Customizing IDL Mappings and ORB Protocols

Girish Welling and Maximilian Ott

C&C Research Laboratories, NEC-USA, Inc.
4 Independence Way, Princeton, NJ 08540, USA
{welling,max}@ccrl.nj.nec.com

Abstract. Current mappings of IDL to implementation languages such as C++ or Java use CORBA specific data-types, which makes it imperative for an object implementation to be CORBA-compliant. While being completely CORBA-compliant ensures portability *and* interoperability, several classes of enterprise applications may *only* require interoperability with other CORBA applications. Other applications may be constrained by such factors as a large existing code-base or a widely used communication protocol. In many cases, these applications can benefit from the concise expressiveness of IDL without committing to the overhead of using a general-purpose CORBA ORB. To aid this process, we propose a new approach to ORB design where the IDL mapping and ORB protocol is completely configurable. As a motivation, we present our use of IDL in the development of a large in-house application. In this application, all interfaces are specified using IDL, which is mapped to C++ using a custom mapping. We then present an architecture for a template-driven IDL compiler and describe the implementation of a prototype we built. With this compiler architecture, an IDL mapping can easily be specified and customized by writing an appropriate template.

1 Introduction

CORBA [1] is an enabling technology for building distributed systems, permitting the integration of distributed components at a higher level of communication than traditional byte-streams. This is achieved by providing the communication infrastructure for heterogeneous, distributed collections of objects, for which CORBA presents the communication abstraction of a method call on remote CORBA objects. The benefits derived are akin to the benefits of utilizing object oriented programming for building non-distributed programs.

In order to promote language and platform independence, CORBA encourages the use of an Interface Definition Language (IDL) specified by the Object Management Group (OMG). OMG IDL can only be utilized to specify the *interface* of a CORBA object, enforcing the separation of interface specification from object implementation. This also ensures that a client of a CORBA object remains unconcerned with the implementation of the object. Moreover, the client is also unconcerned with the implementation language of the CORBA object, simplifying the integration of distributed components that are implemented in different languages.

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 396–414, 2000.

Current mappings of IDL to implementation languages like C, C++ or Java use data-types that are CORBA or ORB-vendor specific. Moreover, in the IDL-C++ or IDL-Java mappings, the inheritance relations between the generated stub/skeleton classes and implementation classes are usually fixed by the IDL compiler. These factors impose additional constraints on the implementation of applications that utilize IDL. The problem is especially acute in legacy applications, which are already associated with a large, well-established code-base.

In order to enable applications to benefit from the concise expressiveness of IDL without committing to being completely CORBA compliant, the mapping of IDL to a particular implementation language should be decoupled from the IDL parser and code-generation engine. This makes it possible to customize the bridge between the application and the underlying ORB, introducing ample flexibility for building both, the application and the ORB. To aid this process, we propose a template-driven IDL compiler architecture. This compiler architecture not only permits the customization of an IDL mapping and generated code, but also enables all aspects of the underlying ORB to be configured. This approach can be considered to introduce the flexibility of tuning middleware to existing code-bases rather than the more common other way around.

The rest of this paper is organized as follows: Section 2 describes the benefits of customizing IDL mappings. Section 3 motivates our approach by presenting the custom IDL to C++ mapping we utilize in HEIDI, an existing in-house application. Section 4 presents the architecture and implementation of our proposed template-driven IDL compiler. Section 5 compares our approach with other approaches to customize ORBs, and Section 6 concludes this paper.

2 The Benefits of Customizing a Mapping

Among the important goals of the CORBA specification are *portability* and *interoperability* of CORBA compliant application code with different ORB implementations. Towards this end, CORBA defines mappings from OMG IDL to various programming languages including Java, C, C++, Smalltalk, COBOL and Modula 3. The mapping process is automated in an IDL compiler, which generates the framework for implementing a CORBA object from the IDL specification of its interface. In addition, the IDL compiler also generates client-side *stubs* and server-side *skeletons*, which collaborate with the underlying ORB to implement such activities as object registration, method call parameter marshaling and unmarshaling, and call dispatching. Usually, stubs and skeletons utilize an abstract interface to the underlying ORB functionality so that the same generated code can be utilized with ORBs that implement different *on-the-wire* protocols. The portability of a CORBA compliant object implementation across IDL compilers from different vendors is ensured in theory by each compiler conforming to the specified IDL to implementation language data-type mapping (Table 1) and object model. The interoperability between applications that utilize ORBs from different vendors is guaranteed by each ORB conforming to such a standard ORB protocol as the Internet Inter-ORB Protocol (IIOP).

Table 1. IDL to C++ Type Mappings

IDL Type	Prescribed C++ Type	Alternate C++ Mapping
long	CORBA::Long	long
boolean	CORBA::Boolean	XBool
float	CORBA::Float	float

The CORBA specification also provides guidelines for generating stubs and skeletons from an IDL interface. A typical inheritance hierarchy for C++ stubs and skeletons is shown in Fig. 1, where the non-shaded classes are generated by the IDL compiler and the shaded classes implemented by the application programmer. In this hierarchy, the implementation of a CORBA object can inherit from the generated skeleton, or remain unrelated to the generated classes, utilizing a *tie* class as a bridge to/from the ORB. Most IDL compilers generate stubs and skeletons conforming to a variation of this inheritance hierarchy.

