Mapping of CGIF to operational interfaces

A. Puder

International Computer Science Institute
1947 Center St., Suite 600
Berkeley, CA 94704–1198
USA
puder@icsi.berkeley.edu

Abstract. The *Conceptual Graph Interchange Format* (CGIF) is a notation for conceptual graphs which is meant for communication between computers. CGIF is represented through a grammar that defines "on–the–wire–representations". In this paper we argue that for interacting applications in an open distributed environment this is too inefficient both in terms of the application creation process as well as runtime characteristics. We propose to employ the widespread middleware platform based on CORBA to allow the interoperability within a heterogeneous environment. The major result of this paper is a specification of an operational interface written in CORBA's *Interface Definition Language* (IDL) that is equivalent to CGIF, yet better suited for the efficient implementation of applications in distributed systems.
Keywords: CGIF, CORBA, IDL.

1 Introduction

Conceptual Graphs (CG) are abstract information structures that are independent of a notation. Various notations have been developed for different purposes (see Figure 1). Among these are the *display form* (graphical notation) or the *linear form* (textual notation). These two notations are intended for human computer interaction. Another notation called *Conceptual Graph Interchange Format* (CGIF) is meant for communication between computers. CGIF is represented through a grammar that defines "on–the–wire–representations".

The reason for developing CGIF was to support the interoperability for CG–based applications that needed to communicate with other CG–based applications. We argue that for interacting applications in an open distributed environment this is too inefficient both in terms of the application creation process as well as runtime characteristics. Applications that need to interoperate are written by different teams of programmers, in different programming languages using different communication protocols.

A generalization of this problem is addressed by so–called *middleware platforms*. As the name suggests, these platforms reside between the transport layer and the application. One prominent middleware platform is defined through the *Common Object Request Broker Architecture* (CORBA) which allows the interoperability within a heterogeneous environment (see [3]). In this paper we will show how to use CORBA for CG–based applications.

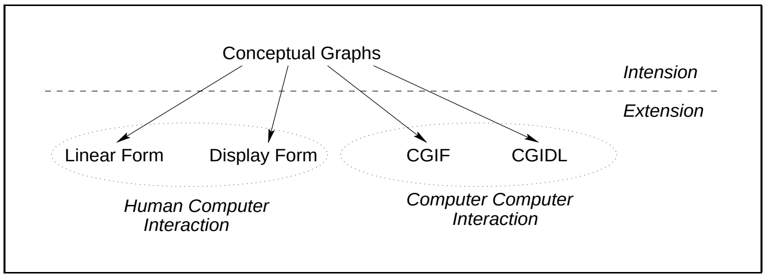

Fig. 1. Different notations represent the intension of conceptual graphs.

The outline of this paper is as follows: in Section 2 we give a short overview of CORBA. In Section 3 we discuss some drawbacks of using CGIF for distributed applications. In Section 4 we present our mapping of CGIF to CORBA which is further explained in Section 5 through an example.

2 Middleware platforms

Modern programming languages employ the *object paradigm* to structure computation within a single operating system process. The next logical step is to distribute a computation over multiple processes on a single machine or even on different machines. Because object orientation has proven to be an adequate means for developing and maintaining large scale applications, it seems reasonable to apply the object paradigm to distributed computation as well: objects are distributed over the machines within a networked environment and communicate with each other.

As a fact of life, the computers within a networked environment differ in hardware architecture, operating system software, and the programming languages used to implement the objects. That is what we call a *heterogeneous distributed environment*. To allow communication between objects in such an environment, one needs a rather complex piece of software called a *middleware platform*. Figure 2 illustrates the role of a middleware platform within a heterogeneous distributed environment.

The *Common Object Request Broker Architecture* (CORBA) is a specification of such a middleware platform. The CORBA standard is issued by the *Object Management Group* (OMG), an international organization with over 750 information software vendors, software developers, and users. The goal of the OMG is the establishment of industry guidelines and object management specifications to provide a common framework for application development. CORBA addresses the following issues:

Object orientation: Objects are the basic building blocks of CORBA applications.

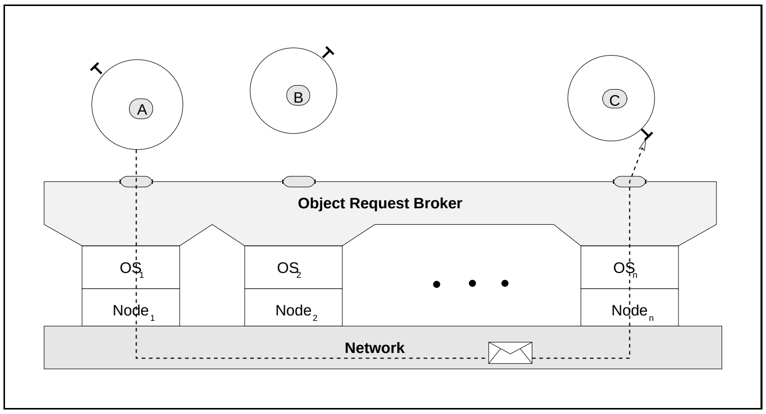

Fig. 2. The middleware hides the heterogeneity of the distributed system.

