Abstract: Nowadays the Distributed Object Oriented Environments are becoming widely used. One of them is OMG’s Common Object Request Broker Architecture (CORBA). The paper deals with the short description of a tool named U_CORBA, which supported the design process of CORBA applications. The new tool was implemented under the MICO CORBA implementation. The tool gives the opportunity of generating IDL files and application C++ headers based on application class diagrams.

Topics: languages and tools, distributed computing and operating systems

1. Introduction

Nowadays Object Oriented models are being used in many applications. On the other hand with the increased necessity of enabling computers to work together a new kind of programming and working environments are now becoming widely used - the Distributed Object Oriented Environments. It is still to early to make guesses on which one of these systems will become the standard. So far the strongest contestants are IBM’s System Object Model (SOM); Microsoft’s Distributed Object Linking and Embedding (OLE); OMG’s Common Object Request Broker Architecture (CORBA). Each one of them has their advantages and drawbacks. The paper deals with the short description of a tool named U_CORBA, which supports design of CORBA applications. The CORBA environment was chosen, as it is the most advanced one in terms of standard definition. Unlike some others, it has already available full programming and runtime environments.

CORBA is a new concept used for creating application in the distributed manner. It is a solution for developing new application or making together some working application [4,7]. Although there are a lot of different developing methodologies and
tools supporting application design, like UML, VPE, TRAPPER and others working at different environments there are no such tool dedicated for developing CORBA applications [1,2,3,6,8]. For designing a CORBA application, Rational Rose is a commonly used environment [6]. This is the main motivation of the work presented in the paper. The main objective of this work is to design and implement a graphical tool that helps in developing CORBA applications. The presented tool named U-CORBA gives opportunity of creation the class diagram of the application using the UML notation and based on it generating the IDL files and application C++ headers. The prototype of the tool was developed with the QT 1.44 library and MICO 2.2.5 [5] on Linux 2.2.5. The structure of this paper is as follows. In section 2, a brief overview is presented of the CORBA architecture. In section 3, the main functionalities of the U_CORBA tool are described. Section 4 presents an example of the use of the tool. Section 5 compares the tool to related tools supporting Object Oriented development. Finally, section 6 presents some conclusions and describes ongoing work.

[For lack of space we omit section 5 in this extended abstract]

2. CORBA Overview

CORBA stands for Common Object Request Broker Architecture and is a platform defined by OMG - the Object Management Group, a consortium of several companies and universities working together in its definition. The main purpose of the OMG is to define a platform for heterogeneous distributed computing in which very different hardware will work smoothly together - from super computers to embedded systems - independently of the operating systems, programming languages and network protocols they might be using.

In CORBA objects interact with each other by means of interface definitions, with the information provided by this interfaces, potential clients of object services are able to know what to expect from objects and how they should interact with them. This interface is defined in OMG’s IDL (Interface Definition Language) which enables object services to be available to other objects written in almost any programming language. By using IDL, the programmer lets the communication infrastructure know the format of all messages an object can receive and send so that, if necessary, they can be automatically transformed from one data representation to another, providing transparent communication between different systems. The communication infrastructure defined by OMG is called the OMA (Object Management Architecture) and it is a set of protocols and services definitions that allow very different objects to interact freely with each other. OMG defined a set of standard interfaces and functions for each component of the OMA. All CORBA services communicate with each other through an ORB (Object Request Broker) that handles and delivers all messages from one component to another so programmers do not have to worry about distribution details, and can concentrate on solving the real problem at hand.

When designing a CORBA application two files are generated: a skeleton and a stub (figure 1). These files, when compiled and linked together with the service implementation will act as translators between the object and the ORB. Because of this the clients and object implementation can even be written in different programming languages, one just has to generate the skeleton and stubs using the appropriate IDL compilers.
There are currently some official IDL languages mapping specifications standardised by OMG: for Java, C, C++, Smalltalk, Ada. Besides these there are other mappings currently not supported by any standard, and many other IDL compilers are specific for a given ORB. As OMG does not force any protocol of communication between the skeletons and the ORB (the IIOP protocol is in common use), a given skeleton will only be able to talk with its corresponding ORB. What happens if the ORB being used is changed? That is not a problem, new skeletons must be generated using the new IDL compiler and linked with unchanged object implementations - thus achieving instant integration. Both client and object implementation are isolated from the ORB by IDL interfaces so that the client does not even have to care about the way objects are implemented, making modifications easy.