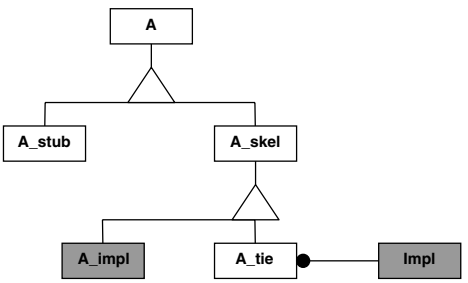

Fig. 1. Inheritance Hierarchy for C++ Stubs and Skeletons

Although portability and large-scale interoperability are important concerns of building software for distributed systems, there are several situations when it is useful to customize the code generated by an IDL compiler:

– **Using legacy code for CORBA-object implementations:** While being completely CORBA-compliant ensures portability across different ORBs and interoperability with other CORBA-compliant applications, several classes of enterprise applications may only require interoperability with other CORBA applications. This is especially true with legacy applications, which are often constrained by such factors as a large existing code-base or a widely used communication protocol. With most current IDL compilers, these applications can benefit from the concise expressiveness of IDL only by becoming completely CORBA compliant. However, a legacy application may utilize the C++ usages shown in Table 2, while the CORBA specification for mapping an IDL interface *A* to a C++ interface class *A* states that it is non-compliant

to declare either an instance, pointer or reference to A. As this shows, it can be an expensive, time-consuming process to integrate a legacy application into a CORBA-based distributed system.

Table 2. CORBA-prescribed and Legacy C++ Usages

CORBA-prescribed	Legacy
A_var a;	A a;
A_ptr p;	A* p;
void f(A_ptr& r);	void f(A& r);

- **Customizing the ORB:** An ORB that fully implements the CORBA specification is usually very big in terms of code size. For many classes of applications, this can be the major reason to decide against utilizing CORBA. This issue has motivated a recent OMG effort towards identifying an irreducible set of capabilities and characteristics for a minimal ORB. Such an ORB would implement a CORBA subset that is useful and acceptable for the applications in consideration. Keeping with this trend, we believe that it is also useful to permit the customization of the ORB to implement exactly the subset of CORBA functionality that is necessary for a particular class of applications.
- **Customizing the ORB Protocol and Messaging Formats:** Utilizing a standard inter-ORB protocol guarantees that an application can easily interoperate with other applications. However, such protocols are often expensive to use because they are designed for generality. Moreover, for many applications, a simple protocol or messaging format may suffice.
  To address this issue, most IDL compilers generate stubs and skeletons that utilize an abstract interface to the ORB. This keeps the IDL compiler, and hence the generated code independent of any particular ORB protocol, permitting the utilization of alternate protocols. With such an approach, utilizing a particular protocol involves choosing the appropriate ORB run-time library.
- **Incorporating Custom Optimizations:** Often, the code generated by an IDL compiler is not well suited for optimization. For instance, many IDL compilers use string comparisons to implement the dispatching logic in the skeleton. Such a scheme can be very expensive for interfaces with a large number of methods with long names. Alternate schemes that utilize nested comparisons [2], or a hash-table can result in faster dispatching.
  Marshaling/Unmarshaling code is typically associated with format conversions and copying. As pointed out in the Universal Stub Compiler (USC) work by O'Malley, et al [3], a user-level specification of the byte-level representations of data types can be effectively utilized to optimize copying operations, and therefore marshaling and unmarshaling code. It is clearly

beneficial to introduce such optimizations in generated stubs and skeletons in order to improve the performance of a remote call.

3 HeidiRMI

In order to demonstrate that it is indeed useful to customize an IDL mapping, we consider the motivation, design and implementation of HeidiRMI, the control-messaging infrastructure for Heidi. Heidi is a large in-house project currently being used to build and test prototype multimedia software systems [4]. In early versions of Heidi, all control messaging between distributed software components utilized a simple text-based request-response protocol over dedicated TCP/IP connections. This approach sufficed for the simple initial prototype applications we built. However, as more complicated prototypes were developed, it clearly became necessary to automate the process of generating control messaging support. OMG IDL was an available alternative, and was well suited to describe the control messaging interfaces in Heidi.

Using IDL along with a general-purpose ORB in Heidi was associated with problems that arose from the large amount of legacy code that was not CORBA-compliant, and the non-blocking nature of communication in a Heidi application. The large existing code-base clearly needed wide-spread changes before it could be integrated with a general-purpose ORB. Even if this were done, it would still be difficult to utilize a general purpose ORB because of the non-preemptive computation model of Heidi.

To avoid the wide-spread changes necessary to make existing Heidi code CORBA-compliant, we modified the OmniBroker[1] IDL compiler [5] to generate an alternate C++ mapping that conforms to existing Heidi code. The HeidiRMI mapping only utilizes Heidi defined data types, which simplifies the use of legacy Heidi code. Besides utilizing only existing Heidi data-types, the mapping also implements a delegation based relation between the skeleton and implementation classes as shown in Fig. 2. This approach ensures that no restructuring of the existing Heidi class hierarchy is necessary.

