Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 285–292.

Supporting parametric polymorphism in CORBA IDL*

Zoltán Porkoláb[a], Roland Király[a], Ilir Kurti[b]

[a]Department of Programming Languages and Compilers,
Eötvös Loránd University, Budapest, Hungary
e-mail: {gsd,kiralyroland}@inf.elte.hu

[b]Universiteti Aleksander Moisiu, Durrës, Albania
e-mail: ikurti@uamd.edu.al

Abstract

The Common Object Request Broker Architecture (CORBA) is a widely accepted standard, that enables software components written in multiple programming languages, and running on multiple platforms to interoperate. Language independency is achieved through the concept of language mapping. CORBA uses the interface definition language (IDL), to specify the interfaces between components. Also, the IDL compiler generates stub and skeleton code that the client and servant links to.

Parametric polymorphism is an emerging programming technique that focuses on abstracting types to collect commonalities in data structures and operations via type parameters. Parametric polymorphism may radically reduce code-repetition thus improves the qulity of the code. It is an unfortunate case that CORBA IDL does not support parametric polymorphism, which causes unnecessary code repetition both in declaration level in IDL and implementation level in stubs and skeletons. In this article we discuss the advantages of introducing templates – a C++ style solution for parametric polymorphism – into the IDL language.

Keywords: CORBA, IDL, Parametric polymorphism

MSC: 68M14 Distributed systems

1. Introduction

The Common Object Request Broker Architecture (CORBA) is a widely accepted standard defined by the Object Management Group (OMG) [11] that enables software components written in multiple computer languages and running on

*Supported by Stiftung Aktion Österreich-Ungarn, Pr.N: 66öu2.

multiple computers to work together [4]. CORBA provides a framework for the development and execution of distributed applications and components as well as some tools to facilitate the implementation of those interfaces using the developer's choice of languages. In addition, the OMG specifies a wealth of standard services, such as directory and naming services, persistent object services, and transaction services. Each of these services is defined in a CORBA-compliant manner, so they are available to all CORBA applications. Finally, CORBA provides all the "plumbing" that allows various components of an application to communicate with each other.

Language independence is a very important feature of the CORBA architecture. Because CORBA does not dictate a particular language to use, it gives application developers the freedom to choose the language that best suits the needs of their applications. Taking this freedom one step further, developers can also choose multiple languages for various components of an application. For instance, the client components of an application might be implemented in Java as an applet, which ensures that the clients can run on virtually any plattform that supports standard HTTP browsers. The server components of that application in the same time might be implemented in C++ for high performance purposes.

2. CORBA IDL

Language independency is achieved through the concept of language mapping. CORBA uses the Interface Definition Language (IDL) to specify the interfaces between components. An object interface indicates the operations the object supports, but not how they are implemented. IDL is purely declarative, that is, in IDL there is no way to declare object state and algorithms.

Language mapping is done by IDL compiler, a tool which converts IDL declarations to their associated representations in the target language. IDL defines language bindings for many different programming languages in a standardized way. Currently there exist mapping to C, C++, Java, Ada, COBOL, Smalltalk, Objective C, and Lisp languages. The definition of IDL is programming language neutral, however CORBA objects exhibit many features and traits of other object-oriented systems, including interface inheritance and polymorphism. What makes CORBA even more interesting is that it provides this capability even when used with non object-oriented languages such as C and COBOL, although CORBA maps particularly well to object-oriented languages like C++ and Java.

The IDL generates stub and skeleton code that the client and servant links to. Client-side stubs represent the CORBA object locally in the actual programming language. The generated code also represents in the language all of the IDL interfaces and data types used to issue requests. The client code thus depends only on the generated client-side stub code. In addition to the files generated for a client, it also generates skeleton code for the object implementation. A skeleton is the entry point into the distributed object. It unmarshals the incoming data, calls the method implementing the operation being requested, and returns the marshaled

results. Thus, the object developer can focus on providing the implementation of the IDL interface.

