Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks

Alexander B. Arulanthu
alex@cs.wustl.edu
Department of Computer Science
St. Louis, MO 63130

Carlos O'Ryan and Douglas C. Schmidt
{coryan,schmidt}@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Michael Kircher
Michael.Kircher@mchp.siemens.de
Siemens ZT
Munich, Germany

Anniruda Gokhale
gokhale@research.bell-labs.com
Lucent Technologies
Murray Hill, NJ

This article will appear in the March 2000 C++ Report magazine.

1 Introduction

To make informed choices among middleware alternatives, developers of distributed object systems should understand the patterns and components used to implement key features in CORBA ORBs. Recent Object Interconnection columns [1, 2, 3] have explored the features of the CORBA Messaging specification [4]. In this article, we describe key C++ features, patterns, and components used to implement an OMG IDL compiler that supports the Asynchronous Method Invocation (AMI) callback model defined in the CORBA Messaging specification.

The CORBA Messaging specification defines two AMI programming models, the *polling* model and the *callback* model. In both models, only clients behave asynchronously, *i.e.*, server applications do not change at all. These AMI models are outlined briefly below:

Polling model: In this model, each two-way AMI operation returns a `Poller valuetype` [5], which is very much like a C++ or Java object in that it has both data members and methods. Operations on a `Poller` are just local C++ method calls and not remote CORBA operation invocations. The polling model is illustrated in Figure 1. The client can use the `Poller` methods to check the status of the request so it can obtain a server's reply. If the server hasn't replied yet, the client can either (1) block awaiting its arrival or (2) return to the calling thread immediately and check back on the `Poller` to obtain the `valuetypes` when it's convenient.

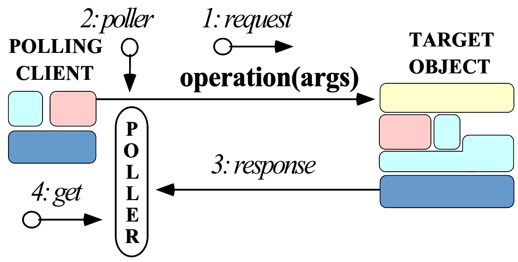

Figure 1: Polling Model for CORBA Asynchronous Twoway Operations

Callback model: In this model, when a client invokes a two-way asynchronous operation on an object, it passes an object reference to a *reply handler servant* as a parameter, as shown in Figure 2. This object reference is not passed to the server, but instead is stored locally by the client ORB. When

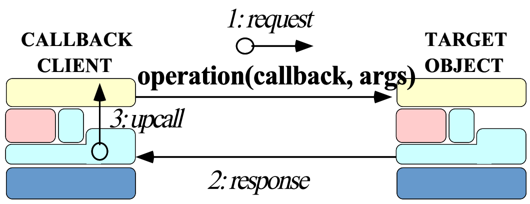

Figure 2: Callback Model for CORBA Asynchronous Twoway Operations

the server replies, the client ORB receives the response and uses the reply handler servant provided by the client application to dispatch the response to the appropriate callback operation. This model requires client application developers to obtain and initialize a POA and to activate objects in the POA, which effectively makes the application behave as both a client and a server.

Reply handler servants are accessed via normal object references. Therefore, servants can be implemented in processes other than the client or the server involved in the original invocation. For instance, it's possible for a reply handler servant to process "third-party" replies. The most common use-case, however, is for the original client to process the response.

In general, the callback model is more efficient than the polling model because the client need not invoke method calls on a valuetype repeatedly to poll for results. Moreover, the AMI callback model provides the following benefits compared to alternative CORBA invocation models:

Simplified asynchronous programming model: AMI allows operations to be invoked asynchronously using the *static invocation interface* (SII). Using SII for AMI eliminates much of the tedium and complexity inherent in the *dynamic invocation interface* (DII)'s deferred synchronous model. In particular, DII requires programmers to insert parameters explicitly into Request objects, whereas the SII-generated stubs automate and optimize [6] parameter.

Improved quality of service: When implemented properly, AMI can improve the scalability of CORBA applications. For instance, it minimizes the number of client threads that would otherwise required to perform two-way synchronous method invocations (SMI). In addition, AMI is important for real-time CORBA applications [7] because it helps to bound the amount of time spent in ORB operations, *i.e.*, only the client processing time has to be considered when sending a request. This decoupling of client processing time from server operation execution time helps to simplify real-time scheduling analysis [8].

The remainder of this article is organized as follows: Section 2 presents an example that illustrates the CORBA AMI callback programming model in more detail; Section 3 describes the C++ features, patterns, tools, and components used in TAO's IDL compiler to implement the CORBA AMI callback model; and Section 4 presents concluding remarks.

2 Programming the CORBA AMI Callback Model

In this section, we review how the AMI callback model works from the perspective of a CORBA/C++ application developer. The steps required to program CORBA AMI callbacks are similar to the development of any CORBA application, *i.e.*, OMG IDL interface(s) must be defined and client code must be written to use the generated stubs. Servers require no changes to work with AMI, however, because they are unaware of whether a client invokes operations synchronously or asynchronously [1]. Thus, the changes required to support AMI applications affect only how clients are written, as described step-by-step below.

Step 1: Define the IDL interface and generate the stubs: Throughout this article, we'll use the following Quoter IDL interface to illustrate how to use and implement the AMI Callback model:

```
module Stock
{
  interface Quoter {
    // Two-way operation to retrieve current
    // stock value.
    long get_quote (in string stock_name);
  };

  // ...
};
```

After IDL interfaces are defined, they must be passed through an OMG IDL compiler, which generates a standard set of C++ stubs and skeletons. For each two-way operation in the IDL interface, an IDL compiler can generate the SMI and AMI stubs that applications use to invoke operations. As discussed in [9], servers remain unchanged. Thus, the skeletons generated by the IDL compiler are no different for AMI than for SMI, so we won't describe them in this article. The stubs for asynchronous operations are different, however. In particular, they are given the name of the corresponding synchronous operation, but with a sendc_ prefix prepended.