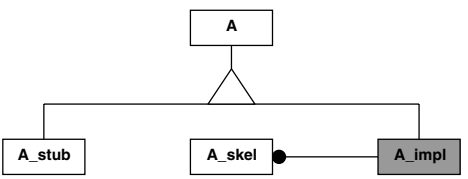

Fig. 2. IDL to C++ Mapping in HeidiRMI

The delegation model of the HeidiRMI mapping is similar to the *tie* approach in the CORBA-prescribed IDL to C++ mapping. A *tie* is usually im-

[1] OmniBroker is now called ORBacus.

plemented as a template to which the implementation class of the target object must be specified. This approach makes it unnecessary for the implementation class to depend on any of the classes generated by the IDL compiler. Although this simplifies the utilization of legacy code, there still is the dependency on CORBA-specific data types because method signatures in the implementation class must exactly match method signatures in the interface class. We therefore believe that *ties* alone are largely insufficient to address the problem of utilizing legacy code. Our approach of using a custom mapping on the other hand, provides the desired flexibility while maintaining a simpler relation between the implementation class and the skeleton. Moreover, such coding conventions as class naming can be easily customized, saving large amounts of otherwise mundane, but time consuming changes.

3.1 Implementation Details

We extended the IDL syntax in support of default parameters and passing parameters by value. Since legacy HEIDI code extensively utilized the ability to specify default parameters to a method in C++, we added support for the specification of default parameters in IDL. Default parameters are indicated as shown in the IDL interface presented in Fig. 3, and have the same effect as that of default parameters in C++ class specifications. Each default parameter to a method in an IDL interface is mapped to an appropriate default parameter in the generated C++ interface class.

In order to support passing parameters by value, we introduced the new *incopy* keyword, which is used as a qualifier for a method parameter. For simple data types, the effect of *incopy* is identical to that of *in*. However, object references passed *incopy* are copied across the IDL interface, if possible. The ORB run-time utilizes marshaling/unmarshaling primitives that the object implementation may have provided. Whether a particular object has actually implemented the required marshaling/unmarshaling primitives is determined by testing if it implements the *HdSerializable* interface. The dynamic type checking support that is implemented in HEIDI is utilized for this purpose. The semantics of passing parameters by value in HEIDIRMI are identical to the effect of passing a *Serializable* object that is not *Remote* as a parameter to a remote method in Java RMI [6].

Also shown in Fig. 3 are the relevant portions of the abstract C++ interface class generated by our customized HEIDIRMI IDL compiler. It can be seen that no CORBA-specific types are utilized: primitive IDL data-types are mapped to primitive C++ types, while **sequence** and **boolean** are mapped to the HEIDI specific *HdList* and *XBool* data types. Also, default parameters are mapped to appropriate C++ constants. Note that the IDL interfaces HEIDI*::A* and HEIDI*::S* are respectively mapped to the C++ interface classes *HdA* and *HdS*. This unconventional mapping facilitates the integration with legacy code, assuming that *HdA* and *HdS* were existing HEIDI interface classes.

Not shown in Fig. 3 is the generated support for dynamic type checking, which all HEIDI classes provide. Methods for marshaling and unmarshaling ob-

```
/* File A.idl */
module Heidi {
 // External declaration of Heidi::S
 interface S;
 // Heidi::Status
 enum Status {Start, Stop};
 // Heidi::SSequence
 typedef sequence<S> SSequence;
 // Heidi::A
 interface A : S
 {
   void f(in A a);
   void g(incopy S s);
   void p(in long l = 0);
   void q(in Status s = Heidi::Start);
   readonly attribute Status button;
   void s(in boolean b = TRUE);
   void t(in SSequence s);
 };
};
```

```
/* File A.hh */
// IDL:Heidi/Status:1.0
enum HdStatus { Start, Stop };
// IDL:Heidi/SSequence:1.0
typedef HdList<HdS> HdSSequence;
typedef HdListIterator<HdS>
                    HdSSequenceIter;
// IDL:Heidi/A:1.0
class HdA : virtual public HdS
{
public:
   virtual void f(HdA*) = 0;
   virtual void g(HdS*) = 0;
   virtual void p(long l = 0) = 0;
   virtual void q(HdStatus s = Start) = 0;
   virtual void s(XBool b = XTrue) = 0;
   virtual void t(HdSSequence*) = 0;
   virtual HdStatus GetButton() = 0;
   virtual ~HdA() {}
};
```

Fig. 3. Example IDL Interface and Generated C++ Interface Class

jects that implement the generated interface have also been omitted. These methods implement the logic for determining if a given object also implements *HdSerializable*, and passing control to the implementation object specific methods for marshaling/unmarshaling object state. This simplifies generated code for stubs and skeletons by putting together what would otherwise be redundantly generated marshaling/unmarshaling code.

