Applying CORBA Standard to Legacy Systems[1]

V. E. Kamensky, A. V. Klimov, S. G. Manzheley, L. B. Solovskaya

The distributed data processing tends to get more and more widespread in the modern computer world. Industrial standards for distributed systems like Object Management Group (OMG) CORBA 2.0 allow creation of open systems in heterogeneous distributed environment. Although CORBA 2.0 standard is primarily intended for new object-oriented applications creation it can be successfully applied to legacy systems distribution as well. One of the problems arising while distributing such systems is a problem of making a specification of a system in OMG Interface Definition Language (IDL).

The paper examines the problems of automatic generation of IDL specification for legacy systems. Attention is also paid to the usage of the generated specification. The paper proposes an extension of IDL to meet the needs of legacy systems usage and a *reverse* compiler from the programming language being in use to IDL.

1. Introduction

The distributed data processing tends to get more and more widespread in the modern computer world. The distribution helps to make a system open for communication with other systems, allows to improve the performance of data processing and to jointly use different machines, operating systems, and programming languages. An interesting initiative is promoted by Object Management Group (OMG), which has proposed a standard of architecture for distributed object system, Object Management Architecture (OMA) [1], and interoperability standard for objects, Common Object Request Broker Architecture, version 2.0 (CORBA) [2, 3]. These documents contain the definition of the OMG Interface Definition Language (IDL) which is intended for system's interface specification, and describe the mappings of IDL to specific programming languages like C, C++, Smalltalk. Besides, the definition of Object Request Broker (ORB) is given. This software component which lies between the network and an application software is to provide the *transparent*[2] interaction of user's objects regardless of their specific location. CORBA allows various object models to coexist in one system and makes it possible for the developer to jointly use various programming languages and environments. This property is referred to as *object model transparency* [4, 5].

The mentioned features of CORBA allow not only to easily create a new object-oriented distributed systems, but also to integrate such systems together as well as with an existing (legacy) object system, however after its modernization. We use here the term *modernization* as denoting an extension of a system towards its openness, which makes it possible for remote objects to use and to be used by a new version of the system. A similar problem of legacy code usage is considered, for example, in Distributed Computing Environment Remote Procedure Call (DCE RPC) [8], where a specific means is provided in DCE IDL and DCE Attribute Configuration Source (ACS) languages for distribution of existing systems. As CORBA does not support the direct

[1] This work is supported by RBRF grant № 96-01-01278 "Research and development of architecture principles of organization of integrated distributed heterogeneous object systems".

[2] The term *transparency* means that the user can invoke methods on both local and remote objects in the same way.

usage of legacy systems, the paper shows how such support can be provided through an extension of OMG IDL.

The usual way to create a CORBA compliant system is to develop it "from scratch": a specification is created first, an implementation second. But implementation for the legacy systems already exists. Therefore, the most important goal of a modernization is to provide the interaction between the legacy and the outer system's object models. Using CORBA for solving this problem will lead to distributable legacy system leaving its source code untouched. Distributed system can be made from the distributable by means of transferring some of its parts into another environment that requires source code modification.

Thus, to modernize an object-oriented legacy system one need to make at least the following steps:

- Select some *object context*[3] from the whole system that will be made open. The purpose for this might be either interaction with another systems or interaction with parts of existing system being moved outwards. Theoretically, the selected context may be the same as the whole interface of the legacy system, but in practice it is more likely to be a part of it. In this paper we consider the case of making a system open, in the sense that its interfaces are exported and other systems' interfaces are imported. When the aim is to make a single system distributed the problems are similar.
- Create in one or another way an IDL specification for the context selected. However, as we will see later it is not always possible.
- Finally, the obtained IDL specification must be implemented using normal CORBA programming technology: one need to compile interface declarations from IDL into the programming language being in use (by means of the IDL compiler), and then implement these interfaces basing upon the source code of the existing legacy system. This is the only step when the system's source code might be modified.

