Patterns and Performance of Dynamic CORBA Middleware

Jeff Parsons
parsons@cs.wustl.edu
Department of Computer Science
Washington University in St.Louis
St. Louis. MO

Abstract

As distributed systems increase in complexity, scope, and ubiquity, the contexts in which they are applied become more open-ended and dynamic. For example, an important and growing class of distributed applications, such as interface browsers, network managers, distributed debugging and visualization tools, and scripting languages, require flexible middleware support where statically typed knowledge of all possible operation names and signatures at compile-time is overly restrictive. Supporting these types of applications effectively requires some form of dynamic typing that enables the discovery of operation names and parameter types at run-time.

This paper provides two contributions to the study of dynamically typed middleware. First, it outlines the key design challenges faced when adding dynamic typing capability to CORBA middleware. Second, it describes how these design challenges were resolved via the systematic application of patterns and object-oriented design techniques. This work was done in the context of the ADAPTIVE Communication Environment (ACE), which is an object-oriented toolkit for developing networked application software, and The ACE ORB (TAO), which is an open-source and widely adopted OMG CORBA object request broker (ORB) that is implemented using frameworks in ACE.

1 Introduction

Emerging trends. Distributed object computing (DOC) middleware, such as CORBA, COM+, and Java RMI, is an advanced, mature, and field-tested paradigm that supports the composition of software objects that can be distributed or collocated throughout networked environments [1]. DOC middleware enables clients to invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals without hard-coding dependencies on the location, programming language, OS platform, communication protocols and interconnects, and hardware. In addition, DOC middleware simplifies distributed application development by automating key quality of service (QoS) properties, such as security, fault tolerance, and transactional semantics.

DOC middleware has matured to encompass a wide range of architectural styles (such as client/server and peer-to-peer) and application domains (such as e-commerce, process automation, aerospace, and telecommunications). Consequently, the environments in which distributed applications must operate – along with the demands made on applications by these environments – are now considerably more complex, heterogeneous, and dynamic. In particular, clients must increasingly interact with objects whose interfaces were unknown or perhaps did not even exist when the application was compiled or deployed [2]. The need for this capability has also grown due to recent standardization of the interaction between DOC middleware and scripting languages, such as Python [3] and CorbaScript [4], that require dynamic typing capabilities in the underlying DOC middleware.

Alternative middleware type systems. Developers of distributed applications that use statically typed programming languages, such as C, C++, and Java, are generally quite familiar with the statically typed capabilities provided by DOC middleware. They are often much less familiar, however, with the dynamically typed capabilities provided by DOC middleware. Sidebar 1 on page 2 briefly describes the differences between static and dynamic typing support for middleware.

Statically typed DOC middleware features generally yield high performance since efficient marshaling and demarshaling object code exists for even the most complex types. However, this performance is achieved at the expense of decreased flexibility and increased memory footprint. Applications that use statically typed middleware are less flexible since they cannot handle types not anticipated in the generated code. For example, any new interface or operation, even a change in a single operation parameter, triggers recompilation and relinking.

Conversely, dynamically typed middleware is often less efficient than static typing due to the extra overhead incurred by the dynamic type discovery and manipulation activities. However, dynamic typing's slower performance may be offset by its reduced footprint in the application, due to the replacement of compiled application-specific code by generic interpreted middleware code. In addition, the development cycle-time of applications that use dynamically typed middleware features can be reduced significantly for the following reasons:

- Scripting languages can be integrated with dynamically

Sidebar 1: Static vs. Dynamic Typing for DOC Middleware

A *type* can be defined an *abstractly* as a set (*e.g.*, an Integer type could be the set of all integers [5]) or *concretely* as a rule for interpreting a sequence of bits starting at some address in memory [6]. A *type system* [7] is a collection of rules for type equivalence, type checking, and type inference, along with methods to construct new types. DOC middleware type systems can be classified into two general forms:

- **Static typing**, which binds objects to their types at compile-time. Code for boilerplate activities, such as marshaling/demarshaling and operation dispatching, can be generated automatically since statically typed DOC middleware requires compile-time knowledge of all operation names and signatures.
- **Dynamic typing**, which binds objects to their types at run-time. Dynamically typed DOC middleware requires the explicit discovery and manipulation of operation names and signatures at run-time.

typed middleware, providing a way for developers to (1) create applications quickly and (2) modify the applications flexibly at run-time.