In HeidiRMI, each object is associated with a stringified object reference. An object reference is composed of three parts: the bootstrap URL, the object identifier, and the object type. The bootstrap URL consists of a protocol-hostname-port tuple that provides a means to open a communication channel to the object. The object identifier uniquely identifies the object in a particular address space, while the type information ensures that the correct stub and skeleton is utilized in accessing the object. A typical stringified object reference is *@tcp:galaxy.nec.com:1234#9876#IDL:*Heidi*/A:1.0*. Although a HeidiRMI object reference may be considered minimal, it is not unlike an object reference in CORBA or any other remote object system.

The interaction diagram on the client-side of a remote method invocation is shown in Fig. 4. When a stub method is invoked, a new *Call* object that provides the generic functionality for making a remote method call is created. The stringified object reference of the target remote object forms the header of the *Call*. After any parameters to the remote method are marshaled into the *Call* object, the *Call* is *invoked*, resulting in the call request being sent to the server-side.

An *ObjectCommunicator* provides the abstraction of a communication channel on which individual requests can be demarcated. The current implementation of *Call* and *ObjectCommunicator* utilize a newline terminated string of ASCII characters to implement the on-the-wire protocol. The *Call* object provides the functions for marshaling and unmarshaling all primitive data types, as well as additional *begin* and *end* functions that permit structuring of the call request so that such composite data types as *structs* or *sequences* can be easily represented.

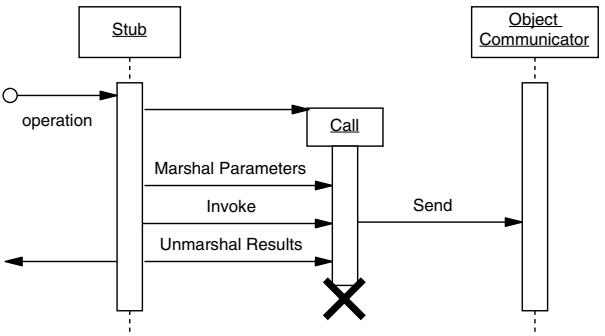

Fig. 4. Remote Method Invocation in HeidiRMI

The interaction at the server-side is shown in Fig. 5. The bootstrap port in each address space serves as means to initiate a communication channel. When a client connects to the bootstrap port (1), a new *ObjectCommunicator* is wrapped around the resulting connection. Connections are cached and reused in HeidiRMI, and only if there is no available connection is a new connection opened. The *ObjectCommunicator* reads in an incoming request (2) and encapsulates it in a *Call* object. The *Call* header contains the stringified object reference, whose type information and object identifier permit the selection of the appropriate *Skeleton*. Control is passed to the *dispatch* method of the selected *Skeleton*, where the remote method call parameters are unmarshaled. The *skeleton* then calls the desired method of the target object implementation, marshals any return value into the *Call* object, and sends the result back to the client-side.

An important aspect of HeidiRMI is that an implementation object is unconcerned with being remote accessible. The skeleton for a particular object is only created when a reference to it is being passed as either the parameter to, or the result of a remote call. Moreover, if the implementation object is *Serializable* and is being passed-by-value, then no *skeleton* is ever created. In this case, the marshaling method defined by the object is utilized to copy the object. At the receiving end, the type information contained in the object reference is utilized to create a stub of the appropriate type. Both stubs and skeletons are cached in each address-space in order to minimize the overhead of their creation.

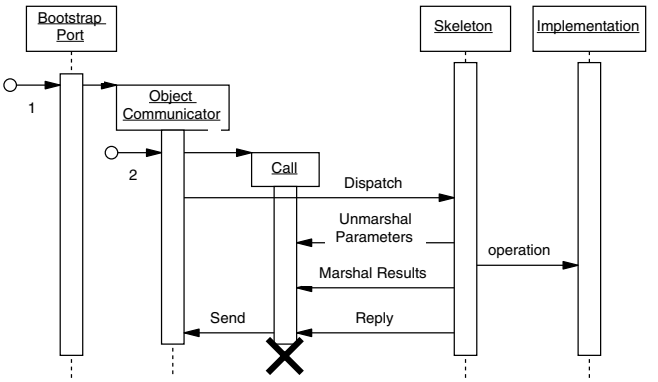

Fig. 5. Server-side Method Call Dispatching in HeidiRMI

The implementation of stubs and skeletons for an IDL interface is straightforward. All stubs inherit from a base *HdStub* class which provides the generic stub functionality. A stub also implements the C++ mapping of the IDL interface, and reflects the IDL inheritance structure appropriately. For the running example, the stub *A_stub* for the IDL interface *A* inherits functionality from the stub *S_stub* for the IDL interface *S*, and in addition implements the methods of interface *A*.

In HeidiRMI, skeletons do not share any inheritance relation with the abstract interface class. However, similar to generated stubs, skeletons also reflect the IDL inheritance structure. For the running example, the skeleton *A_skel* for interface *A* inherits from the skeleton *S_skel* for interface *S*. The *dispatch* method of *A_skel* first attempts to dispatch an incoming request to methods defined in the interface *A*. If this fails, then dispatching is delegated to the dispatch method of *S_skel*, continuing recursively up the skeleton class hierarchy. If *A* inherits from more than one interface, then dispatching is delegated to each of the corresponding skeleton super-classes in order.