The first step seems to be the most difficult one which can be done only by the developer of a system, i.e. by a person. Any automated tools can assist in solving the problem, but cannot provide a full automation of choosing the object contexts from real legacy systems. We assume that, in one or another way, by method of trials and errors, a context of a legacy system which needs to be specified in IDL is chosen.

The second step is the most interesting regarding the possibility of its automation. The subject of the paper is to consider various ways of generation of the IDL specification from legacy system code, accompanying problems, as well as possibilities of optimization. As a language for writing examples of the source code we will use Protel-2 [7]. Though not widespread, it has much in common with many other procedural object-oriented programming languages. Therefore, the problems with using Protel-2 legacy system are most likely to be the same as with those written in a well-known language like C or C++.

The third and the last step is in general indispensable (regarding implementing the interfaces). However, our purpose will be to minimize the needed efforts as much as possible on the base of reuse of the legacy code.

[3] *Object context* is a set of object interfaces.

The paper is arranged as follows. Section 2 “How to get an IDL specification” is devoted to manual creation of IDL specification. Subsection 2.1 “Manual IDL specification receipt” illustrates some of accompanying problems while subsection 2.2 “Using pragmas in IDL” offers their solutions making use of the standard IDL feature called pragma directive, or simply pragma. A taxonomy of pragmas provided for Protel-2 is given together with examples of their usage. Section 3 “IDL to programming language compiler” proposes a solution of the same problem on the base of an automatic tool, namely, a compiler from the language of legacy system to IDL. Subsection 3.1 “Automatic IDL with pragmas specification generation” explains how such a compiler should do its work and discusses problems of mapping the programming language into IDL (using Protel-2 as an example of the source language). An exhibited list of ambiguities covers essentially all reasonable cases for Protel-2. Finally, subsection 3.2 “Down with pragmas!” makes the next step towards the solution of the problem by means of combining the standard IDL to programming language compiler with an reverse one. It makes it possible to avoid the intermediate step of generating the IDL specification, provided that an implementation for the object context being opened already exists. The latter condition is valid, in particular, for legacy systems.

2. How to get an IDL specification

Before automating given problem solving we will try to solve it by hand. That is, we will deal with it in the same way the developer faced with the problem to make his system open but having no any available tool does.

2.1 Manual IDL specification receipt

The simplest way to create IDL specification for some existing code from the organizational point of view is to create it by hand. Obviously, developers consider this way the most laborious and error prone. Let us consider example.

We assume the legacy system has the following types declared (Protel-2 code is shown):

```
TYPE ProtelInfo STRUCT
      importance $longint,
      valid      bool
ENDSTRUCT;
TYPE legacy_class_ptr PTR TO ANY legacy_class;
TYPE legacy_class CLASS REFINES $OBJECT
            field ProtelInfo
      OPERATIONS
            field_set METHOD (UPDATES; REF x ProtelInfo),
            field_get METHOD (REF) RETURNS ProtelInfo
ENDCLASS legacy_class;
```

As said above the choice of object contexts which should be opened is non-trivial problem. Not all objects of legacy system can be exported. For example, many systems especially real-time systems have groups of loosely coupled objects. Externalization of one object would lead to broken links (for example, temporal ones) with other objects and require system's architecture modification. Sometimes it is impossible to export an object due to techniques used in its implementation (for example, all fields and methods of the object might be declared accessible only by object itself and/or its ancestors). Here and below we assume that some object context to be opened is already chosen.