- New interfaces and operations can be introduced readily, without triggering recompilation and relinking.

The more a distributed system's topology resembles a star (*i.e.*, a central server and peripheral clients that do not talk directly to each other), the more likely a large amount of generated code will exist at the server location, since the server must then have knowledge of multiple, mostly disjoint, sets of operations. A dynamically typed implementation of such a topology need not include statically compiled code for each potential client, and can therefore handle a potentially unbounded number of unique operations and operation signatures.

R&D challenges. The work described in this paper was done in the context of The ACE ORB (TAO) [8], which is a widely adopted open-source implementation of CORBA. Our prior work on TAO has explored many dimensions of high-performance and real-time ORB design and performance, including scalable event processing [9], request demultiplexing [10], I/O subsystem [11] and protocol [12] integration, connection architectures [13], asynchronous [14] and synchronous [15] concurrent request processing, adaptive load balancing [16], meta-programming mechanisms [17], and IDL stub/skeleton optimizations [18].

Since its inception in 1996, the primary focus of research with TAO has been on high performance and predictable behavior in distributed real-time and embedded (DRE) systems, where statically typed applications have predominanted. What focus there has been on dynamic typing scenarios has been motivated by the intention to point out the performance penalties incurred by certain requirements of the OMG specification [19]. This paper extends our earlier work on statically typed DOC middleware by focusing on patterns and design techniques that address the following challenges of dynamically typed DOC middleware:

- Devising efficient and persistent techniques for dynamically storing and retrieving descriptions of interfaces and operations.
- Minimize the increase in footprint associated with CORBA dynamic typing capabilities.
- Assuring that applications that do not use dynamically typed middleware features do not incur time/space overheads.

Paper organization. The remainder of this paper is organized as follows. Section 2 describes the design of each dynamic CORBA feature in TAO, ranging from the most basic to the most advanced; Section 3 evaluates the dynamic CORBA capabilities provided by TAO and summarizes lessons learned from our experiences; Section 4 discusses other middleware research that is related to our work described in this paper; Section 5 summarizes areas of future work; Section 6 contains concluding remarks; and Section 7 expresses acknowledgements. For completeness, Appendix A provides details of the underlying data structures used to store dynamic type information in TAO.

2 The Design of Dynamic CORBA Capabilities

2.1 Overview of Dynamic CORBA

This section describes dynamic CORBA features and the design of dynamic typing capabilities provided by TAO. Figure 1 illustrates the key features that comprise dynamic CORBA. As shown in this figure, dynamic CORBA consists of the (1) TypeCode, which provides a structural type representation, (2) Any, which represents a value in dynamic CORBA applications, (3) NamedValue/NVList, which provide a dynamic representation of operation arguments and signatures, (4) Dynamic Invocation Interface (DII), which defines the client-side interface for dynamic CORBA applications, (5) the Dynamic Skeleton Interface (DSI), which is the server-side counterpart to the DII, (6) TypeCodeFactory, which is used to create types dynamically, (7) Dynamic Any, which is used to create and examine values in dynamic CORBA applications, (8) the Interface Repository (IFR), which is a distributed service that provides run-time access to CORBA type information, and (9) the IFR loader, which populates the Interface Repository with entries that correspond to IDL declarations.

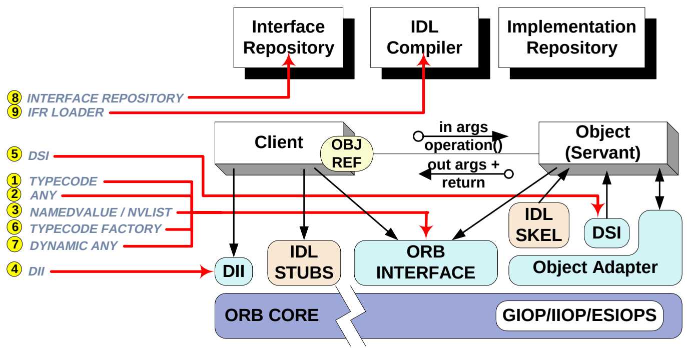


Figure 1: Features in Dynamic CORBA

The preceding synopsis outlines the features in dynamic CORBA, but does not explain what these features do in detail. More importantly, there is no motivation for *why* these features are important for middleware applications. The remainder of this section therefore explains why these features are needed in dynamic CORBA by explaining the key software development problems they solve, which include:

1. Representing type information at run-time
2. Defining a container for a value of any type
3. Assembling an operation's parameter list at run-time
4. Constructing a request at run-time
5. Handling a request of unknown signature
6. Creating new type representations at run-time
7. Composing and decomposing values of unknown type
8. Implementing run-time type discovery
9. Managing type information storage

Sidebar 2 presents the IDL example we use throughout this section to illustrate the use and interaction of the various dynamic CORBA features. We begin with the lowest level of dynamic CORBA features and proceed to the most advance dynamic CORBA features, each level building upon and subsuming the capabilities provided in the previous ones. This sequence matches the chronology shown in Figure 2. The item at left, labeled below the timeline, represents the dynamic CORBA features added to TAO by previous researchers in the DOC group. The items labeled above the timeline represent the contributions of the work described in this paper. The rightmost item in the figure refers to a topic of future research, which is discussed in Section 5.5.1.

2.2 Representing Type Information at Run-time

Context. An application that needs to be able to introspect on the types of its objects in order to function correctly.

Sidebar 2: IDL Example

The following OMG IDL is representative of an online music e-commerce system. We use it throughout this section to illustrate the use and interaction of various dynamic CORBA features.

```
interface Warehouse
{
  exception NotCarried {};
  struct format_info {
    float price;
    boolean in_stock;
  };
  struct title_info {
    format_info cd;
    format_info cassette;
  };
  typedef unsigned long sales_rank;
  title_info GetInfo (in string artist,
                      inout string title,
                      out sales_rank rank)
    raises (NotCarried);
};
```

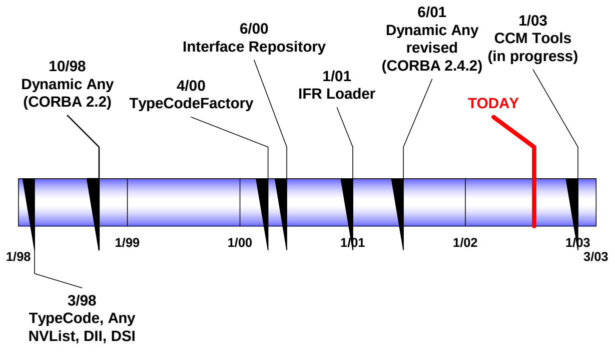


Figure 2: Timeline of Dynamic CORBA Features in TAO

Problem. While statically typed applications may occasionally require a way to represent a type at run-time, dynamically typed applications have no compile-time knowledge of at least some of the types it encounters. A dynamic CORBA implementation must therefore have a way to represent a type and to extract information from it at run-time.

Dynamic CORBA solution → TypeCode. Define a *TypeCode* that represents the structure of an IDL type. In dynamic CORBA, a TypeCode keeps track of the type of a value and a set (possibly empty) of parameters associated with that particular type. For example, the TypeCode for `title_info` in our example IDL file is shown in Figure 3. The left side of the figure shows the abstract representation of the TypeCode, with its type format and associated parameters. The right side of the figure shows the concrete representation, with the actual values corresponding to the associated IDL declara-