The CORBA IDL was designed to express common language features without regards to any particular programming languages. IDL is able to describe modules as a collection of object interfaces; attributes and parameterized operations, and exceptions raized by certain operations.

The IDL data types are basic data types, (like `string`, `long`, `short` ...), constructed data types (like `struct`, `union`, `enum`, `sequence` ...), a dynamically typed value (`any`), and object references.

The following figure demonstrates the structure of IDL definitions:

```
module <id>
{
    <type_decl>;
    <const_decl>;
    <exception_decl>;

    interface <id> [:<base>]
    {
        <type_decl>;
        <const_decl>;
        <exception_decl>;

        <attrib>;

        [<mode>] <id> (<params>)
            [raises <exception>] [context];
    };
    // other interfaces ...
};
```

3. Parametric polymorphism in IDL

Parametric polymorphism – abstracting commonalities on data types and operations – is a widely accepted programming paradigm today [3]. Appeared first in earlier dynamic programming languages, the idea won greater attention in the languages ADA [1], and Eiffel [5] where generics were important elements for abstraction. In the C++ programming language template is a fundamental language construction [8]. Recently most object-oriented languages has some language elements implementing parametric polymorphism. In Java, erlier attempts from Pizza [7] to Generic Java [2] has lead to Java Generics in language verison 5. [9]. In C#, generics also part of the current specification [10].

However, there is two completely different way to implement parametric polymorphism. ADA, Eiffel and C++ use instantiation: every time a client refers a

template with a new type parameter a new instance is created. This automatic code generation is called instantiation. In that method all the type parameter occurances are served by different concrete classes generated from the template. On the other hand, Java and C# use type erasure technique. Here no separate instances are created, all type parameter occurances are served by the same single code – typically working on the root class of class hierarchy. In Java, this class is Object.

CORBA and generic programming have already met at the implementational level. Most modern implementations of CORBA IDL mappings are heavily based on parametric polymorphism programming techniques. The IDL C++ mapping generates template classes for reducing code repetition.

However, at the interface definition level the programmer is still unable the define generic constructions. Introduction template machanism into IDL language will highly reduce code duplication, therefore would improve the quality of the code.

3.1. IDL templates

To introduce parametric polymorphism to CORBA IDL the most straitforward way is to follow the C++ template mechanism, both in syntax and in semantics. In this model we require only minor changes in the IDL languager. The main concept here is the immediate instantiation of templated IDL definitions: changing the placeholder identifier to the concrete type used in the place the template is referred creates concrete instantiations. Such a mapping between templated and ordinary IDL definitions can be done by a macro-like precompiler. The result is a non templated IDL file. Language bindings and IDL compilers to create the stub and skeleton codes are not affected.

A general definition of the templates of IDL language can be the following:

```
template <typename identifier>
    interface|function|other_elements;
```

The following is an example for a templated IDL code.

```
template <typename T>
interface stack
{
    void   push (in T v2);
    T      pop();
    short  size();
};
```

The code is parsed and syntactically checked by the templated IDL precompiler IDL<T>. However, similarly to C++ template instantiation, no other action is taken place until some other part of the IDL does not explicitelly refer. Such a code snippet could be the following:

```
typedef stack<long> long_stack;

interface calculator
{
    stack<short>  expresson;
};

interface non_templated
{
    void evaluate(in stack<float> expr);
};
```

In the first case, typedef is used to name a new type long_stack as a stack instantiated with type parameter long. This is similar to the regular usage of typedef regarding with expressions like sequence<long>.

In the second case a new interface calculator declared with a stack as an attribute. Here we instantiate an unnamed instance of stack interface with parameter type short. This is a different interface from the previous instance stack<long>.