IDL specification corresponding to the example might look like the following:

```
struct IDLInfo {
        long            importance;
        boolean         valid;
};
interface legacy {
        void            field_set(in IDLInfo info);
        IDLInfo         field_get();
};
```

The developer will confront with some problems when he/she tries to write the specification and use standard IDL compiler in order to generate required code on the target programming language. Namely:

1. The types IDLInfo and legacy will be declared as in IDL specification (and, therefore, in the generated code), and in the legacy system. It is the same types. Because of this the developer will be forced to delete the doubled declarations by hand.
2. Logically, one need to leave legacy system's declarations of types untouched and extend generated code with references to legacy declarations. Obviously, it should be done by hand. Also the `uses` list of generated by the IDL compiler module must be extended with those module names that contain declared types.
3. At last, the names generated from IDL types will differ from existing ones. Therefore one need to correct these names in all procedures and types generated by the IDL compiler. Again, this work should be done by hand.

The programmer may find convenient to declare this or that IDL type under different name. In this case he/she should rename generated by the IDL compiler type names (or declare synonyms) as in places of the declaration, and in places of use.

The example also demonstrates one of ambiguities arising when reverse mapping of the programming language to IDL is done. In our case the access to attribute `field` of the `legacy_class` in Protel-2 is done with the help of two methods -- `field_set` and `field_get`. Both are visible to the user. In IDL specification similar methods are declared. But, as IDL attribute is simply a pair of methods, it would be possible to declare one attribute `field` in IDL specification, and the effect would be the same[4].

[4] It should be said that in this case one need to replace default method names with existing ones in generated by IDL compiler code.

Surely, all above problems can be solved by hand editing of IDL generated code. But we strive to save the developer from this routine work and to charge it to the computer. Especially it is actual because generated code is modified, that means all edits made will be lost if regeneration required. It is logically to edit only source code charging all rest to automatic tools. For this purpose we are making first step.

2.2 Using pragmas in IDL

The disadvantages, bounded up with the use of IDL for the existing code, can be eliminated. The IDL language has built-in means for system-dependent information indication to the compiler -- pragmas [2]. By use of pragmas, developer could refer to declarations of existing in legacy system types, could substitute names, could use another IDL specification already compiled. Pragmas are meaningful only for those compilers that know how to interpret them. Standard IDL compiler simply ignores unknown pragmas.

Generally speaking, IDL pragma is a single line of text beginning with #pragma keyword. We introduced the following classes of pragmas in order to solve the given problem -- the use of existing code in legacy systems [6]. (Further classes of pragmas for the specific programming language Protel-2 are listed. However, in the same way IDL compiler to any object-oriented programming language can be extended):

- pragmas destined for inclusion of IDL or Protel-2 code into IDL specification or generated Protel-2 modules respectively. Pragmas of this class are called includeProtel, includes, inline, useProtel, uses;
- pragmas allowing change of default IDL compiler generated names: prefix, replaceid, filename;
- pragmas stating the IDL specification of a given interface will be used as client or server only: noclient, noserver;
- finally, pragmas allowing replacement of IDL compiler generated type, class or attribute of class with existing one (for example, from legacy system): maptype, mapinterface, mapattribute.

The most important from the existing code use point of view is last class of pragmas. All its pragmas work in the same way. When IDL compiler finds such pragma in IDL specification it does not generate default type, class, or attribute, but rather uses pointed out Protel-2 type, class, or attribute respectively. We can rewrite IDL specification for the example as follows:

```
#pragma Protel useProtel legmod
#pragma Protel maptype IDLInfo as ProtelInfo
struct IDLInfo {
  long          importance;
  boolean       valid;
};
#pragma Protel mapinterface legacy as legacy_class with legacy_class_ptr
interface legacy {
  #pragma Protel mapattribute field asmethods field_get field_set
```

```
        attribute IDLInfo field;
    };
```

As one can see four pragmas have been added. All of them have common syntax:

- the keyword #pragma;
- the word Protel (it points the pragma relates to the IDL to Protel-2 compiler);
- the type of pragma -- one word (example includes useProtel, maptype, and so on);
- finally, parameters of pragma. Every type has its own parameters.

The first of pragmas (useProtel) forces IDL compiler to add listed names to the `uses` list of generated module. It makes possible to use declarations from the modules mentioned in Protel program[5].