3.2 Shortcomings of This Approach

Early use of our compiler involved reverse-engineering existing C++ interfaces into suitable IDL interfaces. However, the ease with which our approach permitted us to quickly build Heidi components led us to begin specifying their interfaces in IDL. HeidiRMI has thus become an integral part of the Heidi development environment, and our custom compiler has evolved into a key tool to build the system. Extensive utilization of HeidiRMI has strengthened our belief that IDL is indeed a powerful tool for specifying the interfaces of modules in a large application. By customizing the IDL compiler for HeidiRMI, we have succeeded in separating the utilization of IDL from the necessity of using a complete CORBA-compatible ORB.

However, the evolution of the HeidiRMI IDL compiler has also raised concerns regarding the limitations of the customization approach. It is evident that even a minor change in the IDL to C++ mapping requires compiler source code changes and recompilation. This concern led us to consider the alternative of template-based code-generation. Here, details of the IDL to implementation mapping are specified in a template, which the IDL compiler utilizes to drive its code generation. This greatly simplifies customization of the IDL compiler to generate code conforming to a desired mapping. Moreover, the very same compiler can be utilized with alternate templates to generate code in different implementation languages.

It should be noted, though, that our extensions for default parameters and passing parameters by value required IDL syntax changes. Such syntax enhancements must be reflected in the IDL parser, and is outside the scope of any template scheme for code-generation.

4 Architecture of a Customizable IDL Compiler

As with OmniBroker, most current IDL compilers hard-code the IDL mapping. Although this approach serves well to ensure CORBA-conformance, the inflexibility restricts the ability to customize generated code. In order to overcome this restriction, we propose the compiler architecture shown in Fig. 6. In this architecture, an IDL compiler consists of a generic parser that creates an *enhanced syntax tree* (EST) representation of the IDL source, and a template driven code-generator that utilizes the EST to generate stub/skeleton code. Figure 6 also shows the languages utilized in our prototype implementation.

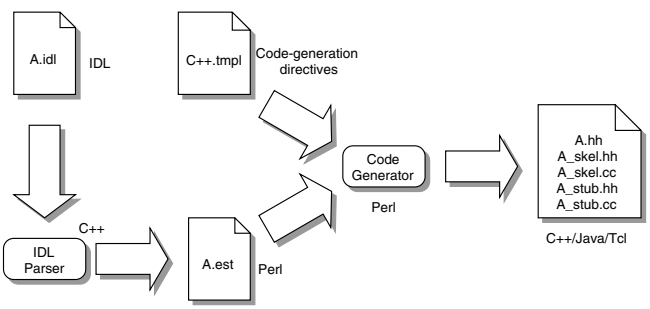

Fig. 6. Template-driven IDL Compiler Architecture

The key point to note in this compiler architecture is that the generated code no longer depends on anything that is hard-coded in the compiler modules. While both modules clearly must understand the EST representation, the parser must additionally understand the IDL syntax, while the code-generator must understand the syntax for specifying a template. The generated code now

depends only on the template that is provided to the code-generator. This makes it possible to tune generated code by only changing the template specification. Moreover, this approach also makes it is possible to generate code for an IDL mapping to any implementation language.

4.1 Prototype Implementation

In order to determine the feasibility of the template approach, we built a hybrid two-stage IDL compiler using the OmniBroker compiler to parse IDL, and a template-driven back-end code-generator that is based on Jeeves [7].

We modified the Omnibroker compiler to generate a perl program that encodes the EST representation of the IDL source. An EST representation is a parse tree that is organized so that similar elements are grouped together. For instance, IDL permits interspersing of attributes and methods in an interface. This can be seen in the example of Fig. 3 where the attribute *button* occurs between the methods *q* and *s*. The children of a node corresponding to an interface in a regular IDL parse tree would therefore be ordered exactly as the corresponding order of attributes and methods in the IDL. On the other hand, an EST would be constructed so that nodes corresponding to all the attributes are grouped, as are those corresponding to all the methods. This can be seen in Fig. 7, where the EST for the IDL interface presented in Fig. 3 maintains the node corresponding to the *button* attribute in a separate sub-tree of the node corresponding to the interface *A*. Irrelevant parts of the EST have been omitted from Fig. 7 for simplicity. A portion of the actual perl program that encodes the EST is shown in Fig. 8.