Instantiation may happen when the new interface is referred in a rather implicite way. In the third example stack<float> is mentioned as an argument of a method. This also causes instantiation. The result is equivalent to the following:

```
interface __T1__
{
    void   push (in long v2);
    long   pop();
    short  size();
};

interface __T2__
{
    void   push (in long v2);
    long   pop();
    short  size();
};

interface __T3__
{
    void   push (in long v2);
    long   pop();
    short  size();
};

typedef __T1__ long_stack;
```

```
interface calculator
{
    __T2__  expresson;
};

interface non_templated
{
    void evaluate(in __T3__ expr);
};
```

Using and compiling templates in IDL is easier than in other languages, because IDL does not contain the function bodies, just the method header definitions, so the expressions and their evaluations require a bit more relaxed examination. However, this is still must not taken as a simple macro replacement. Strict syntax checking, name lookup, and controlling previous instantiations must be taken place.

The main advantage of this solution is that we can translate the templated IDL into the ordinary IDL declarations without the neccessity to bother the IDL bindings for different languages. Existing IDL compilers can be used to generate stub and skeleton codes.

3.2. Generating generic stub and skeleton

When we extracting templated IDL code the result is a non-templated IDL definition ready to processing with ordinary IDL compilers. However, in this model we lose the opportunity to further exploit the commonality of the templated definitions. All the further processing of the generated interfaces happens independently.

In programming languages directly supporting parametric polymorphism we can generate the stub and skeleton code directly from the original templated IDL definitions. Hence the size of the generated source and in some cases the executable can be radically reduced.

For languages like ADA and C++ generation of the templated proxy types, holder and helpers are straitforward [6]. In C# and Java we have to fix the problems arising from the nature of type erasure. In the case of languages where no support for parametric polymorphism exists, we can use the method described in the previous subsection.

4. Conclusion

The introduction of parametric polymorphism to CORBA Interface Definition Language offers the opportunity to create abstractions over type parameterized data structures, operations and other language elements. Thus the designer of distributed systems creates clearer interfaces and improves the quality of the code.

There are different levels of implementing parametric polymorphism in IDL mapping. The most natural and portable way is the template instantiation method

at IDL level. In this case templated IDL definitions are unfolded to non-templated standard IDL definitions. Hence, maximal portability is achieved but we lose the opportunity to further exploit the commonality in these definitions.

An advanced solution is possible for languages supporting parametric polymorphism. In this case stub and skeleton generation could utilize the commonality in IDL declarations, therefore more advanced templated code helps the implementor

References

[1] Barnes, J., Programming in Ada 95, 2nd Ed. *Addison-Wesley*, ISBN-10: 0201342936, (June 1998).

[2] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., Making the Future Safe for the Past: Adding Genericity to the Java Programming Language, *OOPSLA'98*, ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, BC, (1998), 183–200.

[3] Cardelli, L., Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, Vol. 17, No. 4, (December 1985), 471–522.

[4] Harkey, D., Orfali, R., Client/Server Programming with Java and CORBA, 2nd Ed, *John Wiley and Sons*, ISBN-10: 047124578X, (March 1998).

[5] Meyer, B., Eiffel: The Language, *Prentice Hall PTR*, ISBN-10: 0132479257, (October 1991).

[6] Henning, M., Vinoski, S., Advanced CORBA Programming with C++, *Addison-Wesley Professional*, ISBN-10: 0201379279, (February 1999).

[7] Odersky, M., Wadler, P., Pizza into Java: Translating Theory into Practice, *POPL'97*, Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, (1997), 146–159.

[8] Stroustrup, B., The C++ Programming Language (Special 3rd Edition), *Addison-Wesley Professional*, ISBN-10: 0201700735, (February 2000).

[9] Torgersen, M., Ernst, E., Hansen, C. P., Ahé, P., Bracha, G., Gafter, N. M., Adding Wildcards to the Java Programming Language, *Journal of Object Technology* 3(11), (2004), 97–116.

[10] Standard ECMA-334 C# Language Specification, 4th Ed. June 2006, *ISO/IEC 23270:2006*.

[11] IDL Syntax and Semantics, Chapter 3 of The Common Object Request Broker: Architecture and Specification, `http://www.omg.org`

Zoltán Porkoláb, Roland Király
Pázmány Péter sétány 1/C
H-1117 Budapest, Hungary

Ilir Kurti
Lagja 1, Rr. Currilave
Durrës, Albania