The second pragma (maptype) instructs compiler the type IDLInfo is declared in Protel as `ProtelInfo`. Thus, compiler should not generate the type declaration, and should produce only procedures necessary to use it. At that, all auxiliary types and procedures will have type pointed out by pragma (but not default type generated by compiler). Type `ProtelInfo` should be declared or should be visible (accordingly to Protel-2 visibility rules) in module `legmod`. For this purpose the latter have been refereed. The structure of type declared must be analogous to the structure of existing type. The pragma maptype can be applied only for those types that are not classes.

Pragma mapinterface is applied in mapping of IDL classes onto existing Protel-2 ones. Pragma has three parameters. Firstly, name of IDL class. Secondly, name of Protel-2 class that will correspond to the IDL class. And, thirdly, type name that is a pointer to the declared class. Often not only objects of a given class are used, but also pointers to the objects are (generated by IDL compiler code uses for the most part pointers), and quite often respective types are declared in the legacy system[6]. Because of this programmer should refer to the existing or declared pointer type in the third parameter. If pragma mapinterface is applied to the IDL interface, IDL compiler does not produce corresponding Protel-2 class and stated in pragma Protel-2 type itself and pointer type is used in all auxiliary types and procedures.

There are constraints on the use of pragma mapinterface. Let assume two IDL interfaces A and B is declared and B inherits from A. Pragma mapinterface is applied to the interface B. If interface A is mapped by one or another way to Protel-2 class `Ap`, then interface B must be mapped to Protel-2 class `Bp` that inherits from `Ap`. Application of pragma mapinterface to IDL interface that does not inherit from another interface involves the following requirement: corresponding Protel-2 class must inherit from `CORBA_Object` (it is a base class for all objects declared in IDL).

The last pragma (mapattribute) instructs IDL compiler that methods `field_get` and `field_set` will be used to access to attribute `field`. This pragma is mainly applied when existing code has access methods for attribute and one need to utilize them. Pragma mapattribute can be employed without pragma mapinterface. This leads

[5] In our case, declarations from the module `legmod` become accessible in compiler generated module.

[6] If pointer type is not declared pragma inline can be used. It makes possible to insert code fragment into compiler generated module.

to use of another names in generated code and even another attributes instead of methods generated by default.

Besides listed pragmas we introduced other ones that make easier to use legacy code. They are enumerated below with brief comments:

- includeProtel -- includes Protel-2 file into interface section of generated module. This pragma is convenient for insertion of user-defined types and/or procedures;
- includes -- includes IDL file into IDL specification. This pragma is used for choice of one of possible interpretations of #include directive in IDL. It is spread to file stated in it;
- uses -- another possible interpretation of #include in IDL. One argument of pragma is a Protel-2 file name that contains IDL specification compiled. At that, when an IDL specification is compiled another can be used, but source code access for the latter is not necessary. Such scheme of work agrees with one found in Protel for programs development;
- inline -- allows to insert small fragment (one line) of Protel-2 code into IDL compiler generated interface section;
- prefix -- makes it possible to change prefix for generated nested names in Protel-2;
- replaceid -- replaces IDL name with the given Protel-2 one. In contrast to maptype and mapinterface pragmas only names are substituted. All types that IDL compiler must generate are generated;
- filename -- changes file names for generated by IDL compiler module;
- noclient and noserver -- say to IDL compiler that interface specification will be used only as client or only as server one. Because of this compiler can skip some code generation.

The extension of IDL compiler with listed pragmas interpretation will make life of developer easier when he/she is going to use legacy code. Problems mentioned in section 2.1 “Manual IDL specification receipt” in fact are solved when IDL specification is created. Advantages and disadvantages of proposed approach are obvious. From one side one need not to manually edit compiler generated code. From another side IDL specification should be extended with pragmas that makes programmer to do some work.

The next step in given problem solving is to save developer from IDL specification creation by hand.

