Using a Meta–Notation in a CORBA environment

A. Puder
Deutsche Telekom AG
Research Centre
P.O. Box 100003
64276 Darmstadt, Germany
puder@tzd.telekom.de

K. Römer
Department of Computer Science
University of Frankfurt
60054 Frankfurt, Germany
roemer@cs.uni-frankfurt.de

December 11, 1997

Abstract

In this position statement we demonstrate how a *meta–notation* can be employed in a CORBA environment. For the scope of this paper the term meta–notation refers to a notation which allows the representation of arbitrary syntactical structures[1]. In an earlier project we have shown how to translate CORBA– and DCE–IDL specifications to specifications written an such a meta–notation. Here we demonstrate how a meta–notation can support a generic user interface to CORBAs DII as well. Thus using a meta–notation in a CORBA environment has the benefit of a homogeneous representation of the various information models found in a middleware platform. We use conceptual graphs as the meta–notation; a knowledge–representation technique widely used in AI research.

Keywords: CORBA, DII, generic user interface, conceptual graphs.

1 Introduction

The *Common Object Request Broker Architecture* CORBA offers solutions for implementing distributed applications in a heterogeneous environment. The CORBA specification has gained wide acceptance and has been used in many commercial applications. The potential heterogenity, which needs to be addressed by a CORBA compliant ORB, results in complex middleware platforms. We advocate to use a meta–notation to facilitate the description to cope with some of this complexity. In an earlier project we showed how a meta–notation can be used for the specification of operational interfaces (see [2]).

The outline of this position statement is as follows: in section 2 be provide a brief introduction to the theory of conceptual graphs. In section 3 we describe CORBAs dynamic invocation interface and the problems related to a generic user interface which allows run–time access to this interface. In section 4 we present the anatomy of an operation declaration as defined by the CORBA standard. In section 5 we finally present our solution for a generic user interface to CORBAs dynamic invocation interface based on a meta–notation. We conclude this position paper with a brief description of the prototype implementation.

[1] In that sense the *Backus Naur Form* for example is a meta–notation, since it allows the specification of the grammar of formal languages.

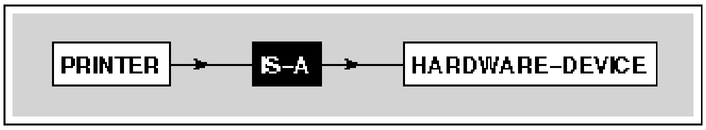

Figure 1: A simple conceptual graph with two concepts and one relation.

2 Conceptual Graphs

The theory of *conceptual graphs* (CG) has been developed to model the semantics of natural language (see [3]). Specifications based on conceptual graphs are therefore intuitive in the sense that there is a close relationship to the way human beings represent and organize their knowledge. From a mathematical point of view a conceptual graph is a finite, connected, directed, bipartite graph. The nodes of the graph are either *concept* or *relation nodes*. Due to the bipartite nature of the graphs, two concept nodes may only be connected via a relation node. A concept node represents either a concrete or an abstract object in the world of discourse whereas a relation nodes defines a context between two or more concepts.

A sample CG is depicted in figure 1. This CG consists of two concepts (white nodes) and one relation (black node). This CG expresses the fact that a printer is a hardware device. The two concepts — `PRINTER` and `HARDWARE-DEVICE` — are placed in a semantical context via the relation `IS-A`. The theory of CGs defines a mapping from conceptual graphs to first–order calculus. This mapping, which is described in [3], would map the CG depicted in figure 1 to the first order formula $\exists x \exists y : \texttt{PRINTER}(x) \wedge \texttt{HARDWARE-DEVICE}(y) \wedge \texttt{IS-A}(x, y)$. As can be seen, the variables x and y form the link between the two concepts via the predicate `IS-A`.

Given a conceptual and relational catalogue, one can express arbitrary knowledge. For this reason the theory of CG represents a *knowledge representation technique*. The work done in [3] focuses on the representation of natural language. We have shown, that with a suitable conceptual and relational catalogue one can translate operational interface specifications to conceptual graphs (see [2]). We have written translators which translate arbitrary DCE and CORBA–IDL specifications to CGs. Thus we have already demonstrated that an implementation of an interface repository, which is based on such a meta–notation, can be used in different middleware platforms. In the following we show how a meta–notation can also be exploited for the construction of a generic user interface to CORBAs *dynamic invocation interface* (DII).

3 Dynamic Invocation Interface

Consider an object–oriented decomposition of a bank application. There will most likely be a set of objects representing *accounts*. Conceptually, an account object keeps track of a balance which can be altered using designated operations. With this informal description one could write an operational interface specification using the notation according to the CORBA–IDL:

```
/* Operational interface for a bank account */
interface Account {
    /* Deposit a certain amount to an account */
    void deposit( in unsigned long amount );
    /* Withdraw a certain amount from an account */
    void withdraw( in unsigned long amount );
    /* Show the current balance of an accout */
    long balance();
```

```
};
```

An object of type `Account` offers three operations at its interface. The operation `deposit()` takes one input parameter which denotes the amount to be deposited. Likewise the second operation `withdraw()` withdraws a specific amount. Finally the third operation `balance()` returns the current balance of the account as a return value. Since the balance can be negative, the type of the result is `long` and not `unsigned long`. A client application written in C++ might for example use this bank account in the following way:

```
Account_ptr acc = ...;  // Obtain a reference to an Account-object

acc->deposit( 100 );
acc->withdraw( 40 );

cout << "Total balance is " << acc->balance() << endl;
```