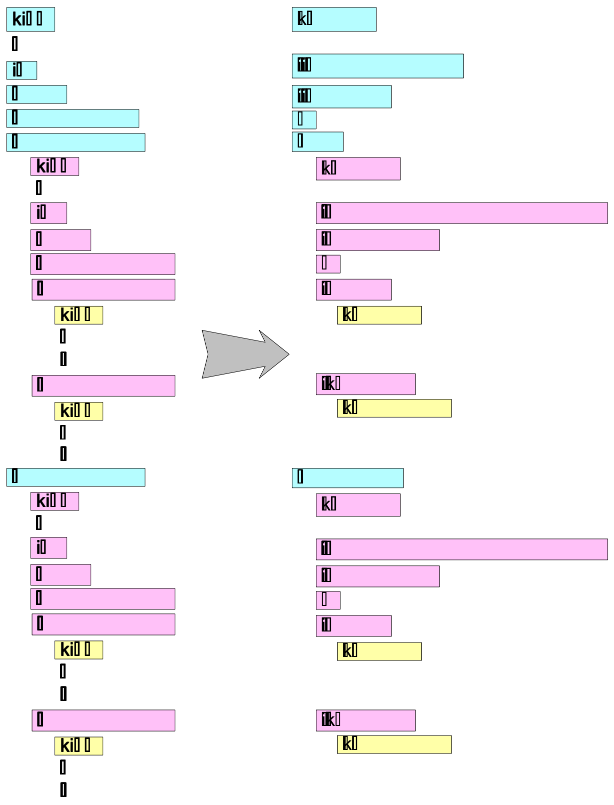


Figure 3: TypeCode

tions filled in. TypeCode constants for the built-in IDL types (such as `short`, `long`, and `double`) are generated by the ORB at startup. TypeCodes for user-defined static types (such as `structs` and `unions`) are generated by an IDL compiler [20] when it parses a file containing interface definitions.

Implementing the solution in TAO. In TAO, the `TypeCode` class stores the structural information in a buffer that is encoded using the *Common Data Representation* (CDR) format defined by the *Internet Interoperable Protocol* (IIOP) [20] required of all compliant ORBs. Class members holding the data in more accessible form, such as a list of member names or list of member types, are computed as necessary.

2.3 Defining a Container for a Value of Any Type

Context. An application that needs to be able to handle data whose type is known only at run-time.

Problem. Dynamically typed applications often encounter unknown types at run-time when they process constants or variables. In such cases, this unknown type will have an associated value. A dynamic CORBA implementation must be able to manipulate, *e.g.*, copy, compare, or pass as an argument, values whose type is unknown at compile-time.

Dynamic CORBA solution → Any. Define an *Any* as a generic container for a value of any type. In dynamic CORBA, an Any consists of a TypeCode and an opaque value of a type described by the TypeCode, as shown in Figure 4. The actual

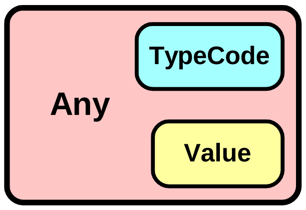


Figure 4: Any

representation of the value in memory can be left to the discretion of the ORB implementor, but it must be general enough to represent any type whatsoever *e.g.*, in C/C++ a likely candidate would be a `void` pointer. Dynamic CORBA provides standard operations to insert and access both the TypeCode and the value. Insertion and extraction of the value is accomplished by means of overloaded operators, which are provided by the ORB for the basic types. Like the TypeCode, an Any may be used by a statically typed application, and in such cases the IDL compiler generates the insertion and extraction operators for application-defined IDL types.

Implementing the solution in TAO. Anys in TAO are optimized in various ways. For example, when an Any is used in a statically typed application, the insertion operator generated by the IDL compiler for Anys of some aggregate type passes in a type-specific destructor for that type, eliminating the need for the Any to refer to its TypeCode when destroying its value. In addition, an Any in TAO may store its value either encoded in CDR form, as with the TypeCode, or as a `void` pointer, depending on the circumstances under which its value is inserted.

2.4 Assembling an Operation's Parameter List at Run-time

Context. A dynamic CORBA client application that needs to be able to pass parameters to an operation whose signature it does not know at compile-time.

Problem. If a dynamically typed application assigns a value to an operation parameter, the Any that might contain that value lacks a local name (such as ”artist” in our e-commerce example) and a direction (such as `in`, `out`, or `inout`) that a

CORBA operation argument has in addition to type and value. Every operation will have a well-defined set of these parameters. A dynamic CORBA implementation must therefore have a way to generically represent both a single operation parameter and an entire operation signature.

Dynamic CORBA solution → NamedValue and NVList. Define a *NamedValue* that represents an operation argument and an *NVList* that represents an entire operation signature. In dynamic CORBA, a NamedValue is a datatype consisting of an Any, a string name, and a flag indicating the direction, as shown in Figure 5. An NVList is a list of NamedValues, as shown in Figure 6. The types and order of the parameters are sufficient conditions for completely specifying the signature of an operation. It is therefore acceptable that the string name in a NamedValue be null. When a NamedValue is used to represent the return type of an operation, the direction and name are ignored. As shown in Section 2.5, an NVList can be used to construct the argument list of an operation incrementally at run-time.