For example, an IDL compiler that supports AMI would generate the following pair of stubs for our Quoter interface:

```
  // Usual SMI stub.
  CORBA::Long Stock::Quoter::get_quote
    (const char *stock_name)
  { /* IDL compiler-generated SMI stub code... */ }

  // New AMI stub (described below).
  void Stock::Quoter::sendc_get_quote
    // ReplyHandler object reference
    (Stock::AMI_QuoterHandler_ptr,
     const char *stock_name)
  { /* IDL compiler-generated AMI stub code... */ }
```

In addition to having a slightly different name, the asynchronous sendc_get_quote operation has a different signature than the synchronous get_quote operation. In particular, sendc_get_quote has no return value and is passed an object reference to an application-defined subclass of the following AMI_QuoterHandler:

```
class AMI_QuoterHandler :
public Messaging::ReplyHandler
{
  // Callback stub invoked by the client ORB
  // to dispatch the reply.
  virtual void get_quote (CORBA::Long ami_return_val)
  { /* IDL compiler-generated stub code... */ }
};
```

The AMI_QuoterHandler is generated automatically by an AMI-enabled IDL compiler; it determines where the reply from the server will be dispatched. Note that the send_get_quote method doesn't need a return value because the value of the stock will be passed back to the get_quote callback operation defined by the AMI_QuoterHandler shown above. For more information on the AMI callback mapping rules for OMG IDL to C++, please see [2].

2. Implement the reply handler servant: Next, a client programmer must implement the reply handler servant by subclassing from AMI_QuoterHandler, as shown below:

```
class My_Async_Stock_Handler
  : public POA_Stock::AMI_QuoterHandler
{
public:
  // Callback method prints stock value.
  virtual void get_quote
    (CORBA::Long ami_return_val) {
    cout << ami_return_val << endl;
  }
};
```

Although this implementation is "correct" it isn't very useful since there is no way to distinguish callbacks resulting from AMI calls to different stocks! The following are common strategies for addressing this problem:

• **Servant-per-AMI-call strategy:** Here's a reply handler servant implementation that keeps track of which stock name it's associated with and prints out this stock name and stock value returned the server in the get_quote callback:

```
class My_Async_Stock_Handler
  : public POA_Stock::AMI_QuoterHandler
{
public:
  My_Async_Stock_Handler (const char *stockname)
    : stockname_ (CORBA::string_dup (stockname))
  {}

  // Callback servant method.
  virtual void get_quote
    (CORBA::Long ami_return_val) {

      cout << stockname_ << " = "
           << ami_return_val << endl;
  }

private:
  CORBA::String_var stockname_;
};
```

Since the My_Async_Stock_Handler servant stores the stockname_ that it's requesting it can easily distinguish callbacks resulting from multiple AMI calls by simply instantiating a different servant for each AMI call. The drawback, of course, is if there are many simultaneous asynchronous calls the memory footprint of the client will increase.

• **Activation-per-AMI-call strategy:** One way to distinguish separate AMI calls without requiring a separate object per-invocation is to explicitly activate the same servant multiple times in the client's POA. As described in [10], each activation can be given a designated object id. The get_quote callback method can then examine this object id to determine to which invocation the reply belongs to, as follows:

```
using namespace PortableServer;

class My_Async_Stock_Handler
  : public POA_Stock::AMI_QuoterHandler
{
public:
  // Save the Current pointer
  My_Async_Stock_Handler (Current_ptr current)
    : current_ (Current::_duplicate (current))
  {}

  // Callback servant method.
  virtual void get_quote
    (CORBA::Long ami_return_val) {
    // Get the object id used for current upcall.
    ObjectId_var oid =
      current_->get_object_id ();
    // Convert the ObjectId to a string.
    CORBA::String_var stock_name =
      ObjectId_to_string (oid.in ());
    cout << stock_name.in () << " = "
         << ami_return_val << endl;
  }
private:
  // Store the POA Current to get fast access
  // to the ObjectId.  Note that the Current
  // can be assigned in one thread and
  // used by another thread.
  PortableServer::Current_var current_;
};
```

Before making AMI calls, we create a POA whose policies allow the client to explicitly activate the same servant multiple times. Then, for each AMI call we create a special object id that stores the stock name, as follows:

```
  // A POA with the USER_ID and MULTIPLE_ID
  // policies:
  POA_var poa = ....;

  // Obtain the POA Current object
  CORBA::Object_var tmp =
    this->resolve_initial_references ("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_duplicate (tmp.in ());

  // Initialize the servant
  My_Async_Stock_Handler servant (current.in ());

  // Make asynchronous two-way calls using
  // the AMI callback model.
  for (int i = 0; i < MAX_STOCKS; i++) {
    // Convert the stock name into an ObjectId.
    ObjectId_var oid =
      string_to_ObjectId (stocks[i]);

    // Activate the Object with that ObjetId
    poa->activate_object_with_id (oid.in (),
                                  &servant);
    // Get the object reference
```

```
    CORBA::Object_var tmp =
      poa->id_to_reference (oid.in ());
    Stock::AMI_QuoterHandler_var handler =
      Stock::AMI_QuoterHandler::_narrow (tmp.in ());

    // Send the request.
    quoter_ref->sendc_get_quote (handler.in (),
                                 stocks[i]);
  }
```

Although this approach is more complex to program, it is more scalable than the servant-per-AMI-call strategy because it uses a single servant for all asynchronous calls. However, both strategies require an entry-per-AMI-call in the client POA's active object map. One way to reduce this overhead, therefore, is to use Servant Locators [11] that activate the client's reply handler *on-demand*, thereby minimizing memory utilization.

- **Server-differentiated-reply strategy:** An alternative strategy for differentiating multiple AMI calls requires a minor modification to the `Quoter` IDL interface. For instance, an `out` parameter can be added to the `get_quote` operation, as follows:

```
interface Quoter {
  // Two-way operation to retrieve current
  // stock value.
  long get_quote (in string stock_name,
                  out string stock_name);
};
```

In this strategy, the server will return the stock name as a parameter to the `get_quote` callback, as follows:

```
void
My_Async_Stock_Handler::get_quote
    (CORBA::Long ami_return_val,
     const char *stock_name) {
  cout << stock_name << " = "
       << ami_return_val << endl;
}
```

Thus, just one servant need be used to distinguish all the AMI callbacks and it only needs to be activated once in the client's POA.

In general, however, the use of an `out` parameter is obtrusive and incurs more network overhead in order to pass the stock name back to the client, compared with allocating a different servant for each AMI call. One way to reduce this overhead is to use the Asynchronous Completion Token (ACT) [12] pattern by adding small, fixed-size `inout` parameter to the `get_quote` operation, as follows:

```
interface Quoter {
  typedef short ACT;

  // Two-way operation to retrieve current
  // stock value.
  long get_quote (in string stock_name,
                  inout ACT act);
};
```

The `ACT` would be initialized by the client to indicate a particular AMI call and then passed to the server. The server would subsequently return the `ACT` unchanged as a parameter to the reply handler servant. This handler could then map the `ACT` to the associated actions and state necessary to complete the reply processing. If the size of the `ACT` was smaller than the size of the stock name this strategy can reduce network bandwidth a bit.