3. IDL to programming language compiler

The considered way of IDL specification's generation is very laborious. The same problem can be solved by means of some automatic tool, capable to generate IDL specification for the given interface declarations written in some programming language. It is logically to call such tool as compiler from the programming language to the IDL (further we refer it simply “compiler”).

3.1 Automatic IDL with pragmas specification generation

Compiler input is a set of files with used programming language text that contains object context to be opened definition. That is a set of constants, types, and classes. All other constructions from the source files are simply ignored since they can not be represented in IDL specification. Compiler output is an IDL specification extended with pragmas from the section 2.2 "Using pragmas in IDL" of the same object context. The use of pragmas is necessary since we need to reference existing code of legacy system in the IDL specification.

General idea of the compiler is simple enough. Construction declared in programming language is put in correspondence IDL construction: structure to structure, array to array, and so on.

In order to make compiler to generate code that is correct not only syntactically, but semantically too, it is necessary to make it to resolve arising ambiguities. The case is, usually several IDL specifications can be generated for a given one written in a programming language. It is explained with not only more wide significant means, virtually, of any programming language comparing with IDL, but with ambiguities arising during reverse mapping too. For example, IDL type union need discriminant that reveals type of contents. Protel-2 type "`struct with ovly`" (analog of IDL union) does not contain discriminant. Therefore compiler runs into the problem of choice of it. It is clear whatever compiler chooses its conclusion might be semantically incorrect and only programmer can help it.

Another significant aspect of systems development is a pragmatic one. Compiler proposed ambiguity resolution might be syntactically and semantically correct, but it does not meet desirable productivity or some other criteria.

Further the most important and typical problems of reverse mapping (programming language to IDL by the example of Protel-2) are discussed.

3.1.1 The problem of choice of discriminant for IDL union type

One of frequently used construction of programming languages is a union type or its analog. Structure with overlay corresponds to the union in Protel-2:

```
TYPE U STRUCT
        OVLY {0 TO 2}
        {0}:  x     $longint
        {1}:  y     {0 TO 255}
        {2}:  z     bool
        ENDOVLY
ENDSTRUCT;
```

IDL has similar type, but it contains discriminant in contrast to some programming languages (like C or Protel). Discriminant is a such tag that basing on it value one can decide what type a content has. For example, Protel example shown above might looks in IDL like follows:

```
union U switch (short) {
        case 0: long          x;
        case 1: char        y;
        case 2: boolean     z;
};
```

Although both declarations seem to be identical on the first glance they have different semantics. Only type of discriminant is declared in Protel and programmer can not use its value in his/her program -- it does not simply exist. Thus, mapping Protel in IDL the problem of choice of discriminant arises. Compiler should choose variable or procedure that computes discriminant values. In the case when such computation is impossible the decision is made that type can not be exported.

Discussed example of Protel structure with overlay illustrates yet another feature of the language. Overlay can be declared only within structure, but not on the top level, besides several overlays can be declared within the same structure. All of them are anonymous types. Structure having several union type fields in IDL should correspond to the structure with several overlays in Protel. Since IDL type can not be declared within another IDL type declaration (for example, union type within struct), it is necessary to generate auxiliary types. Surely, names of these types were not used in Protel program and default names can be generated in IDL. From the other side it is desirable to allow user to give instructions what names to use for these types.

3.1.2 The problem of pointers in Protel-2

As in many other programming languages, programmer can use pointers in Protel-2. From the implementation point of view Protel-2 pointer is a memory cell address that contains value of a given type. It is true not only for pointers to base and aggregate types, but for the pointers to classes too. Obviously value of a pointer itself makes sense only in that context, that address space, that it was created in.

The following example shows parameter of a method declared as a pointer:

```
m METHOD (REF; p PTR TO T)
```

IDL language does not have such notion as pointer, therefore the problem of mapping of legacy system objects and types containing pointers arises.

