Abstract

In this paper we present a tool - Visual Distribution of Objects (ViDiO) - for graphical deployment of CORBA objects. It allows easy association of hosts with IDL interfaces and produces client, server, and object code in Java based on the drawn diagram and provided XML based templates. It is based on a library designed for platform independent display of computer networks.

1. Introduction

Current trends in software engineering reject central monolithic solutions or pure client/server application in favour of true distributed systems. Intended results are increased flexibility, fault tolerance, and scalability.

Two common problems encountered by developers implementing such a solution in CORBA (Common Object Request Broker Architecture) [4] are the complexity of the problem domain component and the extreme redundancy of written code in the various client and server programs. The first problem is even acknowledged by the OMG (Object Management Group) in their statement that

"CORBA's flexibility gives the developer a myriad of choices, and requires a vast number of details to be specified. The complexity is simply too high to be able to do so efficiently and quickly." [3]

A graphical presentation of both the object distribution and the access by clients can help to comprehend the current design. This allows designers to identify bottlenecks or get initial hints for profiling applications.

With regard to the problem of code redundancy, it is known that the registration of objects and resolution of names to get the corresponding references are tedious at best and invite automatization.

ViDiO is a graphical tool that allows users to visually associate the IDL (Interface Definition Language) interfaces they designed with an arbitrary number of hosts and decide upon the details of naming resolution, object functionality etc. This information is then used to generate code based on templates for various ORBs (Object Request Brokers). An XML DTD (eXtensible Markup Language [10], Document Type Definition) is defined to allow users to write new templates for exotic ORBs or to keep up with changes in the specifications, should they occur.

Fig. 1 illustrates the details of the ViDiO generation cycle.

![Figure 1: The ViDiO Process](image)

The ViDiO tool is passed an IDL file as command line parameter and starts the IDL compiler to obtain stubs, skeletons, and most importantly Java interfaces. These greatly simplify the IDL parsing (see section 2).

When the parser has identified all interfaces, these are shown in the diagram and can be manually associated with clients and servers. The data represented in the diagram is passed to the code generator in the final step. Correct templates are used to create the source for all clients and servers.

Of particular interest are the following topics:

- graphical interface
- IDL parsing
- code generation and XML-based template concept

These shall be discussed in the following sections.
2. The graphical interface

The graphical interface is based on a generic library which provides all classes used in the presentation of the deployment graph.

2.1. The viror.teach.anim.net library

Originally, the viror.teach.anim.net library \cite{2} was produced for the VIROR project (Virtuelle Hochschule Oberrhein) which (amongst other things) analyses the impact of multimedia tools on university lectures. It is a collection of classes intended to facilitate the visualization of network interactions via animation, providing ready-to-use code for packages, routers, and software components. The latter two make it perfectly suitable for the user interface needed for visual component distribution.

2.2. Functionality

viror.teach.anim.net is particularly suitable for the display of all kinds of graph-like structures. This is due to its two-layer architecture. The top layer provides mechanisms for projecting computer networks as described before and was designed with maximum extensibility in mind. New animation elements requiring different data structures are easily integrated.

If the provided level of flexibility does not suffice the lower layer can be used as a basis for completely new animation/graph elements by using its context-free graphics routines which are based on the Java2D API \cite{9}. The 2-layer division is summarized in Figure 2.

![Figure 2: The Layered Library Design](image)

Ignoring components of the library dispensable for the software described, the library provides the following building blocks:

- A canvas for the graphical representation of all elements involved (AnimDisplay)
- A data structure to store and manipulate those elements (SimpleAnimList)
- Classes to store relevant element information one of which is AbstractRouter. It serves as the base class for Node and IDLInterface, that the ViDiO implementation uses to store data on clients, servers, and software components.
- Classes used for visualization of these elements (BitmapRep and VectorRep). All of them are derived from AnimRepresentation.

The AnimDisplay inherits from JPanel and is the component added to the GUI to make use of the library functionality. It periodically repaints the data stored in SimpleAnimList updating the position of the various network elements. Information on these can be obtained via mouse selection which will cause an information/editing panel to pop up showing the relevant data.

The AnimDisplay and SimpleAnimList are powerful classes that allow selection, dragging, dynamic re-sizing, as well as efficient insertion and deletion. Furthermore, platform independence (e.g. screen resolution) was a major design goal.