Step 3: Programming the client application: After the IDL compiler generates the synchronous and asynchronous stubs, programmers can develop a client that works much the same as any other CORBA application. For example, the client must obtain an object reference to a target object on a server and invoke an operation. Unlike a conventional two-way SMI call, however, when a client invokes a two-way AMI operation, it passes an object reference to a reply handler servant as a parameter. This object reference is not sent to the server, however. Instead, the client ORB stores it locally and uses it to dispatch the appropriate callback operation after the server replies to the client.

The following code, excerpted from [2], illustrates how a C++ programmer would program the AMI callback model using the servant-per-AMI call strategy described earlier. First, we define a `get_stock_quote` function that makes AMI calls:

```
// Issue asynchronous requests.
void get_stock_quote (void)
{
  // Set the max number of ORB stocks.
  static const int MAX_STOCKS = 3;

  // NASDAQ abbreviations for ORB vendors.
  static const char *stocks[MAX_STOCKS] =
  {
    "IONAY"  // IONA Orbix
    "INPR"   // Inprise VisiBroker
    "IBM"    // IBM Component Broker
    "BEASYS" // BEA Web Logic Enterprise
  };

  // Reply handler servants.
  My_Async_Stock_Handler *handlers[MAX_STOCKS];

  // Reply handler object references.
  Stock::AMI_QuoterHandler_var
  handler_refs[MAX_STOCKS];

  for (int i = 0; i < MAX_STOCKS; i++) {
    // Initialize the servants.
    handlers[i] =
      new My_Asynch_Stock_Handler (stocks[i]);

  // Initialize object references (note that
  // _this() interacts with the client-side POA).
  handler_refs[i] = handlers[i]->_this ();
}

  // Make asynchronous two-way calls using
  // the AMI callback model.
  for (int i = 0; i < MAX_STOCKS; i++)
    quoter_ref->sendc_get_quote (handler_refs[i],
```

```
                                  stocks[i]);

  // ...
  // Clean up dynamically allocated resources.
}
```

After making asynchronous invocations, a client typically performs other tasks, such as checking for GUI events or invoking additional asynchronous operations. When the client is ready to receive replies from server(s), it enters the ORB's event loop[1] using the standard `work_pending` and `perform_work` methods defined in the CORBA `ORB` interface, as follows:

```
// Event loop to receive all replies as callbacks.
while (/* ... */)
  if (orb->work_pending ())
    orb->perform_work ();
  else
    /* ... potentially do something else ... */
```

When a server responds, the client ORB receives the response and then dispatches it to the appropriate method on the reply handler servant so the client can handle the reply. In other words, the ORB turns the response into a request on the corresponding reply handler object reference passed to the ORB during the client's original invocation. Figure 3 illustrates how our client application uses the AMI Callback model.

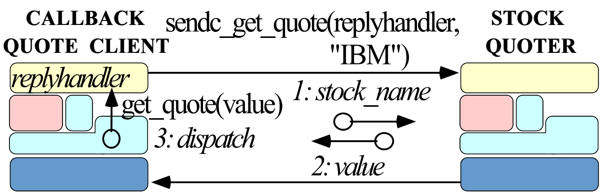

Figure 3: AMI Callback Quoter Use-case

In the example above, the client implements the reply handler servant locally. Thus, after a reply arrives from the server, the client ORB invokes the `get_quote` stub on the `AMI_QuoterHandler` callback object. This stub marshals the arguments and invokes the virtual `get_quote` method on the `My_Async_Stock_Handler` reply handler servant.

Note that a reply handler also can be identified by an object reference to a remote object. In this case, its servant will receive "third-party" replies resulting from requests invoked by other clients.

[1] In addition, if asynchronous replies arrive while a client is blocked waiting for a synchronous reply, the asynchronous reply can be dispatched in the context of the waiting client thread.

3 IDL Compiler Support for CORBA AMI Callbacks

Section 2 outlined how to program the AMI callback model from a CORBA application developer's perspective. This section explains the C++ features, patterns, and components used by TAO's IDL compiler to generate the stubs necessary to support the AMI callback model. TAO is an open-source[2] CORBA-compliant ORB designed to address the quality of service (QoS) requirements of high-performance and real-time applications [8]. Below, we present a general overview of TAO's IDL compiler and explain how it generates C++ code that implements AMI callbacks.

3.1 Overview of TAO's IDL Compiler

An IDL compiler is a critical component of an ORB. It is responsible for mapping IDL input files into *stub* and *skeleton* classes, which serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [13] and provide a strongly-typed, static invocation interface that marshals application parameters into a common data-level representation. Conversely, skeletons implement the *Adapter* pattern [13] and demarshal the data-level representation back into typed parameters that are meaningful to an application. In addition, generated skeletons are responsible for demultiplexing operation names carried in client requests to their associated methods on servants.

The process of translating OMG IDL into the standard C++ mapping is complex. Moreover, IDL compilers must be flexible, *e.g.*, in order to generate compiled and/or interpretive (de)marshaling code that meets the needs of various types of applications [6]. In addition, the ORB and the IDL compiler must collaborate to provide optimal performance for parameter marshaling and demarshaling. For example, optimizations in the ORB may trigger changes to its IDL compiler so that generated code exploits the new features.

TAO's IDL compiler parses IDL files containing CORBA interfaces and data types, and generates stubs and skeletons, which are then integrated with application code, as shown in Figure 4. Figure 5 illustrates the interaction between the internal components in TAO's IDL Compiler. The front-end of TAO's IDL compiler parses OMG IDL input files and generates an abstract syntax tree (AST). The back-end of the compiler "visits" the AST to generate CORBA-compliant [14] C++ source code. We describe the front-end and back-end of TAO's IDL compiler in more detail below.