3. Application Design Using U_CORBA

The presented tool helps CORBA users in developing CORBA applications (servers and clients) by generating the skeleton of the application by means of IDL files and application C++ headers. A specially designed graphical interface gives the user opportunity of creation the application class diagrams using UML notation and based on it generating the IDL and application C++ header files. The tool is equipped with an user’s editor for writing application body. The main window of the application is presented at figure 2. The specific functions of the tool are accessible using Diagram and Code options. Using the Diagram option the user can create a class diagram and when using the Code option can generate both mentioned files and writes the body of an application.
There are two main problems that appear during tool implementation: definition of the data structures for class diagram representation and the translation grammar for the application C++ headers and IDL code generation. [The description of these functionalities will be detailed in the full paper]

The attribute name and type describe the class attributes. The name is stored in the string and the type can be set to the one of many possible options. That set includes the most popular types like *int, float, string, char*, etc. The class abstracting the attribute has to be equipped with the copy constructor and overloaded assignment operator for the handling of these structures is not just the question of bit to bit copy. Such solution gives the opportunity of future tool developing. Considering attributes in context of the OOP one more feature has to be added - a visibility. Visibility specifies the way of attribute accessing. The name, type, and list of attributes and visibility describe the operations. Name, type and visibility describing operations are stored in similar way as for attributes, but the list of attributes causes more problems. It can be empty or the number of eventual attributes is unknown. Then operation attributes are stored in the list. The list contains only the pointers to the objects. That requires a lot of caution while processing the entities of operations. The list itself is hold by the type provided by the Qt toolkit.

The code is generated for the specific class on the class diagram. The name of that class indicates the name of the file. Different suffixes are added depending on the file type generating * .idl* for the IDL files and * .h* for the application C++ headers. The process of file generation is divided on three steps. In the first step the class name using the described above procedure is established. During the second step information about the links between classes are retrieved from the association database created from the class diagram, it causes some extra lines of code at both generated files. In the last step the analysis of attributes and operations is performed. As long as the number of used types for the attributes or operations is limited to the basic ones the process of generation both files is not complicated. However the generation of application C++ headers for complex types is not so obvious, usually attributes and operations belonging to the one type of visibility are grouped in one place, and depending on the convention those groups are placed in the different places in the generated header file. We assume that for the more clear declaration reading the group with *public* visibility comes first.

Presently the tool covers only a part of the functionality that normally are supported by CASE tools, however experiments performed using our prototype indicate that it is useful during the design of CORBA applications. The prototype is being extended to support all functionality’s required by a full development process.
4. An Example of Using U_CORBA

To present the functionalities provided by this tool, we present the following simple example. Let’s consider the simplified version of a bank account server. It is responsible for managing a user account by depositing and withdrawing the required sum of money, and reporting about the current state of the account only. The server needs to hold some information, like current balance, account identification number, etc. Let’s assume that a bank client is interested in obtaining the credit. Then both of them can be represented as objects in CORBA environment as presented in figure 3.

<table>
<thead>
<tr>
<th>BankAccount</th>
<th>BankClient</th>
</tr>
</thead>
<tbody>
<tr>
<td>int accountNumber</td>
<td>string name</td>
</tr>
<tr>
<td>string userName</td>
<td></td>
</tr>
<tr>
<td>float balance</td>
<td>int takeCredit(float amount)</td>
</tr>
<tr>
<td>int deposit(float amount)</td>
<td></td>
</tr>
<tr>
<td>int withdraw(float amount)</td>
<td></td>
</tr>
<tr>
<td>float report()</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3 Class diagrams for the server and client respectively.

After defining both class diagrams using the implemented graphical environment presented in figures 4 and 5 the IDL files and application C++ headers for the server and client are generated.

```cpp
interface bankAccount {
    attribute long accountNumber;
    attribute string userName;
    attribute float balance;
    long deposit(in float amount);
    long withdraw(in float amount);
    float report();
}
```

```cpp
interface bankClient {
    attribute string name;
    long takeCredit(in float amount);
}
```

Fig. 4 IDL files generated for the server and client respectively.

```cpp
Class bankAccount {
    private:
    long accountNumber;
    string userName;
    float balance;
    public:
    long deposit(float amount);
    long withdraw(float amount);
    float report();
}
```

```cpp
Class bankClient {
    private:
    string name;
    public:
    long takeCredit(float amount);
}
```

Fig. 5 Application C++ headers generated for the server and client respectively.
After translation of generated IDL files to the architecture specific C++ headers the user can start to build the implementation of the server and client behavior using the Code option of the tool (programmer editor).

5. Conclusions

The project is in current study, presently we finished building a prototype. The prototype still misses a lot of features that could be implemented in the further versions. However experiments performed using our prototype indicates that the presented tool will be useful for designing CORBA applications. No specific knowledge about the IDL language is required to build CORBA objects. After translation of the IDL files to the architecture specific C++ headers the user can start to build the implementation of the server/client behavior. The tool helps the developer out in IDL knowledge and allows him to concentrate on the essential part of implementation. The decision of using OO Technology for coding and a GPL implementation of the CORBA system will contribute to ease the implementation of the full set of desired functionality. Moreover, those decisions make the development of further aspects and kinds of experiments possible, since all layers of the implementation are accessible. Having the Rose tool so sophisticated and supporting so many features and recently also ported to the Unix systems, the question about the motivation of building other tool with a similar functionality arises. With a tool released under the General Public Licence it is possible to freely access the code, develop new features and make it more efficient. The dedication to the MICO that we have followed in this project allows taking some benefits and widening the functionality to areas that are not possible for a tool that is so general like Rose. That all opens a chance for a flexible support for developers that are not capable of obtaining the commercial expensive tools.

References