Figure 3 visualizes the interaction between the various classes.

![Figure 3: Typical Animation Element Classes](image)

2.3. Utilizing the library for ViDiO

The graphical representation of ViDiO displays a set of hosts (which can be clients, servers, or both) and software components (in our case interfaces of CORBA objects). Lines between components and servers signify deployment, and between components and clients they represent a dependency. Hosts that are client and server
at once are a special case which - in the current version - requires manual modifications of the generated code.

Consider the example of an interface I that is to be deployed on a server S and to be used by instances of the client C. All the programmer has to do is connect S and I as well as C and I in the graph. This will result in the correct code for both client and server.

Owing to the potency of the classes provided little implementation had to be done for the user interface of this project. The class viror.teach.anim. StandardButtons provides the GUI components for deletion and insertion, as well as access to the data stored in servers and components. The classes used to store this data (namely Node and IDLInterface) and widgets to alter their content were in fact the major work for the human-machine interaction part of the project.

3. The IDL parsing

Normally, the parsing required for IDL files would be tremendous and necessitate usage of powerful parsing tools such as ANTLR (ANother Tool for Language Recognition) [6]. Considering the fact however, that a thorough parsing is already done by the IDL compiler such dire measures are rendered obsolete. Parsing Java interfaces created by the compiler instead of the original file offers several advantages:

- Java interfaces produced by this compiler have far less legal tokens than the original definition and can be analysed with simpler techniques. Also, syntactic correctness is virtually guaranteed since the output is machine generated. This allows a lean parser implementation which is easier to maintain.
- Future changes in the IDL specification do not require modifications in the parser since it is only analysing Java code.
- Since different CORBA products achieve different levels of compliance with the standard [1], using the corresponding compiler will ensure that the generated code will work with its ORB despite its deviations from the OMG specification.

Obviously, a full-fledged IDL parser would not take advantage of these facts, instead it would be a complex piece of code requiring constant maintenance and change.

4. Code generation

After the visual design part, ViDiO automatically generates code for the user. The code generation algorithm is based on templates, i.e., several pieces of generic code exist and can be used according to situation. For instance there is a template for a generic client.

As mentioned above, care was taken to allow the creation of other templates which fit into the CORBA context. Therefore it was deemed suitable to define an XML DTD which describes the syntax of these templates. The reason for using XML was its widespread popularity as well as suitability for the task at hand:

- The generic code is text with interspersed tags to be replaced by context specific strings (like interface names, ORB-specific parameters, etc.). This can be easily represented by mixed content and empty tags.
- Some pieces of code have to be inserted several times (based on the number of interfaces for example). Beginning and end of such code sections can be easily marked by putting them into a non-empty tag.

The correct set of templates and ORB settings is chosen according to an initialization file (ViDiO.ini) which is read by the code generation module. Example variables are the ORB class (ORBCLASS) and the ORB singleton class (ORBSINGLETONCLASS). These are

```
<!--This DTD facilitates the parsing of the template files-->
<!ELEMENT filename EMPTY>
<!ELEMENT orbclass EMPTY>
<!ELEMENT orbsingletonclass EMPTY>
<!ELEMENT interface EMPTY>
<!ELEMENT name EMPTY>
<!ELEMENT resolve (#PCDATA | interface | name)>
<!ELEMENT declare (#PCDATA | interface)>
<!ELEMENT create (#PCDATA | interface)>
<!ELEMENT register (#PCDATA | interface | name)>
<!ELEMENT read (#PCDATA | interface | name)>
<!ELEMENT write (#PCDATA | interface | name)>
<!ELEMENT template (#PCDATA | interface | name | filename | orbclass | orbsingletonclass | resolve | declare | create | register | read | write)>
```

The correct set of templates and ORB settings is chosen according to an initialization file (ViDiO.ini) which is read by the code generation module. Example variables are the ORB class (ORBCLASS) and the ORB singleton class (ORBSINGLETONCLASS). These are
required, amongst other reasons, because of the recent inclusion of CORBA classes into the JDK (Java Development Kit), which introduces the possibility of name conflicts which can thus be avoided.