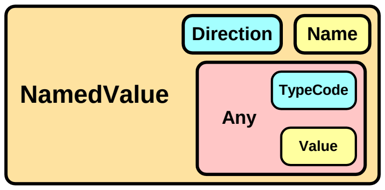


Figure 5: NamedValue

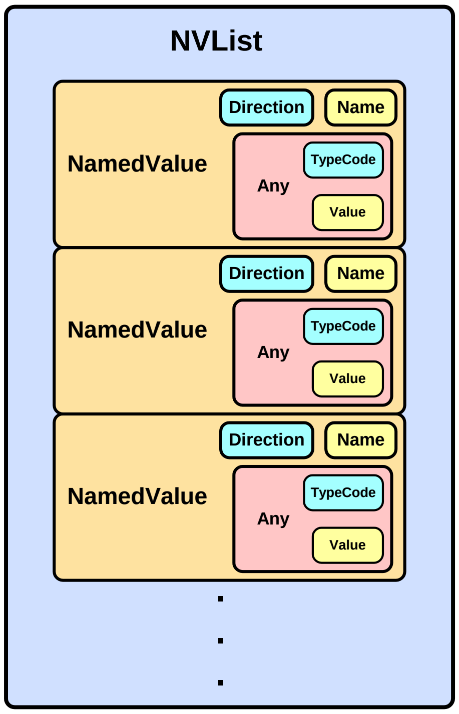


Figure 6: NVList

Implementing the solution in TAO. In TAO, some NVList methods can be passed a “lazy evaluation” parameter to optimize for cases where the list is empty or all the NamedValues contain Anys that store their values as a `void` pointer.

2.5 Constructing a Request at Run-time

Context. A dynamic client application that must invoke operations on objects whose IDL interfaces are not known until run-time.

Problem. Without compile-time knowledge of an IDL interface’s operations and their signatures, a dynamically typed application must explicitly construct and invoke a request at run-time [2]. A dynamic CORBA implementation must therefore be able to encapsulate information about a request and add to this information incrementally at run-time.

Dynamic CORBA solution → Dynamic Invocation Interface. Define a *Dynamic Invocation Interface (DII)* that enables a client to make a remote request on a target object for which no generated stub code exists. In dynamic CORBA, this interface is embodied in the *Request* object, which contains the target object reference, the operation name, an NVList containing the argument values, a NamedValue for the return value, and a list (if it is known at run-time) of possible user-defined exceptions that the operation may throw. All these entities are shown in Figure reffig:request. The `Request` object also provides methods for explicit invocation and for recovery of the return value, and the values of any `inout` or `out` arguments that may have been passed.

Using our music e-commerce example from Sidebar 2 on page 3, the following C++ code constructs and invokes a DII request, then extracts the results.

```
CORBA::Object_var target_obj = ...
  // get object reference from command line,
  // file, web page, etc.
CORBA::Request_var req =
  target_obj->_request ("GetInfo");
req->set_return_type (
    Warehouse::_tc_title_info
  );
req->add_in_arg ("artist") <<= "The Beatles";
req->add_in_arg ("title") <<= "Abbey Road";
CORBA::Any any (Warehouse::_tc_sales_rank);
req->arguments ()->add_value ("rank",
                              any,
                              CORBA::ARG_OUT);
req->exceptions ()->add (
    Warehouse::_tc_NotCarried
  );
req->invoke ();
```

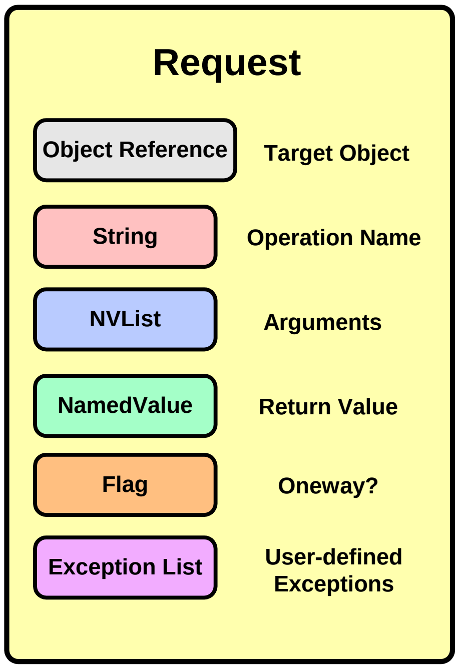


Figure 7: DII Request Object

```
const Warehouse::title_info *info;
req->return_value () >>= info;
CORBA::Any_ptr out_value =
  req->arguments ()->item (2)->value ();
CORBA::ULong sales_rank;
(*out_value) >>= sales_rank;
```