[2] The source code and documentation for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

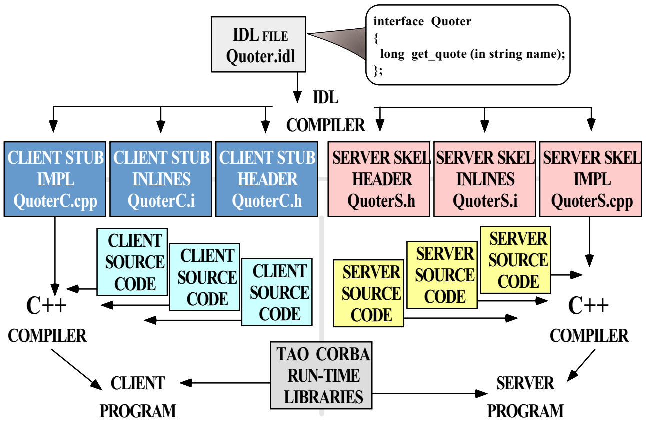

Figure 4: C++ Files Created by TAO's IDL Compiler

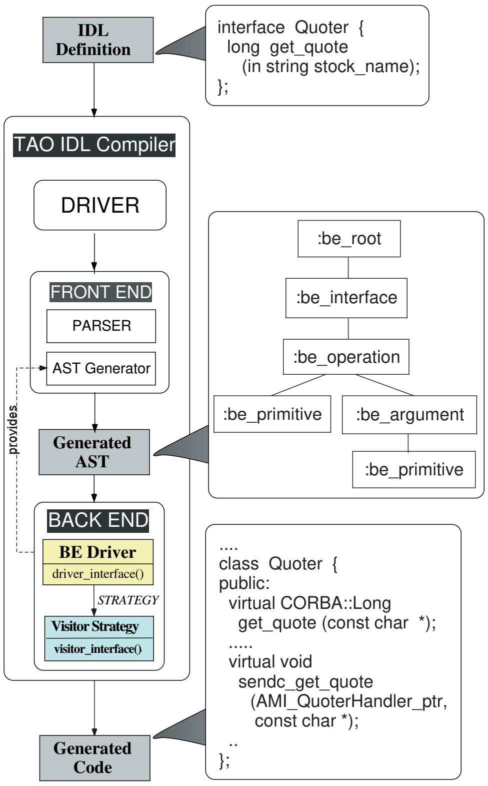

Figure 5: Interactions Between Internal Components in TAO's IDL Compiler

3.1.1 TAO's IDL Compiler Front-end Design

TAO's IDL compiler front-end is a heavily modified extension of the freely available SunSoft IDL compiler front-end, with many new CORBA features, portability enhancements, and bug fixes. The following components are contained in TAO's IDL compiler front-end:

OMG IDL parser: The parser is generated from a `yacc` [15] specification of the OMG IDL grammar. The action for each grammar rule invokes methods on AST node classes to build the AST. The AST is stored in main memory and shared between the front-end and the back-end.

Abstract syntax tree generator: Different nodes of the AST correspond to different OMG IDL features. The front-end defines a base class called `AST_Decl` that maintains information common to all AST node types. Specialized AST node classes, such as `AST_Interface` or `AST_Union`, inherit from this base class, as shown in Figure 6. In ad-

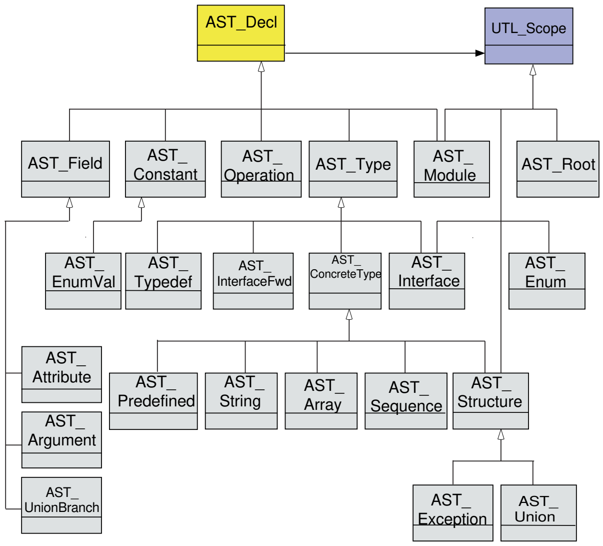

Figure 6: TAO's IDL Compiler AST Class Hierarchy

dition, the front-end defines the `UTL_Scope` class, which maintains scoping information, such as the nesting level and the list of components that form a fully scoped names, *e.g.*, `Stock::Quoter`. `UTL_Scope` is a base class for all AST nodes representing OMG IDL features that can define scopes, such as `structs` and `interfaces`.

Driver program: The driver program invokes the helper programs and directs the parsing and AST generation process. First, it parses command-line arguments and invokes the C++ preprocessor to import `#includes` for each input file. Next, it invokes the IDL parser on the C++ preprocessor output to generate the AST in-memory. Finally, the driver passes the AST to the back-end code generator.

3.1.2 TAO's IDL Compiler Back-end Code Generator Design

The original SunSoft IDL compiler front-end only parses OMG IDL files and generates the corresponding abstract syntax tree (AST). To create a complete OMG IDL compiler for TAO, we developed a back-end for the OMG IDL-to-C++ mapping. TAO's IDL compiler has been designed to be scalable and configurable to support various optimization techniques [6]. For example, TAO's IDL compiler back-end can selectively generate C++ code that is optimized for (1) a GIOP protocol interpreter [16] or (2) compiled (de)marshaling in stubs and skeletons [6].

TAO's IDL compiler back-end employs design patterns, such as Abstract Factory, Strategy, and Visitor [13], that simplify its design and implementation and allow it to generate stubs/skeletons that use either compiled or interpretive (de)marshaling [6]. These patterns also make it easier to support new requirements, such as AMI stub generation as described in Section 3.2. In addition, TAO's IDL compiler back-end employs the `gperf` perfect hash function generator [17], which creates optimal operation demultiplexers automatically.

Below, we describe how these patterns and tools were applied to resolve the key design challenges faced when developing TAO's IDL compiler back-end.

Enhancing IDL compiler back-end maintainability

• **Context:** An IDL compiler should be maintainable. For example, it should be possible to add new features without requiring extensive compiler modifications. Likewise, it should be easy to debug the compiler, *e.g.*, if its generated code deviates from the CORBA language mapping specifications.