Now that template specifics are known, the diagram can be transformed into code. All objects to be deployed on servers are assumed to be servants (i.e. programming language entities of CORBA objects) and cause the generation of `<interface>_Impl.java` files, where `<interface>` is replaced by the correct name. These files contain empty methods unless the user has given files which provide code to be inserted based on method name.

Also, server code is generated to instantiate these objects, to register them with the POA (Portable Object Adaptor), and to make them available to clients. This is done either via the CORBA Naming Service [5], or by file, according to user choice.

The following code snippet is an example from the name service based server template. The code between `<create>` and `</create>` is included for every CORBA object specified in the diagram. `<interface/>` is replaced by the interface name. The code instantiates the servant and registers it with the POA.

```java
byte[] id = null;
<create>
 <interface/>_Impl _<interface/>impl = new <interface/>_Impl();
 id = rootPOA.activate_object(_<interface/>impl);
 obj = rootPOA.id_to_reference(id);
 <interface/> _<interface/> = <interface/>Helper.narrow(obj);
</create>
```

In the case of clients, code is generated to obtain a reference on remote objects which are connected to the client node in the diagram. Naturally, the business logic has to be inserted afterwards but this can be done with more simplicity now.

The names used for registration and resolution are automatically supplied by the tool ensuring consistency. For clients as well as for servers, the required exception handling will be provided.

5. User guide

ViDiO is a tool that is easy to use. It is started from the command line using the IDL file and output directory as parameters. The main screen is shown in Figure 4.

The buttons at the top (1) allow to generate clients and servers that match the situation in the main diagram and to quit the program. The lower buttons (2) are used for deletion of elements (including links), insertion of new nodes etc. The main display is where the actual modelling takes place.

![Figure 4: The Design Screen](image)

All identified IDL interfaces are shown here (3) as well as all clients and servers the user has specified (4). They can be connected to show which servers provide which interfaces as well as usage of these by clients. Selecting diagram components allows the user to display detailed information and adjust properties such as source code to use or details of name resolution (see Figure 5).

![Figure 5: Element Properties](image)
6. Conclusion and future enhancements

The solution presented in this paper offers several interesting advantages. It allows to solve the problems connected to deployment in a distributed system visually - therefore simplifying the process of developing CORBA-based applications. The user can identify bottlenecks and other problems in an early phase which are far more obvious when presented in graphical form. The graphical library generic nature allows other middleware technologies such as RMI (Remote Method Invocation) [8] or DCOM (Distributed Component Object Model) [7] to be integrated into the tool. Furthermore, the code generation module allows the programmer to concentrate on the business logic part by taking care of the code "infrastructure".

A special focus was to allow simple customisation by using XML-based templates which can be edited without altering the actual code of the tool. Thus changes in the CORBA standard or special requirements on behalf of the user can be easily integrated into the automated generation process.

Clearly, ViDiO is in a prototype stage and therefore the possibilities for improvement are limitless. A suitable starting point might be to improve the generation of machines that are both clients and servers. This could be done by distinguishing between two association types - deployment (on servers) and dependency (for clients). At the moment the meaning of an association is context specific. Also UML (Unified Modeling Language) notations could be integrated.

Another desirable extension would be automatic generation of IDL code from Java classes such that a user without any CORBA knowledge can at least create simple applications using this tool.

To proceed further into this direction one could try to find algorithms that automatically group Java classes into components (forming a three-tier architecture) and find a suitable interface.

An enhancement of a completely different nature would be to use the existing animation functionality of the graphics library to perform simulation runs. This would increase the visual support function even more!

All these considerations aside however ViDiO can be said to be a step into a direction which could be considered promising for the future. IDEs like the Borland JBuilder show that programmers have an increasing desire to use visual tools for routine work. ViDiO provides just that in an easy-to-use way.

7. References


Appendix

This appendix contains additional information based on insights gained after the publication of the original paper.

First of all, using a self-made graphics library is probably a waste of time. GEF, which is also used for ArgoUML, is a solid and flexible package that will be used for our current code generation efforts. To go even further, maybe it is even possible to use UML Profiles and generate code based on the XMI output of some professional UML tool. This would save even more work and allow better focus on the core topics.

Due to a change in project focus, ViDiO will not be further pursued. However, many of the ideas will be used for the code generation effort within the COBANA project.