General approach to the problem solution consists in pointer replacement with some IDL interface. For a given example Protel-2 to IDL compiler might generate the following auxiliary interface and method declaration:

```
interface T_wrapper {
        attribute T value;
};
…
void m (in T_wrapper p);
```

Object of type T_wrapper (that is, object reference) is passed to method invoked and each access to this value will lead to remote invocation. Attribute value (in fact a pair of methods) give access to the value of variable. Type `T` surely must be exported. If `T` is a class then compiler needs not to generate auxiliary interface and respective IDL method can be declared as follows:

```
void m (in T p);
```

The problem of pointers is solved in the same way for types declared as pointers and aggregate types that contain pointers within themselves. Auxiliary interface is produced for every pointer declaration (so, this interfaces can be nested). Compiler can choose names of such interfaces and generate default ones, from another side user can do it.

Under some circumstances auxiliary interface requires method of explicit object of class creation. Usually object creation method is a class method. The problem of class methods export is discussed in section 3.1.3 "Class methods in Protel-2".

Though the problem of pointer's solution been considered is correct, it might be found inefficient in some cases. It is explained the access to the pointed value is done with use of ORB. Sometimes compiler and/or developer can optimize generated code, to say pointers can be eliminated without sacrificing semantics and can be replaced with the pointed type in IDL. (For example, this is the case when structure is a field of another structure, but it is represented as a pointer within the latter). In general, the decision whether pointer should be eliminated or not can be taken only by developer of a system.

3.1.3 Class methods in Protel-2

Many object-oriented programming languages allow declaration of so called "class" methods in classes, that is methods that might be invoked not on the class instance, but rather on the class itself. Protel-2 is one of such languages. IDL does not support class methods and they might be contained in IDL specification only as normal methods. In order to invoke any method (using standard mapping to target programming language [6]) one must have class instance, that is object, and therefore object reference. If class contains class methods but there are no instances of this class created yet, it is impossible to invoke class method.

The following solution for the legacy systems can be proposed. Compiler could create auxiliary class in IDL specification that contains only class methods from the class considered. Implementation section of IDL compiler generated module should be extended with procedure that creates and initializes only one object of auxiliary class. Thus, ORB will be responsible for object reference creation and user will be capable to invoke class methods on auxiliary class even the objects of the primary class are not created yet.

3.1.4 Updates parameters in procedures

Protel-2 language, as many others, though, offers two modes for parameter passing in procedures. They are passing by value and passing by reference. When passing by reference mode is used one need to point out whether parameter is modified within procedure or not. Protel-2 uses two keywords for this purpose -- `updates` and `ref` respectively. If some parameter in the legacy system procedure is declared with `updates` keyword, the question arises: what passing mode should correspond to in IDL?

It is known the IDL supports three passing modes for parameters: in, out and inout. One can give examples where the two last modes can be used for Protel `updates` parameters.

It is logically to assume the inout mode must be used in IDL for all `updates` parameters. This case is typical, but the following example shows that such practice can be dangerous sometimes. Protel-2 has descriptor type. In fact it is a variable length array. Descriptor contains pointer to data and number of elements. Let assume some procedure has `updates` parameter with descriptor type. IDL specification states this parameter passing mode is inout. The procedure does not use incoming value of parameter and simply overwrites it with outcome. Programmer that knows this behavior passes non-initialized descriptor to the procedure. Then ORB packing parameters for transfer over network tries to fetch descriptor pointer resulting in exception.

Thus compiler should analyze source code in order to know whether `updates` parameters are used within procedure bodies. If it does not sure one of two cases take place, it must stop with error message. Default case (inout mode is used always) application is possible, but it is dangerous and user must check compliance of generated IDL specification with existing code.