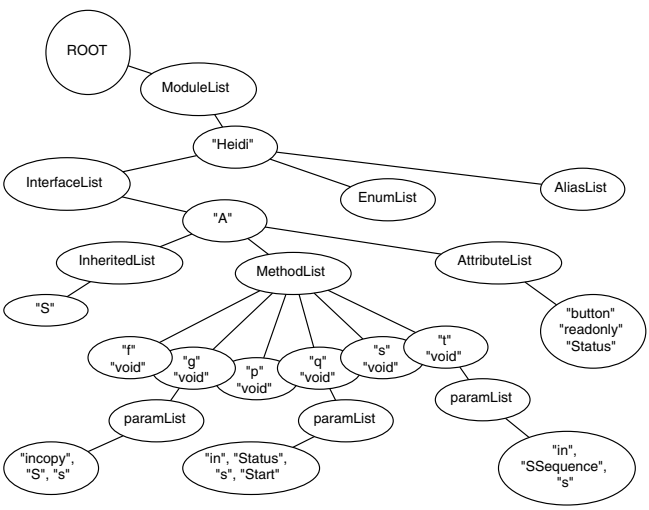

Fig. 7. Extended Syntax Tree for A.idl

```
#!/usr/bin/perl
use Ast;
use JeevesUtil;
$ROOT = $n0 = Ast::New("Root");
#
# IDL:Heidi:1.0
#
$n1 = Ast::New("Heidi", "Module", $n0);
    #
    # IDL:Heidi/Status:1.0
    #
    $n2 = Ast::New("Status", "Enum", $n1);
    @m = [ Start, Stop ];
    $n2→AddProp("members", @m);
    #
    # IDL:Heidi/SSequence:1.0
    #
    $n2 = Ast::New("SSequence", "Alias", $n1);
    $n2→AddProp("type", "sequence");
    #-————————————
        $n3 = Ast::New("", "Sequence", $n2);
        $n3→AddProp("type", "objref");
        $n3→AddProp("typeName", "Heidi_S");
        $n3→AddProp("IsVariable", true);
    #
    # IDL:Heidi/A:1.0
    #
    $n2 = Ast::New("A", "Interface", $n1);
    $n2→AddProp("Parent", "Heidi_S");
        #
        # IDL:Heidi/A/f:1.0
        #
        $n3 = Ast::New("f", "Operation", $n2);
        $n3→AddProp("type", "void");
        #-————————————————
            $n4 = Ast::New("a", "Param", $n3);
            $n4→AddProp("type", "objref");
            $n4→AddProp("typeName", "Heidi_A");
            $n4→AddProp("getType", "in");
    . . .
```

Fig. 8. Portion of the generated Perl program representing the EST

Grouping similar nodes in the EST simplifies the specification of a template that drives the code-generator back-end. This can be seen in Fig. 9, which presents a template for the C++ interface class header as defined in the HeidiRMI mapping. The template syntax is straightforward: the '@' character serves as an escape for code-generation commands, while the other lines are just printed out with appropriate substitutions. The '$' indicates the name of an attribute of the node under current consideration, and is substituted by its text value before being printed out. The use of a *map* makes it possible to convert an IDL name into one that is suitable in the context of the code that is being generated, changing Heidi*::A* to *HdA*, for instance. The *foreach* command walks through a list of nodes, examining each node in sequence. Since the EST has already classified the nodes into separate sub-trees according to their types, using the *foreach* command will in fact exhaustively enumerate all elements of the lists of methods, attributes, or parameters.

In our current implementation, code-generation is a two-step process. In the first step, a perl program that represents the actual code generator is automatically produced from the given template. A modified version of Jeeves [7] is utilized for this process. This program is then executed together with the perl program generated in the IDL parse stage to produce the desired IDL mapping. The latter program essentially rebuilds the EST within the perl interpreter, while the former uses the EST to generate the desired code based on the template.

Although the two-step code-generation stage is akin to recompiling the compiler, it is possible to merge the two code-generation steps as we plan to do in the future. It can also be noted that the first step of the code-generation stage need only be performed once for a particular code-generation template. Moreover, evaluating a perl program that directly rebuilds the EST, as we do in the second code-generation step, is certainly more efficient than parsing an external representation of the EST.

4.2 Experience

Our template approach to generating code introduces the flexibility of quickly building an ORB to suit an existing application. For instance, it took us about two weeks and 700 lines of **tcl** code to build an IIOP compatible **tcl** ORB. This exercise enabled the integration of an existing **tcl** management GUI application with a CORBA-based distributed system. We utilized our template-driven IDL compiler to generate an IDL-**tcl** mapping that suited the existing **tcl** code (Fig. 10). Our experience goes to show that the template approach has introduced the option of quickly developing an ORB to suit an existing application, as opposed to only having the option of making the existing application CORBA-compliant.

We have also utilized our hybrid compiler to generate an experimental HeidiRMI compatible IDL-Java mapping. The goal of this work was to enable the use of HeidiRMI to configure a generic Heidi engine from within a Java program. The class inheritance structure in our IDL-Java mapping was similar to the HeidiRMI C++ mapping, but expanded multiple super-classes in order to get

```
@foreach interfaceList -map interfaceName CPP::MapClassName
@openfile ${interfaceName}.hh
/* File ${interfaceName}.hh */
class ${interfaceName} :
@foreach inheritedList -ifMore ',' -map inheritedName CPP::MapClassName
        virtual public ${inheritedName} ${ifMore}
@end inheritedList
{
@foreach attributeList -map attributeType CPP::MapType
        ${attributeType} ${attributeName};
@end attributeList
public:
@foreach methodList -map returnType CPP::MapReturnType
  virtual ${returnType} ${methodName}(
@foreach paramList -ifMore ',' -map paramType CPP::MapType
@if ${defaultParam} == ""
                    ${paramType} ${ifMore}
@else
                    ${paramType} ${paramName} = ${defaultParam} ${ifMore}
@fi
@end parameterList
                                ) = 0;
@end methodList
  virtual ~${interfaceName}() {}

  // Attribute access methods
@foreach attributeList -map attributeType CPP::MapType
        ${attributeType} Get${attributeName}() const = 0;
@if ${attributeQualifier} ≠ "readonly"
        void Set${attributeName}(${attributeType}) = 0;
@fi
@end attributeList

};
@end interfaceList
```