• **Problem:** The SunSoft IDL compiler front-end that forms the basis of TAO's IDL compiler uses a `yacc`-generated parser to build an AST representation from IDL input files. Developers can add back-ends that generate code from the AST by using the Strategy and Abstract Factory patterns [13], as described below. Although these two patterns simplify the creation of multiple back-ends and allow back-end developers to control the AST representation, they do not, by themselves, solve the following key design challenges:

- *Need to know the exact type of a node* – As the back-end traverses the AST to generate code, the exact type of the node being visited must be known. For example, the declaration of an input argument will change depending on the type of the argument. Basic IDL types, such as `short` and `long` are passed by value. Conversely, IDL `structs` are passed by reference. The original SunSoft IDL compiler used downcasts to determine the exact type of an AST node. This mechanism was tedious and error-prone, however, because it forced developers to (1) write `if/else` and `switch` statements to detect the type of node being processed and then (2) use a C++ downcast operator to obtain node-specific information needed for code generation.
- *Dependency of the mapping on the context* – The same types of AST nodes can have different mappings depending on their *context*, *i.e.*, their location in the AST and what portion of the code is being generated. For example, the mapping of an object reference as the type of an structure field is `T_var`, whereas the mapping as an input argument is `T_ptr`. Not all types follow the same rules. For example, both a field and an input argument of type `short` are mapped as `CORBA::Short`. Moreover, the same input argument is used to generate the stub method declaration, the skeleton declaration, and multiple times to generate the definitions of the stub and the skeleton, in each case with slightly different variations used. In general, the type of the AST node and the context where the node is used affects the C++ code emitted by TAO's IDL compiler.
- *Poor scalability of virtual methods* – A potential solution to the problem outlined in the previous paragraph would be to use virtual methods to represent each context. Each node type could then override the virtual methods to generate the appropriate code. Unfortunately, virtual methods do not scale effectively as the number of different contexts increases.

  For instance, the SunSoft IDL compiler uses the same node to represent an operation argument multiple times, *e.g.*, in the stub declaration, in the stub definition, before marshaling the request, during the request marshaling, and while demarshaling the reply. Likewise, this same node is used multiple times for similar purposes in the skeleton. The mapping also depends on whether the argument node represents an `in`, `inout`, `out`, or `return` parameter.

  Each time a different variation is required, the number of virtual methods can increase. Although clever tricks can be used to minimize the number of virtual methods, the result is still overly complex. Moreover, an IDL compiler can be a non-trivial software application, *e.g.*, TAO's IDL compiler contains over 120,000 lines of code. Thus, if the code generation logic for a particular context is spread across the compiler source it may be hard to maintain.

• **Solution:** The Visitor pattern [13] allows operations to be applied correctly to nodes in a heterogeneous object structure, such as an abstract syntax tree. This pattern is commonly used in languages that do not support "double dispatching," *i.e.*, the polymorphic operations cannot depend on message arguments, only on the object receiving the message. We used the Visitor pattern in the TAO IDL compiler's back-end to resolve the following problems outlined above:

- The type of an AST node is determined easily because (1) the visitor has a different callback operation for each type and (2) it receives the most derived type as an argument. Other solutions, such as Interpreter [13], would require the tedious downcasts described earlier.
- Different contexts are represented by different visitors. Thus, it is easy to add more contexts as needed. The callback method of each visitor can be used to treat each type differently depending on its context. For example, TAO's IDL compiler uses an operation argument more than 10 times when generating the code for stubs and skeletons. Given the sheer number of contexts where a single AST node can be used, therefore, it would be inflexible to use a single object to keep track of the current context.
- The code for each context can be found easily because it is isolated in a single visitor class. The code for a type in that context is also easily found by using the callback method. Checking a type across all contexts is slightly harder, but the names of the callback methods are unique enough that a simple tool like `grep` can locate them automatically.
- Changing the behavior of the IDL compiler only requires substituting the visitors involved. As we describe later, using the Abstract Factory pattern to create the visitors further simplifies these substitutions.

The Visitor pattern was appropriate because the actions performed on a particular AST node depend on both the context where the actions occur *and* the type of the particular node. As we will see below, this solution also allows us to modify the generated code by simply changing some of the visitors.

To implement the Visitor pattern in TAO's IDL compiler back-end, we added methods to our back-end AST nodes so they could be traversed by visitor objects. A single visitor represents a particular context in the code generation, *e.g.*, whether to generate argument declarations or to marshal the arguments to generate a request. The visitor can consult the type of the node and generate proper code depending on its context and type.

Each visitor usually delegates part of its work to other visitors. For example, the compiler generates the method declarations for a skeleton class using one visitor, which delegates to another visitor to generate the argument declarations. This decoupling between (1) the different contexts in the code generation and (2) the types being processed allows us to customize a particular task without affecting other portions of the IDL compiler. For example, TAO's IDL compiler supports both compiled and interpretive marshaling. By using the Visitor pattern, however, most of the differences between those marshaling techniques are concentrated in the generation of the stubs and skeletons, *i.e.*, the generation of the header files remains unmodified.

Enhancing IDL compiler back-end flexibility

- **Context:** An IDL compiler should be flexible, *e.g.*, capable of being adapted to generate code with different space/performance tradeoffs or even different language mappings, such as Java or C.

- **Problem:** End-users can select whether TAO's IDL compiler generates interpretive or compiled (de)marshaling code on a per-file or per-operation basis [6]. This selection affects the generation of the stubs and skeleton methods, part of the `AST_Operation` mapping, and requires the generation of CDR stream insertion and extraction operators, *i.e.*, it adds new phases to the code generation process. Most of the code to generate the stub and skeleton *declarations* remains unchanged, however. Although it is possible to generate a completely different syntax tree for each case, this approach could cause significant duplication of code because each change is relatively small.

- **Solution:** The Strategy pattern [13] provides an abstraction for selecting one of several candidate algorithms and packaging these algorithms within an OO interface. We used this pattern to allow different code generation algorithms to be configured as visitors. By using the Strategy pattern, for instance, the only visitors that must be replaced to switch from compiled to interpreted code generation are those responsible for generating stub and skeleton implementations. The generation of IDL structures and sequences, stub declarations, and skeleton declarations remains unchanged.

Ensuring semantic consistency of complex, related strategies

- **Context:** Users can select different (de)marshaling techniques via command-line options. As described above, the IDL compiler uses the Strategy pattern to select different Visitors that generate this code.

- **Problem:** Many strategies and visitors must be replaced when changing the style of code generated by the compiler. If these strategies and visitors are not changed in a semantically consistent manner, the generated code will not work correctly and possibly will not even be valid input for for a C++ compiler.