Distribution transparency: A caller uses the same mechanisms to invoke an object whether it is located in the same address space, on the same machine, or on a remote machine.

Hardware, OS, and language independence: CORBA components can be implemented using different programming languages on different hardware architectures running different operating systems.

Vendor independence: CORBA compliant implementations from different vendors interoperate.

One important aspect of CORBA is that it is a *specification* and not an *implementation*. CORBA just provides a framework allowing applications to interoperate in a distributed and heterogeneous environment. But it does not prescribe any specific technology how to implement the CORBA standard. The standard is freely available via the World Wide Web at `http://www.omg.org/`. Currently there exist many implementations of CORBA focusing on different market segments.

Because CORBA objects can be implemented in different programming languages, the description of an object's *interface* and *implementation* has to be separated. The implementation is done using a particular programming language, and the interface is specified using the *Interface Definition Language* (IDL). The IDL allows the specification of user defined data types and *operational interfaces*.

Different ORB implementations can interoperate through the *Internet Inter–ORB Protocol* (IIOP) which describes the on–the–wire representations of basic and constructed IDL data types as well as message formats needed for the protocol. With that respect it defines a transfer syntax, just like CGIF does. The design of IIOP was driven by the goal to keep it simple, scalable, and general.

3 Using CGIF in a heterogeneous environment

In this section we explain how CGIF might be used in constructing distributed applications and the disadvantages this has. Figure 3 depicts a typical configuration. The application consists of a client and a server, communicating via a transport layer over the network. The messages being exchanged between the client and the server contain CGs as defined by CGIF. First note that this is not sufficient for an distributed application. CGIF only allows to code parameters for operations, but the kind of operations to be invoked at the server is out of scope of CGIF. The distinction between parameters and operations corresponds to the distinction between KIF and KQML. In that respect there is no equivalent to KQML in the CG–world.

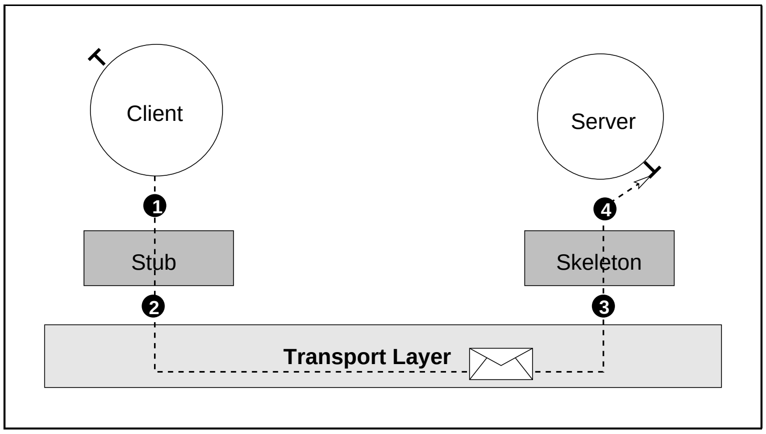

Fig. 3. Marshalling code is contained in stubs and skeletons.

One of our premises is, that the client and server can be written in different programming languages, running on different operating systems using different transport media. A programmer would most certainly define some data structures to represent a CG in his/her programming language. In order to transmit a CG, the internal data structure needs to be translated to CGIF. This is accomplished by a *stub*. On the server side the CG coded in CGIF needs to be translated to an internal data structure again. This is done by a *skeleton*.

The black nodes in Figure 3 show the important steps in the translation process. At step 1 the CG exists as a data structure in the programming language used to implement the client. The stub, which is also written in the same programming language as the client, translates this data structure to CGIF (step 2). After transporting the CG over the network, the CG arrives at the server (step 3). The skeleton translates the CG back into a data representation in the programming language used for the implementation of the server. At step 4 the CG can finally be processed by the server.

CGIF does not prescribe an internal data structure of a programming language. I.e., using CGIF for transmitting CGs, a programmer must first make such a definition based on his/her programming language followed by a manual coding of the stub and skeleton. This imposes a high overhead in the application creation process. The main benefit of using CORBA IDL is, that stubs and skeletons are automatically generated by an IDL compiler and there is a well–defined mapping from IDL to different programming languages. Furthermore, the IDL specification does not only allow the specification of parameters but also for operations which makes it suitable for the specification of operational interfaces between application components.

Although an IDL specification induces a transfer syntax through IIOP similar to CGIF, an IDL specification is better suited for the design of distributed applications. An IDL specification hides the transfer syntax and focuses on the user defined types, which are mapped to different programming languages by an IDL compiler. CGIF on the other hand exposes the complexity of the transfer syntax to an application programmer who is responsible for coding stubs and skeletons.

4 CG interface through IDL

In this section we show how to translate some of the basic definitions of the proposed CG standard to CORBA IDL. The complete mapping would be out of scope of this paper. Therefore, we limit the following discourse to the basic principles we employed while mapping the CG definition to CORBA IDL. The mapping covers definitions 3.1 (Conceptual Graphs), 3.2 (Concept) and 3.3 (Conceptual Relation) of the proposed CG standard (see [4]).