Fig. 9. Template for Generation of C++ Interface Class Header

```
if {[info vars "IDL:Receiver:1.0"] ≠ ""} return
set IDL:Receiver:1.0 1

BOA::addIdlMapping ::Receiver "IDL:Receiver:1.0"

class ReceiverStub {
  inherit Stub

  constructor {ior connector} {
    Stub::constructor $ior $connector
  } {}

  public method print {text} {
    set c [$pb_connector_ getRequestCall $this "print" 0]
    $c insertString $text
    $c send
    # void return
    $c release
  }
}

class ReceiverSkel {
  inherit Skel

  constructor {implObj} {
    Skel::constructor $implObj
  } {}

  public method print {c} {
    set text [$c extractString]
    $pb_obj_ print $text
    # void return
  }
}
```

Fig. 10. Sample **tcl** stub and skeleton code

around the unavailability of multiple inheritance in Java. The IDL-Java mapping we implemented also does not support default parameters as the corresponding C++ mapping does.

The template approach also makes it easy to customize primitive ORB functionality and protocols. Assuming that all generated code utilizes generic ORB functionality provided by an ORB library, it is possible to write templates for stubs and skeletons that only use portions of the ORB library to minimize the ORB footprint as may be required for small embedded devices. HeidiRMI itself utilizes an entirely text-based wire-protocol that suffices for the control messaging needed in Heidi. Utilizing such a text-based protocol permitted a *"human"* client to telnet into the bootstrap port of a Heidi application and type in simple HeidiRMI requests to debug the system. This was made possible by writing templates that utilized a custom *Call* object that implemented the appropriate marshaling/unmarshaling functionality.

5 Related Work

Although ORB customization has received the attention of many researchers, current work has mostly concentrated on finding an appropriate balance between ORB functionality, code-size and efficiency, while preserving a fixed, conforming programming interface. By addressing the problem of customizing the ORB interface, our approach can be considered to add an additional degree of flexibility to ORB design. We first compare our approach with other approaches to ORB customization, and then with other approaches to building configurable IDL compilers.

One approach to customizing an ORB is to synthesize it from primitive components. For instance, *Quarterware* [8] provides the core components required for middleware implementations: data marshaling/unmarshaling, object references, transport, dispatching, invocation policy, and wire protocol. Specific middleware like CORBA or Java RMI are implemented by suitably selecting and customizing these *Quarterware* components. Similarly, *Jonathan* [9] provides interface references, binding types, and binding factories using which a CORBA or RMI *personality* can be implemented. While this approach can clearly be utilized to customize ORB functionality to fit application requirements, it does not simplify customizing the language interface presented by the ORB to the application. Our template driven code-generation can therefore be considered to complement the synthesis approach. Moreover, the availability of primitive components will certainly simplify designing suitable templates for a particular class of applications.

A less flexible approach to synthesizing an ORB is for it to expose certain object patterns and interfaces. With this approach, certain aspects of a core ORB engine can be customized by attaching a custom module. For instance a *strategy* may be attached in TAO [10] and *dynamicTAO* [11,12], a *subcontract* in Spring [13], or a *policy* in the extensions to RMI suggested in [14]. The CORBA standard [1] provides the Object Adaptor (OA) through which server objects interact with the ORB. An OA can make such services as a database appear as

an object. ORB implementations too provide features based on this approach: Orbix [15] provides *filters* that are triggered in the dispatch path, and *smart proxies* that can cache object state. Visibroker [16] provides similar features called *interceptors* and *smart stubs*. Java RMI [6] permits the customization of its reference layer so that alternate invocation semantics can be implemented. While this approach certainly permits the customization of ORB functionality, the degree of flexibility introduced is clearly limited to only those aspects of the ORB that are actually exposed.

Our two-stage compiler architecture is not unlike that of the Omnibroker compiler itself. The Omnibroker parser stores an abstract representation of the IDL source in a possibly persistent global *Interface Repository* (IR) in support of a distributed development environment. The code-generation stage then queries the IR for details of each required IDL interface, generating code as it walks the IDL parse tree. We believe our own code generator would integrate well with the OmniBroker framework to directly utilize the OmniBroker IR. The EST that our template code-generation requires could either be generated on the fly from the parse tree in the IR, or the IR could modified to store the EST instead of the parse tree.

An extensive effort towards modularizing IDL compilers has been made in the *Flick* project at the University of Utah [2]. Although the *Flick* compiler framework has been designed with the goal of supporting multiple IDLs, implementation languages and protocols, the flexibility that has been introduced does not simplify the tuning of generated code. Each new IDL mapping would typically require the design and development of a new *Flick* back-end module, which in turn would require recompilation for every change in the mapping. In contrast, our approach of specifying the IDL mapping in a template clearly simplifies the customization of a mapping. However, our approach of building an IDL compiler is consistent with that of *Flick* and we believe that it is possible to incorporate the template approach into the *Flick* framework by writing a suitable template-driven back-end.