- **Solution:** The Abstract Factory pattern [13] provides a single component that creates related objects. We applied abstract factories in the TAO IDL compiler's back-end to localize the construction of the appropriate visitors, as shown in Figure 7. Controlling the creation of visitors using the Abstract

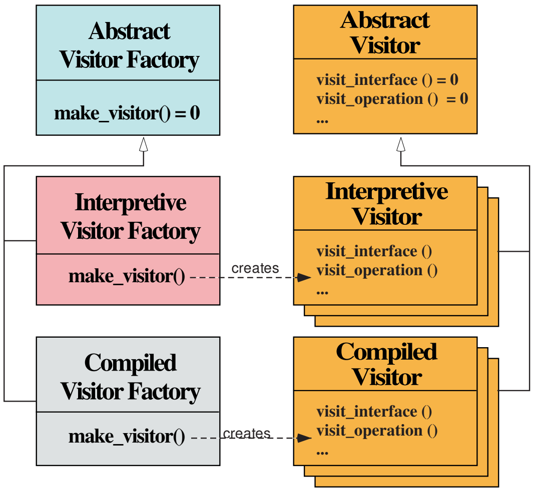

Figure 7: Creating Visitors in TAO's IDL Compiler Using Abstract Factories

Factory pattern allows CORBA application programmers to make a wholesale selection of alternative stubs and skeletons implementations. Moreover, this pattern makes it straightforward to disable certain features, such as the generation of insertion and extraction operators to and from `CORBA::Any` objects, the generation of implementation templates where the user inserts new code, and the generation of AMI stub code to reduce the footprint of applications that do not use AMI features.

Optimizing operation demultiplexing in skeletons

• **Context:** Once an ORB's Object Adapter identifies the correct servant for an incoming CORBA request [18], the next step is for the generated skeleton to demultiplex the request to the correct operation within the servant. Figure 8 illustrates operation demultiplexing.

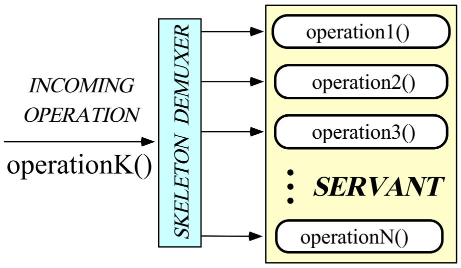

Figure 8: Operation Demultiplexing in Skeletons

• **Problem:** For ORBs like TAO that target real-time embedded systems, operation demultiplexing must be efficient, scalable, and predictable [18]. This requires the skeleton generated by TAO's IDL compiler to locate the C++ method that matches the operation name passed with the incoming client request in constant time.

Solution: To generate constant time operation demultiplexers, the TAO IDL compiler's back-end uses `gperf` [17], which is a freely available perfect hash function generator distributed with the TAO release. The `gperf` tool automatically generates a perfect hash function from a user-supplied list of keyword strings. The generated function can determine whether an arbitrary string is a member of these keywords in constant time. Figure 9 illustrates the interaction between the TAO IDL compiler and `gperf`. TAO's IDL compiler invokes

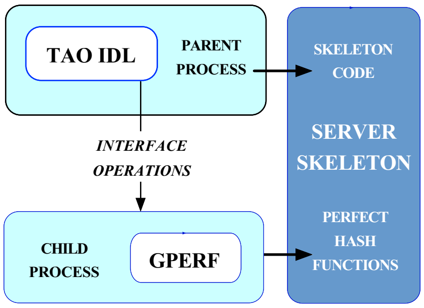

Figure 9: Integrating TAO's IDL Compiler and GPERF

`gperf` as a co-process to generate an optimized lookup strategy for operation names in IDL interfaces.

Figure 10 plots operation demultiplexing latency as a function of the number of operations. This figure indicates that

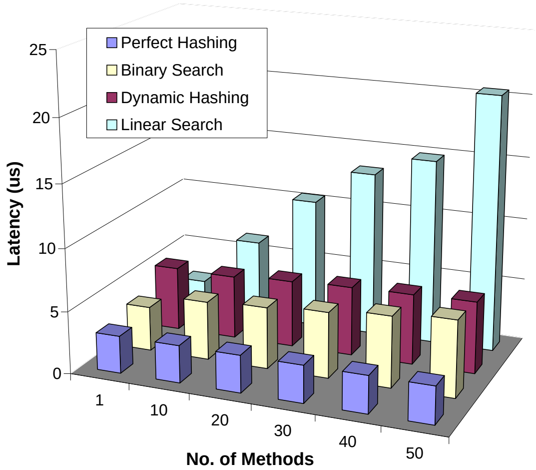

Figure 10: Operation Demultiplexing Latency with Alternative Search Techniques

the perfect hash functions generated by `gperf` behave predictably and efficiently, outperforming alternatives such as dynamic hashing, linear search, and binary search.

3.2 Overview of IDL Compiler C++ Code Generation

Now that we've outlined the components and patterns in TAO's IDL compiler, we'll explain how it generates C++ code for AMI callbacks.[3]

3.2.1 Generate Implied-IDL

An IDL compiler that supports the CORBA AMI callback model is responsible for mapping OMG interfaces to so-called "implied-IDL" interfaces [2]. Each implied-IDL interface consists of the `sendc_` operation for each two-way operation and the reply handler interface corresponding to each interface found in the original IDL file. For instance, the implied-IDL for our `Quoter` IDL example is shown below:

```
// Implied-IDL
module Stock
{
  interface Quoter {
    // Original two-way operation.
    long get_quote (in string stock_name);

    // Implied asynchronous operation.
    void sendc_get_quote
      (in AMI_QuoterHandler handler,
       in string stock_name);
   };

  // ...

  // Implied type-specific callback ReplyHandler.
  interface AMI_QuoterHandler :
    Messaging::ReplyHandler {
    // Callback for reply.
    void get_quote (in long ami_return_val);

    // ...
  };
};
```

Alternative strategies for generating implied-IDL: There are several strategies for modifying an existing SMI IDL compiler to generate the mapping code for implied-IDL AMI interfaces:

1. One-pass memory-based strategy: An IDL compiler's existing SMI code generation logic can be modified to produce AMI stubs at the same time as it produces the SMI stubs. This solution leverages existing IDL compiler features. However, it requires the modification of existing AST nodes to represent multiple IDL constructs. For example, the node that represents an `interface` would also represent the `ReplyHandler` for that `interface`. Likewise, the node representing an operation must also represent the `sendc_` method and the callback operation on the reply handler. Such a design would either require (1) multiple new visitors with special mapping rules for each node type or (2) more state to be maintained in each node to indicate how it should be used. In any case, the complexity of the IDL compiler implementation increases, which makes it harder to maintain.

2. Two-pass file-based strategy: One way to reduce some complexity of the one-pass strategy is to modify the IDL compiler to run in two passes. The first pass transforms the original IDL file into an implied-IDL temporary file. The second pass then reads this temporary file and generates C++ stubs and skeletons. Unfortunately this solution is not practical in many environments. For instance, in platforms that do not support namespaces the code for the AMI reply handlers must be inserted into the same scope where the stub classes are generated. Such an approach would require generating a complete implied-IDL file, and then taking the generated code and inserting some portions of it in different scopes of the generated code. This design is hard to implement and increases the time the IDL compiler requires to generate code.

