files of DCE RPC application. The logic of this module is implemented using LEX.

GUI - Graphical User Interface This module is responsible for presenting the Porting Tool to the user in a friendly manner. It is developed using OSF/MOTIF according to the suggested style (OSF, 1990a 1990b 1990c).

Figure 7 shows a high level block diagram of the modules in the Porting Tool.

4 Implementation Details

In this section, some of the salient features of the implementation is discussed. It is assumed that the reader is familiar with language processing concepts such as context free grammars, lexical analysis, parsing and tools like LEX and YACC.

4.1 Implementation of X2IDL Module

The X2IDL module converts a .x file into to a .idl file. As explained in the design overview, the X2IDL module needs to generate two additional files other than .idl file. Figure 8 shows the

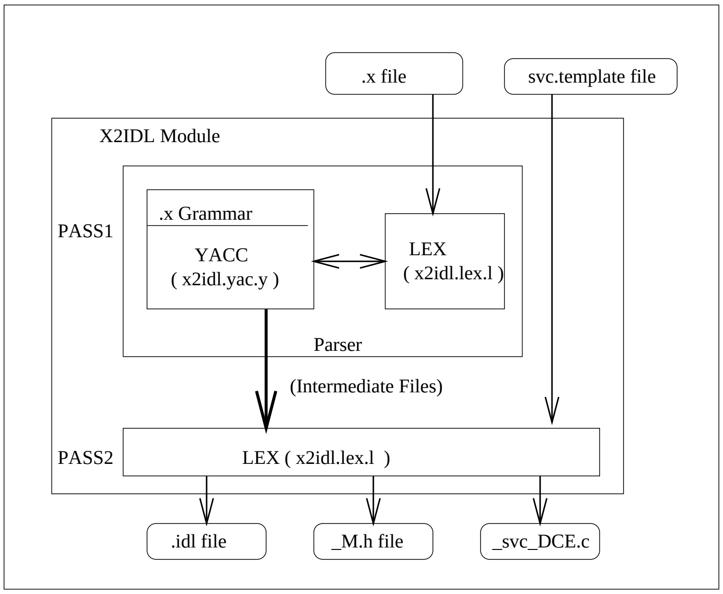

Figure 8: Block Diagram of X2IDL module.

block diagram of the X2IDL module. It also shows the sub modules of the X2IDL.

The X2IDL module takes two input files and produces three output files. The input files are:

.x file : This is the ONC RPC interface definition file.

_svc.template file : This file contains the standard code for registering the interface with the DCE name service and DCE runtime. This file has some *place-holder* variable names. These variable names are replaced by appropriate interface related names, extracted from the .x file.

The output files of the module are:

.idl file : This is the converted .x file. It is the interface file for the DCE RPC.

_M.h file : This is the macro (header) file. The contents of this file comes from the .x file. It contains .x file statements that do not have .idl equivalents. The rationale for this file was explained before. It also contains macro definitions that will change the structure of ONC RPC calls so that now they work in DCE. It also contains some other miscellaneous definitions. For details see SDD (1994).

_svc_DCE.c file : This file is the _svc.template file with the proper interface related variable names substituted.

The X2IDL module has two major sub modules called *pass1* and *pass2*. In effect, these two modules make two passes on the .x file. It is necessary to have two passes for the following reason: the program-name in the .x file is available somewhere at the middle of the the .x file and it is needed for deriving the interface name for the .idl file. In .idl file interface name appears near the beginning of the file. For details refer to SDD (1994).

Module pass1 is the parser for X2IDL. It uses .x grammar (ONC RPCL grammar) to parse the .x file. The parser is implemented using YACC and calls the LEX module for getting the tokens from the .x file. The logic of this module is straight forward. When it can reduce a set of tokens (according to the grammar) it outputs a string to the intermediate file(s). This string may contain the actual .idl statement or *place-holder variables* which will be substituted in the second pass through these intermediate files. The content of the output string depends on the input tokens read and the conversion specifications identified in the design stage. This module also stores some values in the global variables. These global variables are used by the pass2 module (eg. .x program-name).

Module pass2 creates the output files. It reads the intermediate files created by pass1 module and substitutes the *place-holder variable* names with the appropriate values (that are stored in the global variables by pass1 module) and writes the complete output files. It also reads the _svc.template file and fills out the interface related names and writes out _svc_DCE.c file.

Module pass2 is implemented using LEX. All this module needs to do is to identify the *place-holder variable* strings in the intermediate files and substitute them by the correct values. It need not understand the syntax of the input files. Thus, LEX is enough to do the job. In the implementation, the same LEX code is used for pass1 and pass2. Of course different lexer states are used in the two passes. Details are given in SDD (1994).

4.2 Implementation of Function Converter

The implementation of the function conversion module (**function converter**) is quite straight forward. As identified in the design stage (section 3), the function converter has to do the following for each of the source files(.c files):

- Replace ONC RPC specific *#include*(header) files by DCE specific header files.
- Include foo_M.h macro(header) file in the converted source file.
- Prefix all ONC function calls *onc_function()* by *DCE_* (i.e. it becomes *DCE_onc_function()*.

From the above requirement, it can be observed that the function converter only needs to recognize the language tokens. Thus LEX is sufficient for implementing the function converter.

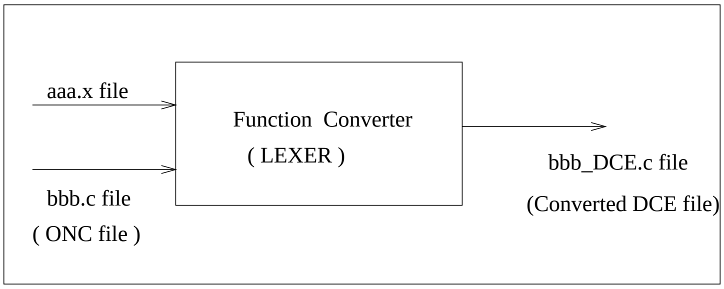

Figure 9: Block Diagram of Function Converter Module.