We believe *Flick* is superior at providing certain sophisticated optimizations, especially those involving marshal buffer management and parameter management. However, code-generation optimizations involving inlining code or nested message demultiplexing can easily be accomplished with the template approach. A good strategy may be to utilize the template approach when code-generation flexibility is desired, but resort to writing a custom *Flick* back-end for incorporating sophisticated optimizations.

Although the ILU project at XEROX, Palo Alto Research Center [17] also emphasizes customizability, the customizability is restricted to primitive ORB functionality rather than IDL mappings. For instance new messaging protocols, URL parsing functions, or authentication and accounting schemes can be specified to the ILU kernel. Many different target languages including C, C++, Java, and Modula-3 are supported, but the code-generation for each of these is based on fixed mappings of ILU's native IDL to the target language. This limitation

makes it hard to utilize ILU to generate code that is compliant with legacy application code.

6 Conclusions

Most ORBs are designed to provide features that satisfy a large class of applications. However, not all available features are necessary for all applications. Moreover, a particular set of features may not suffice for certain classes of applications. This makes it necessary for an ORB to be customizable and tunable to the requirements of a particular class of applications.

In this paper, we first illustrated that there do indeed exist several classes of applications where it is useful to customize the code that is generated to bridge application code with the underlying ORB. We then presented a flexible template-driven code-generator where the mapping of IDL to the implementation language is specified in a template. This approach simplifies tuning the IDL mapping, and can be used to complement other approaches of ORB customization. Extensive utilization of this approach suggests that it is a powerful technique for tailoring an ORB to application requirements. Moreover, the template approach can also be utilized to quickly generate the framework for object implementations, which are often associated with fixed code *patterns*.

We have already strengthened our belief in the template approach by building support for an IDL-Java mapping for HeidiRMI (without support for default parameters), and a new IDL-**tcl** mapping that utilizes a custom **tcl** ORB. In the future, we plan to further consolidate our position by considering the design of IDL mappings for minimal, real-time ORBs based on IIOP.

References

1. Object Management Group, Inc., *The Common Object Request Broker: Architecture and Specification*, Aug. 1996. Document PTC/96-08-04, Revision 2.0. 396, 411
2. E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A flexible, optimizing compiler," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Las Vegas, Nevada, USA), June 1997. 399, 412
3. S. O'Malley, T. Proebsting, and A. Montz, "USC: A universal stub compiler," in *Proceedings of the Conference on Communication Architectures, Protocols and Applications (SIGCOMM)*, (London, UK), Aug. 1994. 399
4. M. Ott, G. Michelitsch, D. Reininger, and G. Welling, "An architecture for adaptive QoS and its application to multimedia systems design," *Computer Communications*, vol. 21, pp. 334–349, Feb. 1998. 400
5. Object Oriented Concepts, Inc., *OmniBroker*. http://www.ooc.com/ob/. 400
6. Sun Microsystems, Inc., *Java Remote Method Invocation Specification*. http://java.sun.com/products/jdk/rmi/index.html. 401, 412
7. S. Srinivasan, "Template-driven code generation," in *Advanced Perl Programming*, ch. 17, O'Reilly Associates, Inc., Aug. 1997. 406, 408

8. A. Singhai, A. Sane, and R. H. Campbell, "Quarterware for middleware," in *Proceedings of the International Conference on Distributed Computing Systems*, (Amsterdam, The Netherlands), May 1998. 411
9. B. Dumant, F. Horn, F. D. Tran, and J.-B. Stefani, "Jonathan: An open distributed processing environment in Java," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, (The Lake District, England), Sept. 1998. 411
10. D. C. Schmidt and C. Cleeland, "Applying patterns to develop extensible and maintainable ORB middleware," *Communications of the ACM*, vol. 40, no. 12, 1997. 411
11. F. Kon and R. H. Campbell, "Supporting automatic configuration of component-based distributed system," in *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, (San Diego, California, USA), May 1999. 411
12. M. Roman, F. Kon, and R. H. Campbell, "Design and implementation of runtime reflection in communication middleware: the *dynamicTAO* case," in *Proceedings of the ICDCS '99 Workshop on Middleware*, (Austin, Texas, USA), May 1999. 411
13. G. Hamilton, M. L. Powell, and J. G. Michell, "Subcontract: A flexible base for distributed programming," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, (Asheville, North Carolina, USA), Dec. 1993. 411
14. G. Welling and M. Ott, "Structuring remote object systems for mobile hosts with intermittent connectivity," in *Proceedings of the 18th International Conference on Distributed Computing Systems*, (Amsterdam, The Netherlands), May 1998. 411
15. IONA Technologies, *The ORBIX Architecture*. http://www.iona.com/products/orbix/. 412
16. Visigenic, Inc., *The New Application Architecture, Version 3.0*, 1997. 412
17. XEROX Corporation, *ILU Reference Manual*. http://pubweb.parc.xerox.com/hypertext/ilu/index.html. 412