3. Two-pass memory-based strategy: One way to avoid the slow processing time of a two-pass file-based IDL compiler is to make an additional pass over the AST *in-memory* before generating C++ code. During this second pass, additional nodes can be inserted into the existing AST to represent the implicit-IDL constructs that support AMI. The second pass can be implemented using new visitors that iterate over the tree and add the new entities.

Implementing the two-pass memory-based strategy in TAO: TAO's IDL compiler uses the third strategy outlined above to generate C++ code corresponding to implied-IDL. We selected a two-pass memory-based strategy because it (1) ran faster than the two-pass file-based strategy, (2) involved fewer intrusive changes to TAO's existing SMI IDL compiler design, and (3) provided a more scalable framework for the polling model (which requires implied-IDL constructs), as well as for future OMG IDL extensions, such as the CORBA Components Model [19].

To implement two-pass memory-based implied-IDL AST generation, we enhanced TAO's existing SMI IDL compiler to use several interface and operation strategies and a new AMI implied-IDL "preprocessing visitor." This preprocessor visitor is executed immediately after the AST created by the front-end of the IDL compiler is passed to the back-end. For every implied-IDL construct one of the following three techniques is then used by the preprocessing visitor to generate the implied-IDL code.

1. Insert: This technique inserts new nodes into the AST. Each node corresponds to a particular type of AMI implied-IDL. For instance, this technique is used for all

[3] To save space, we do not discuss AMI exception handling [4] in this article.

ReplyHandlers because they need normal stubs and skeletons.

2. Strategize: This technique applies strategies on existing nodes to trigger additional code generation, rather than inserting new nodes into the AST. For example, the sendc_ operation cannot be inserted in the AST because that would also generate a corresponding operation in the skeleton and the sendc_ operation must be visible only on the client. Using the Strategy pattern solves this problem cleanly without requiring major changes to the IDL compiler's design.

3. Insert and strategize: This technique is a combination of the two previous ones. Some nodes representing the implied IDL code are inserted into the tree. Other code is generated using strategies that modify the behavior of some visitors. For example, the reply handler operations are inserted, but also strategized to generate *reply-stubs*, which are described below.

3.2.2 Generate Stubs for Asynchronous Invocations

For each two-way operation defined in an IDL interface, an IDL compiler generates the corresponding sendc_ method used by client applications to invoke AMI operations. The first argument of a sendc_ operation is an object reference to the reply handler, followed by the in and inout arguments defined in the signature of the original two-way IDL operation. The return type for the sendc_ operation is void because the stub returns immediately without waiting synchronously for the server to reply.

In our Quoter application, for example, the IDL compiler generates the sendc_get_quote stub method in the client source file, as outlined below:

```
// Stub for asynchronous invocations.
void Stock::Quoter::sendc_get_quote
  // ReplyHandler object reference
  (Stock::AMI_QuoterHandler_ptr reply_handler,
  const char *stock_name)
{
  // Step 1. Marshal arguments.
  request_buffer << stock_name;

  // Step 2. Setup connection, store ReplyHandler
  // and stub to handle reply-stubs in the ORB.
  Asynch_Invocation invocation
    (reply_handler
     &Stock::AMI_QuoterHandler::get_quote_reply_stub,
     request_buffer);

  // Step 3. Send request to server and return.
  invocation.invoke ();

  // Note: No demarshaling necessary.
}
```

We will examine each of these steps in more detail in a subsequent article.

3.2.3 Generate Reply Handler Classes

For each interface in the IDL file, the IDL compiler generates an interface-specific reply handler that inherits from the standard Messaging::ReplyHandler base class. The client ORB can use this subclass to dispatch server replies to application-defined reply handler servants. For example, the client stub header file generated by TAO's IDL compiler for the Quoter interface contains the following reply handler skeleton, with the methods shown:

```
namespace Stock
{
  class AMI_QuoterHandler
    : public Messaging::ReplyHandler
  {
  public:
    // Reply handler reply-stub.
    static void get_quote_reply_stub
      (Input_CDR reply_buffer,
       AMI_QuoterHandler_ptr reply_handler);

    // Callback stub invoked by Client ORB
    // to dispatch the reply.
    virtual void get_quote
      (CORBA::Long ami_return_val);
  };
};
```

The get_quote_reply_stub and get_quote methods are stubs generated automatically by TAO's IDL compiler, as described below.

Reply-stubs: SMI dispatching is straightforward because demarshaling is performed by the stub that invoked the operation. For two-way SMI calls, this stub is always blocked in the activation record waiting for the server's reply. AMI dispatching is more complex, however, because the stub that invoked the operation goes out of scope after the request is sent and control returns to the client application. Thus, for two-way AMI calls, the stub does not block waiting for the server's reply.[4]

As shown in Figure 11, when an AMI reply arrives, the client ORB must demultiplex to the reply handler servant (*e.g.*, My_Async_Stock_Handler) defined by the client application developer, demarshal the arguments, and dispatch the appropriate callback method (*e.g.*, get_quote). To simplify the demultiplexing and dispatching of asynchronous replies, TAO's IDL compiler generates a concrete static method for each two-way AMI operation. These methods, which we call *reply-stub*s, perform the following steps: (1) declare the parameters corresponding to signature of the operation, (2) demarshal the reply, (3) invoke the callback method on the reply handler provided by the client, and (4) clean up any dynamically allocated memory used to process the reply. In contrast,

[4]This also makes it hard for the ORB to react on LOCATION_FORWARD GIOP messages. The current version of the CORBA Messaging specification does not mention how to solve this issue, though future versions hopefully will.

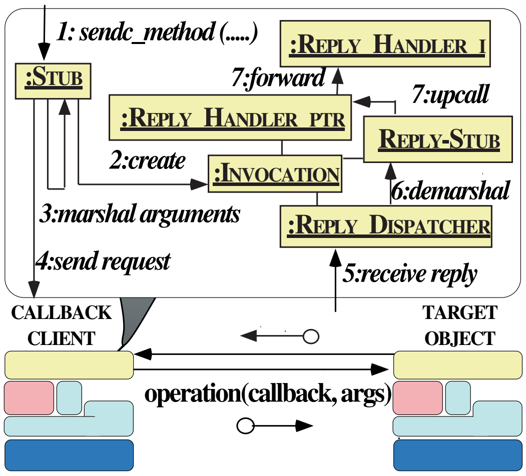

Figure 11: Client ORB Interactions for an Asynchronous Twoway Operation

SMI stubs are much simpler; they simply demarshal the reply into the parameters provided by the caller and return control to the client application.