If we assume that the current balance of the server object was 0 when the variable `acc` was bound with a refence to this object, then this program fragment prints out *"Total balance is 60"*. It should be clear that this program fragment requires the definition of the class `Account_ptr`. This class, which allows a type safe access to a CORBA object implementing the interface `Account`, is generated using an IDL compiler. Thus the type of the operational interface of the server object is known at compile time. But what if we did not know about the interface `Account` at compile–time? The only possible way to access the object in this case is to use CORBA's *dynamic invocation interface* (DII). This interface to an ORB offers the possibility to invoke operation calls whose signature was not known at compile time. The following code excerpt shows the usage of the DII:

```
CORBA::Object_ptr obj = ...;
CORBA::Request_ptr req = obj->_request( "deposit" );
req->add_in_arg( "amount" ) <<= (CORBA::ULong) 100;
req->invoke();
```

Note that the variable `obj` is of type `Object_ptr` and not `Account_ptr`. The code fragment demonstrates how to model the operation call `acc->deposit( 100 )` from the code fragment above. It does not require the `Account_ptr` client stub as in the last example. Despite the generic manner how the operation is invoked, the problem remains how to write a generic user interface to access CORBAs DII. Such an interface would allow a user to invoke arbitrary operations of *a priori* unknown interfaces. Next section gives a brief overview of the specific details of an operation invocation.

4 Anatomy of an operation declaration

Section 3.10 of the CORBA 2.0 specification describes the syntax of an *operation declaration* (see [1]). The syntax is part of the Interface Definition Language (IDL). The grammar presented in that section describes the syntax which induces a formal language. In figure 2 the anatomy of an operation declaration is given, using a graphical representation of the grammar where the arrows denote "consists of" relations. Thus, according to the CORBA 2.0 standard, an operation declaration consists of a result type, an ordered list of parameters and so on. A parameter declaration itself consists of a directional attribute (`IN`, `OUT` or `INOUT`), a parameter type and an identifier.

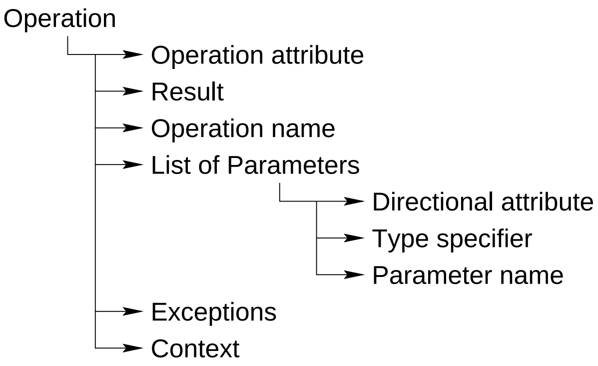

Figure 2: Syntax of an operation declaration.

Note that the "graph" depicted in figure 2 already has some resemblance to a conceptual graph. We propose to model the information pertinent to an operation invocation through a CG. The anatomy of an operation declaration as depicted in figure 2 provides a hint on how to accomplish this task.

5 A generic DII interface

Just consider if we had an application which allowed the browsing of an interface repository. A user would find a suitable interface at *run–time* and decide to invoke operations without having to write a specific client object. What would be nice to have is a *generic client* which could cope with *a priori* unknown operational interfaces. As we have seen in figure 2 and from the discussion of the previous section, an *operation invocation* consists of the following elements:

- a name of the operation
- a return type
- an ordered list of actual parameters

With this "anatomy" of an operation invocation we can assemble a domain–specific conceptual and relational catalogue. We have developed such a catalogue which provides the "vocabulary" to express the information needed for the specification of an operation invocation. The conceptual graph depicted in figure 3 shows how to translate the operation invocation for `deposit( 100 )` using the DII (again concept nodes are denoted by white rectangles and relation nodes by black rectangles). As can be seen, a meta–notation based on CG provides an easy readable, formal specification of an operation invocation. Although it is not shown in this paper, it should be clear that the CG template can be extended arbitrarily to cover the specifics of the CORBA–IDL like complex type definitions of sequences.

6 Reference implementation

We have implemented an interactive conceptual graph editor in Tcl/Tk and Java. We have setup a demonstration which can be accessed via the WWW. In this demonstration the user can download an applet which allows him to build CG as depicted in figure 3[2]. After constructing

[2]The pictures in figure 1 and 3 are actually screenshots taken from our tools.

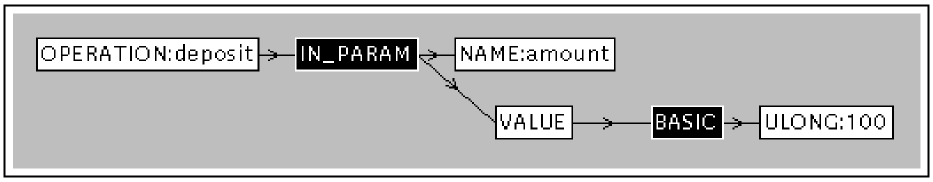

Figure 3: Conceptual graph representing the specification of the operation `deposit()`.

a CG for a method invocation which is supported by point–and–click techniques and context sensitive menus, the user can send this operation invocation specification to our server where it is dispatched to an appropriate object. For the purpose of the demonstration we have a CORBA object running on our machines which implements the operational interface presented in section 3. The demonstration as well as the complete sources of the graphical CG editor are available under the GNU general public license at:

`http://www.vsb.cs.uni-frankfurt.de/~mico/`

References

[1] Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 2.0, July 1995.

[2] A. Puder. Introduction to the AI–Trader Project. http://www.vsb.informatik.uni–frankfurt.de/projects/aitrader/, Computer Science Department, University of Frankfurt, 1995.

[3] J.F. Sowa. *Conceptual Structures, information processing mind and machine.* Addison–Wesley Publishing Company, 1984.