Another solution of the problem of use as `ref` and `updates` parameters in existing program consist in substitution of the following declarations:

```
m1 METHOD (REF; REF P T1, UPDATES Q T2)
```

with

```
m1 METHOD (REF; P PTR TO VAL T1, Q PTR TO T2)
```

At that `PTR` keyword must be put before all actual parameters in all procedure invocations. So the problem reduces to the problem of pointer mapping considered above (refer to section 3.1.2 "The problem of pointers in Protel-2"). This solution is more reliable (though it is less efficient) and might be used by compiler working in batch mode. It is clear that way choice is a human prerogative.

3.1.5 The problem of attributes in Protel-2

On of typical situations is the case when class contains not only public methods, but also public attributes. For the legacy system the problem is to generate respective IDL specification, that means the class must be exported along with its attributes. For this purpose one can take advantage of mapattribute pragma in Protel-2 (refer to 2.2 "Using pragmas in IDL"). But then IDL compiler could not generate Protel code suitable for the specification. The case is a stub (a mediator between client of the class and its implementation [2]) is produced for every IDL interface. Stub is a class that inherits from the abstract class corresponding to the declared in IDL and overrides all its methods for remote calls processing [6]. However, Protel-2 language does not support attributes overriding in classes and so IDL compiler can not create stub.

The problem solution is to necessarily generate noclient pragma by Protel-2 to IDL compiler. It cancels client code production. That is, those classes of legacy system that contain public attributes can be used only as servers.

3.1.6 Alternative mappings for some types

There are several alternative mappings to IDL for some types of programming language. The choice of alternative might be dictated by convenience, efficiency, and other considerations. We will give some examples illustrating the most typical ambiguities arising when Protel-2 mapping is the case.

Using Protel programmer can declare type-interval having integer values from the given range. Any integer IDL type can be put in to accordance to it. It is enough all values of the range are represented. Thus, for the interval `{0 TO 2}` in Protel one can use types octet, char, short, unsigned short and so on in IDL. Your mileage may vary.

Another example is linked with type `table` (array) in Protel. It is logically to represent it by array in IDL too. But there is another possibility. If we want to pass array not by value (all elements will be copied from the caller to callee side), but rather by reference, we could put in accordance to it in IDL an interface[7]. In this case object reference is passed on method invocation. When elements of array are accessed remote invocations arise. This approach is expedient in the case of large arrays when copying is very expensive and caller object accesses a few elements. Generally speaking, every type of programming language can be mapped on the most appropriate in IDL or on the interface.

Finally, the last example is an ambiguity for class mapping from the used programming language to IDL. Standard variant consists in representing classes by interfaces in IDL. As an alternative one can propose use of IDL structures for this purpose. In the case when existing class contains only attributes and no methods this way of mapping makes it possible to pass objects not only by reference, but also by value that might be convenient.

3.1.7 Dialog with compiler

The list of problems examined makes sense that not all object contexts processed by IDL to programming language compiler might be opened. In some situations, compiler can produce warnings and errors (refer, for example, to section 3.1.1 “The problem of choice of discriminant for IDL union type”). When this is the case, the developer should revise source specification.

Ambiguity resolution makes compiler to deeply analyze the source code. The way an ambiguity will be resolved might be based on choice of default case. However, it is convenient and sometimes necessary to allow the user to make the decision himself/herself. For this purpose one can take advantage of the following two approaches.

The first approach is to use interactive compiler when dialog between the program and user occurs during compilation process (it may essentially use graphical user interface). Compiler might support different levels of interactivity. Beginning with fully automatic level, when all decisions are made without user help, and ending with fully interactive one, when any problem (ambiguity resolution, name choice for auxiliary type, and so on) stops compilation process and causes user intervention. In any case, after use of compiler programmer should check resulting IDL specification for conformity with his/her object context. And it might be done only by hand.

The second approach consists in batch mode implementation. User should in one or another way specify source object context and his/her decisions regarding ambiguity resolution. This problem should be solved differently for different programming languages. In the Protel-2 case it is possible to place all necessary information into auxiliary file (written in Protel-2 again). Programmer could add supplementary code in this file too.