The basic design principle of the operational interface is to exploit some common features of the object paradigm and the CG definitions. A conceptual graph is a bipartite graph consisting of concept and conceptual relation nodes. This definition resembles an object graph, where objects represent the nodes of a CG and object references arcs between the nodes. Therefore it seems feasible to model the nodes of a CG through CORBA objects and the links between the nodes through object references.

This way of modelling a CG through CORBA IDL has several advantages. Since object references are used to connect relation with concept nodes, one CG can be distributed over several hosts. The objects, which denote the nodes of a CG are not required to remain in the same address space, since an object reference can span host boundaries in a heterogeneous environment. Furthermore, a CG does not necessarily need to be sent by value, but rather by reference. Only if the receiving side of an operation actually accesses a node, it will be transferred over the network. This scheme enables a lazy evaluation strategy for CG operations.

It is common to place all IDL definitions related to a particular service in a separate namespace to avoid name conflicts with other applications. Therefore, we assume that the folloing IDL definitions are embraced by an IDL module:

```
module CG {
  // Here come all definitions related to
  // the CG module
};
```

Using the inheritance mechanism of CORBA IDL we first define a common base interface for concepts and conceptual relations. This interface is called `Node` and defined through the following specification:

```
typedef string Label;

interface Node {
  attribute Label type;
};
```

The interface `Node` contains all the definitions which are common for concept and relation nodes. The common property shared by those two types of nodes is the type label. The type label is represented through an attribute of type `Label`. Note that we made `Label` a synonym for `string` through a `typedef` declaration. Next, we define the interface for all concept nodes:

```
interface Concept : Node {
  attribute Label referent;
};
typedef sequence<Concept> ConceptSeq;
```

The interface `Concept` inherits all properties (i.e., attribute `Label`) from interface `Node` and adds a new property, namely an attribute for the referent. The following `typedef` defines an unbounded sequence of concepts. This is necessary for the definition of the relation node:

```
interface Relation : Node {
  attribute ConceptSeq links;
};
```

Just as the interface `Concept`, the interface `Relation` inherits all properties from `Node`. The new property added here is a list of neighboring concept nodes, represented by the attribute `links`. Note that `ConceptSeq` is an ordered sequence. The length of this sequence corresponds with the arity of the relation. The first item in a sequence refers to the concept node pointing towards the relation. The final definition gives the IDL abstraction of a CG:

```
typedef sequence<Node> Graph;
```

A CG is represented by a list of interface `Node`. For brevity reasons the IDL type is called `Graph`. The order of appearance of the individual nodes is of no importance. This data structure suffices to transmit a simple CG over the network. In the following section we provide a little example how this definition might be used in a real application context.

5 Example

Given the basic specifications from the previous section, how does a programmer develop an application using CG? Using the CORBA framework the programmer would have to design the interface to the application to be written based on the definition from the previous section.

E.g., consider a simple application which would offer database functionality for CGs, such as save, restore, etc. The resulting IDL specification would look some like the following:

```
#include "cg.idl"

interface DB {

  typedef string Key;

  exception Duplicate {};
  exception NotFound {};
  exception IllegalKey {};

  Key save( in CG::Graph c ) raises( Duplicate );

  CG::Graph retrieve( in Key k ) raises( NotFound, IllegalKey );

  void delete( in CG::Graph c ) raises( NotFound, IllegalKey );
};
```

This example assumes that the basic definitions for CGs from Section 4 are stored in a file called "`cg.idl`". The definitions are made known through the `#include` directive. Access to the database is defined through interface `DB`. The database stores CGs and assigns a unique key to each CG. The database allows to save, retrieve and delete CGs. If a CG is saved, the database returns a unique key for this CG. The operations `retrieve` and `delete` need a key as an input parameter. Errors are reported through exceptions.

The IDL definition of interface `DB` is all that a client program will need in order to access an implementation. As pointed out before, the language and precise technology that was used to implement the database are irrelevant to the client.

6 Conclusion

The construction of CG–based applications can benfit from the usage of middleware platforms. In this paper we have shown how to translate the basic definition for a conceptual graph to the CORBA IDL. A conceptual graph is represented through a set of CORBA objects which do not need to reside in the same address space. Besides language independance, this has the advantage of supporting lazy evaluation strategies for CG operations.

As a next step the existing CG applications should be re–structured using CORBA (see [1]). By doing so those applications could more easily exploit specific services they offer among each other and to other applications. There are several CORBA implementations available including free ones (see [2]). Applications, which use CORBA as a middleware platform, can be easily accessed in a standardized fashion from any CORBA implementation, such as the one included in the Netscape Communicator.

References

1. CGTOOLS. Conceptual Graphs Tools homepage. http://cs.une.edu.au/∼cgtools/, School of Mathematical and Computer Science, University of New England, Australia, 1997.
2. MICO. A free CORBA 2.0 compliant implementation. http://www.vsb.informatik.uni–frankfurt.de/∼mico, Computer Science Department, University of Frankfurt, 1997.
3. Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 2.0, July 1995.
4. J.F. Sowa. Standardization of Conceptual Graphs. *ANSI Draft*, 1998.