When sending a request, the `sendc_` stub for an AMI call passes the client ORB a pointer to the reply-stub method and a pointer to the reply handler servant. When the reply arrives from the server, the client ORB passes the reply buffer and the reply handler servant to this reply-stub. For instance, when TAO's IDL compiler parses the `get_quote` operation of the `Quoter` interface, it generates the follow `get_quote_reply_stub` method in the client stub source file:

```
// Reply handler reply-stub.
void
Stock::AMI_QuoterHandler::get_quote_reply_stub
  (Input_CDR reply_buffer,
   AMI_QuoterHandler_ptr reply_handler)
{
  // Step 1. Result arguments.
  CORBA::Long ami_return_val;

  // Step 2. Demarshal results from <reply_buffer>
  // using CDR extraction operators.
  reply_buffer >> ami_return_val;

  // Step 3. Call reply handler callback method via
  // its reply-stub.
  reply_handler->get_quote (ami_return_val);

  // Step 4. Perform any needed cleanup activities.
}
```

This reply-stub performs the four steps outlined earlier.

Stubs for reply handler servant callback operations: These stubs are invoked by the reply-stubs on behalf of the client ORB. They make *synchronous* invocations on the reply handler servant to dispatch the reply to the appropriate callback operation (*e.g.*, `get_quote`). The first argument in the callback operation is the result of the asynchronous operation, followed by all the `out` and `inout` arguments defined by the two-way operation in the original IDL interface.

The reply handler servants follow the same rules required to implement any CORBA objects. For instance, users must activate their reply handler servants within a POA and the ORB must invoke operations transparently on object references to remote or local reply handlers. Thus, an IDL compiler must generate all the code for implied-IDL that is required for any other IDL interface.

For the `Quoter` interface, the TAO IDL compiler generates the `get_quote` callback method shown in the first code fragment in Section 3.2.3.

3.2.4 Generate Reply Handler Servant Skeletons

An OMG IDL compiler that supports CORBA's AMI callback model must also generate skeletons for reply handler interfaces. These reply handler skeletons contain methods whose signatures define the result arguments, *i.e.*, the return value, followed by the `out` and `inout` arguments of the original two-way operation.

As with regular IDL, each two-way operation in an implied-IDL interface generates a static reply handler servant skeleton method. This method performs the following steps: (1) allocates memory for the arguments, (2) demarshals the request into those arguments, and (3) dispatches the operation through the POA. In general, skeletons for reply handlers are simpler than skeletons for general IDL interfaces because they have no return values or output arguments. Moreover, they only have `in` arguments, which are derived from the return value and any `inout` and `out` arguments defined in the original operation.

TAO applies the same collocation optimizations [20] for AMI reply handlers as it applies for conventional SMI stubs and skeletons. These optimizations are particularly important for AMI because reply handlers are most commonly collocated with the client ORB. In this case, no extra marshaling and/or demarshaling steps are needed to process the reply. To support remote reply handlers, however, an ORB must be able to generate requests while processing a reply. Thus, it must be reentrant and allow new requests to be dispatched by the ORB Core.

For the `Quoter` interface, `ReplyHandler` servant code genreated by TAO's IDL compiler in the client-side header file is defined as follows:

```
namespace POA_Stock
{
  class AMI_QuoterHandler
    : public POA_Messaging::ReplyHandler
  {
  public:
    // Pure virtual callback method (must be
    // overridden by client developer).
    virtual void get_quote
```

```
      (CORBA::Long ami_return_val) = 0;

    // Servant skeleton.
    static void get_quote_skel
      (Input_CDR cdr, void *reply_handler);
  };
}
```

The implementation of the generated `get_quote_skel` servant skeleton extracts the AMI return value and `out`/`inout` parameters from the `Input_CDR` buffer and dispatches the upcall on the appropriate servant callback method. For example, the following code is generated by TAO's IDL compiler for the `Quoter` interface:

```
void
POA_Stock::AMI_QuoterHandler::get_quote_skel
(Input_CDR cdr,
void *reply_handler)
{
  // Step 1: Demarshal the AMI ``return value.''
  CORBA::Long ami_return_val;
  cdr >> ami_return_val;

  // Step 2: Downcast to the reply handler servant.
  POA_Stock::AMI_QuoterHandler *skeleton =
  static_cast <POA_Stock::AMI_QuoterHandler *>
    (reply_handler);

  // Step 3: Dispatch callback method on this servant.
  skeleton->get_quote (ami_return_val);
}
```

This skeleton performs the three steps outlined earlier.

4 Concluding Remarks

The Asynchronous Method Invocation (AMI) callback model is an important feature that has been integrated into CORBA via the OMG Messaging specification [4]. A key aspect of AMI callbacks is that operations can be invoked asynchronously using the *static invocation interface* (SII). This feature avoids much of the complexity inherent in the *dynamic invocation interface* (DII)'s deferred synchronous model.

In this article, we explain how an IDL compiler can be structured to support the CORBA AMI callback model. We also show how C++ languages features, such as inheritance and dynamic binding, can be guided by familiar design patterns, such as Visitor, Abstract Factor, and Strategy, to provide a very flexible and robust IDL compiler architecture that can adapt to changes not foreseen in the original design. These patterns allowed us to concentrate on implementing the new CORBA AMI callback features, rather than wrestling with internal IDL compiler design issues.

In a subsequent article, we will discuss the various components an ORB should support in its run-time architecture to implement the AMI callback functionality. We'll also show performance results that demonstrate the benefits of using AMI versus the SMI and DII deferred synchronous models.

References

[1] D. C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, November/December 1998.

[2] D. C. Schmidt and S. Vinoski, "Programming Asynchronous Method Invocations with CORBA Messaging," *C++ Report*, vol. 11, February 1999.

[3] D. C. Schmidt and S. Vinoski, "Time-Independent Invocation and Interoperable Routing," *C++ Report*, vol. 11, April 1999.

[4] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[5] Object Management Group, *Objects-by-Value*, OMG Document orbos/98-01-18 ed., January 1998.

[6] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[7] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[8] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[9] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 9, November/December 1997.

[10] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 10, April 1998.

[11] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 10, Sept. 1998.

[12] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[15] S. Johnson, *YACC - Yet another Compiler Compiler*. Bell Laboratories, Murray Hill, N.J., Unix Programmers Manual ed.

[16] A. Gokhale and D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, January 1998.

[17] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," *C++ Report*, vol. 10, November/December 1998.

[18] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[19] BEA Systems, *et al.*, *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

[20] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, November/December 1999.