Figure 9 shows a block diagram of the function converter. Note that the function converter takes two input files. The foo.x file is needed just to derive the name of the **_M.h** macro file. This macro file is to be included in the converted source file (*_DCE.c file*).

The function converter module, has a look up table. The table entries are of the form *[onc_function, status]*. The status is a character variable that can take a value of either "y" or "n" that indicates whether the respective onc_function has a corresponding DCE function (or equivalent DCE function has been implemented). Figure 10 shows the logic flow of the function converter module. There are only a couple of ONC specific header files (eg. *<rpc/rpc.h>*), so these file names are hard coded in the program. Also at the beginning of the output file, a statement to include the generated macro file is generated (*#include "foo_M.h"*).

4.3 Implementation of GUI

The Graphical User Interface (GUI) provides a user friendly environment, to use the Porting Tool. GUI is developed using OSF/MOTIF in accordance with the suggested style (OSF, 1990a 1990b 1990c). It provides convenient pull down menus, text entry fields and file browse windows for the easy access of Porting Tool components. The implementation logic of GUI follows the *event-driven* programming model that is generally used in the case of Motif/X Window system programming. The details are omitted here.

The GUI provides the following facilities to the user.

- General interface for the user to select file(s) (file browser).
- Interface to invoke X2IDL and Function Converter modules.
- A facility to store the current selection of file name(s) in a **.prj** file. This .prj file can be reloaded later. This saves some keystrokes.

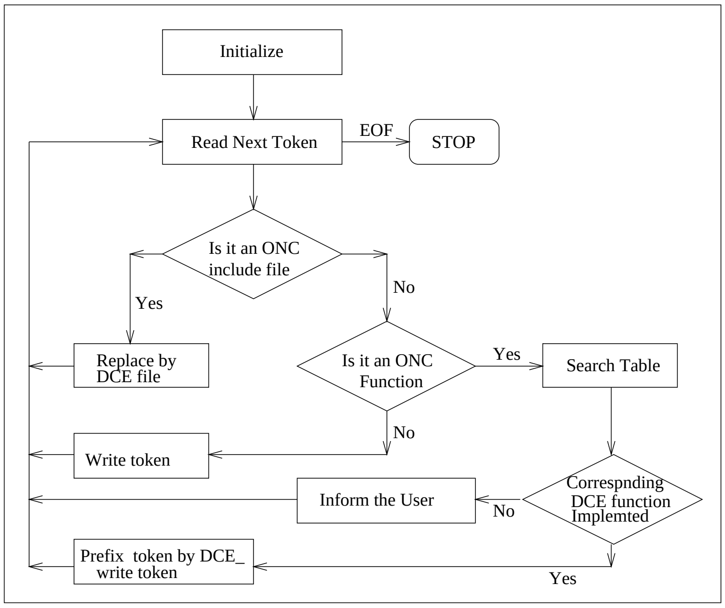

Figure 10: The Logic Flow of Function Converter Module.

- Displays the statistics of the conversion process (the files involved and the number of lines of code etc.).

5 Summary and Limitations

This paper briefly described the design and implementation of DCE Porting Tool. The objective of the project was to convert ONC RPC programs to DCE RPC programs automatically. The current implementation of the tool on an IBM AIX platform works as per specifications. However, the tool has some limitations.

The tool works only if the ONC RPC application was developed using RPCGEN (i.e., with the .x file). As noted before, an ONC RPC program can be developed without using the .x file. In this case the tool cannot be used. Also even while using RPCGEN, one can use low-level ONC function calls (like call_rpc()) which cannot be easily mapped to an equivalent DCE call. This is because, in DCE it is assumed that all applications will be developed using IDL (the .idl file). Also while converting the ONC programs to DCE, the new facilities provided by DCE (like name service) are not effectively used. We wanted to run the ONC RPC applications in DCE without re-writing the application. Changing the application logic for taking the advantage of DCE services was not considered. Makefiles are not converted. However, this is not a serious limitation. The future work will concentrate on removing some of the above limitations.

Acknowledgment: The authors also would like to express their appreciation to Mr. Scott Page at IBM Austin for his valuable comments throughout this project. Finally, we are grateful to the referees (ICDP'96), whose comments helped to improve the presentation.

References

[1] Bloomer, J. (1992) Power Programming with RPC, O'Reilly & Associates, Inc., Sebastopol, CA.

[2] Levine, J.R, Mason, J. and Brown, D. (1992) LEX & YACC, O'Reilly & Associates, Inc., Sebastopol, CA.

[3] Lockhart, H.W. (1994) OSF DCE, Guide to Developing Distributed Applications, McGraw Hill, Inc., CA.

[4] Open Software Foundation, (1990a) OSF/MOTIF Programmers Guide, Prentice Hall, New Jersey.

[5] Open Software Foundation, (1990b) OSF/MOTIF Programmers Reference, Prentice Hall, New Jersey.

[6] Open Software Foundation, (1991a) OSF DCE Application Development Reference, Prentice Hall, New Jersey.

[7] Open Software Foundation, (1991b) OSF DCE User's Guide and Reference, Prentice Hall, New Jersey.

[8] Open Software Foundation, (1990c) OSF/MOTIF Style Guide, Prentice Hall, New Jersey.

[9] SDD, (1994) System Design Document for DCE Porting Tool, Prepared for IBM Austin, Contract No. 32525-42640-CS, Texas A&M University.

[10] Shirly, J. (1993) Guide to Writing DCE Applications, O'Reilly & Associates, Inc., Sebastopol, CA.