[7] The element access method must be declared in to it.

Both approaches have advantages and disadvantages. However, by virtue of fact interactive compilers are hard to use and usually they are not settled down, we choose the second approach.

3.2 Down with pragmas!

Since the process of IDL specification generation is automated we can try now to optimize it. Let consider the typical sequence of steps the developer using CORBA standard should undertake in order to modernize a legacy system.

The first step after object context choice in legacy system is use of programming language to IDL compiler that generates IDL specification. Then this specification must be compiled with the IDL compiler resulting in modules written in programming language. If the latter and its environment does not coincide with those applied in legacy system then this sequence of steps is the only possible. In other case when generated code will use the same programming language as legacy system this sequence is obviously redundant. We propose to combine two compilers -- programming language to IDL and IDL to programming language, in one. Surely, the possibility to use only one of them must be available.

In this way one need not to use pragmas (section 2.2 "Using pragmas in IDL"). Indeed, pragmas will be useless for IDL to another programming language compiler and will be ignored. For the exploited programming language code will be generated directly (for example, from the Protel-2 legacy code all necessary types, procedures, classes, and their implementations will be generated using Protel-2 again).

Thus, combined compiler has specification of object context in exploited programming language as an input, and IDL specification[8] and modules the standard IDL compiler produces as an output. The information required by compiler to define source object context, to choose alternatives, and so on, can be provided in dialog (in interactive form) or as additional files (refer to section 3.1.7 "Dialog with compiler"). For Protel-2 language additional files might be written, for example, in Protel-2 again. They might be designed as interface sections containing pseudocomments with all required information. Module produced by IDL compiler [6] might extend these interface sections resulting in a single module. So developer can easily use both his/her own code specifying object context and code generated by IDL compiler. He/she needs only insert name of module in `uses` list of using modules. If implementation required to be extended then generated module can add implementation section.

4. Conclusion

The paper examines some problems arising when CORBA standard is applied to legacy systems. They are automatic generation of IDL specification for existing code and use of specification got for modernized system implementation. These problems can be solved by extending OMG IDL and by use of tool we called "reverse" compiler (the compiler from the used programming language to IDL). The extension discussed uses standard means of IDL -- pragmas. Besides, compiler that combines functions of normal IDL compiler and "reverse" compiler is proposed as an optimization.

[8] IDL specification is necessary for those parts of a system that will be externalized from the legacy environment. It will be also used by third-party objects.

For now the work of extending IDL compiler with the set of pragmas that supports legacy code use is done. Besides, the compiler from extended IDL is implemented. “Reverse” compiler and problems concerned with it are under development.

Acknowledgment

We are expressing our sincere gratitude to Victor Petrovitch Ivannikov for draft reading and valuable remarks he was made. We are also thankful to him for insistence without which the paper hardly will be published.

References

1. The Object Management Architecture Guide. Object Management Group. 1990. Revised 1995.
2. The Common Object Request Broker: Architecture and Specification. Revision 2.0. Object Management Group. 1995.
3. J. Siegel. CORBA -- Fundamentals and Programming. John Wiley & Sons. 1996.
4. S. Danforth, I. R. Forman. Reflections on Metaclasses Programming in SOM. OOPSLA’94 Conference Proceedings, 1994.
5. I. R. Forman, S. Danforth, and H. Madduri. Composition of Before/After Metaclasses in SOM. OOPSLA’94 Conference Proceedings, 1994.
6. A. V. Klimov, M. R. Kovtun, and S. G. Manzheley. Protel-2 ORB System Guide. Technical Documentation. Institute for System Programming, Nortel. 1996.
7. M. Turnbull. Protel-2 Reference Manual. For Compiler Version P2BRxx. Bell Northern Research. 1995.
8. AES/Distributed Computing Remote Procedure Call. Revision B. Open Software Foundation. 1995.