Identifiers prefixed with `_tc_` refer to TypeCodes, which are used to set the return type and to create the `out` argument, since the client does not assign values to these parameters.

Implementing the solution in TAO. A DII request can be sent asynchronously, *i.e.*, where the client does not block waiting for the reply. In such a case, the TAO `Request` object delegates this job to a *Reply Dispatcher* class, which is responsible for notifying the `Request` object when the reply has been received. Sidebar 3 describes the performance implications of DII verses the CORBA Static Invocation Interface (SII).

2.6 Handling a Request of Unknown Signature

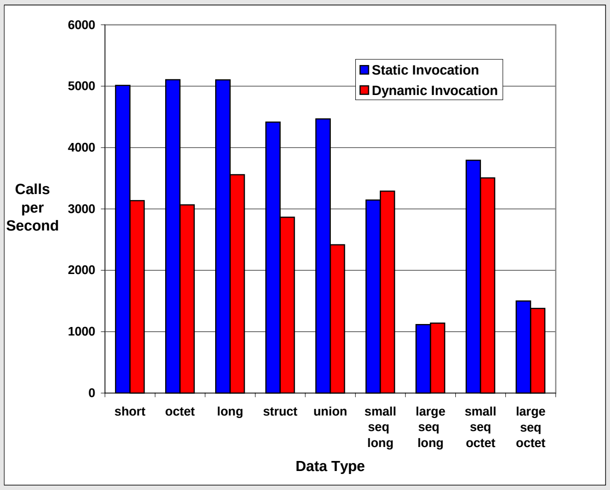
Context. A dynamic CORBA server application that needs to be able to handle requests whose signatures are not known at compile-time.

Problem. DII provides the means to assemble and invoke a request dynamically, and is therefore useful only to a client. A server in a dynamically typed application will need similar functionality in order to do its job. A dynamic CORBA implementation must provide a way to receive and handle requests without compile-time knowledge of operations or their signatures.

Dynamic CORBA solution → Dynamic Skeleton Interface. Define a *Dynamic Skeleton Interface* to provide functionality to a server that corresponds to what DII provides for a client. In dynamic CORBA, DSI takes the form of a pure virtual *Dynamic Implementation Routine* (DIR), defined in the ORB and overridden by the application. One of the DIR arguments is an instance of the class `ServerRequest`, which contains methods to demarshal the request arguments into the `NamedValue` and `NVList` containers described above, and

Sidebar 3: Performance Implications of DII

The extra overhead of request creation, plus addition of return type, arguments, and possibly exceptions results in a performance penalty. The following figure compares the throughput for regular SII CORBA requests and DII requests (the test was performed using TAO version 1.2.3, running on Windows 2000, and compiled by Visual Studio version 6.0).



The number of calls-per-second were computed by timing 250 roundtrip requests for each data type, in which the servant calculates the return value by cubing the input argument, or each element of the input argument if it is a sequence. Sequence lengths were set to 16 for octets and 4 for longs, to give an overall size of 16 bytes for small sequences. For large sequences, the length was set to 4096 for octets and 1024 for longs, to give an overall size of 4096 bytes. The figure above shows the greatest difference in performance for small request payloads, and more comparable performance as the request payload becomes larger. The overhead for marshaling and demarshaling is therefore more significant compared to the overhead for creating and populating the DII request.

to marshal the reply.

Implementing the solution in TAO. When the `ServerRequest` class was first added to TAO, it handled both static and dynamic invocations, despite of the fact that the DSI functionality of the class was not needed for the static case. This class was later factored into two parts:

- The original `ServerRequest` class, which now contained only DSI functionality, and
- A lightweight `TAO_ServerRequest` class.

Instances of this new class are now created by the underlying ORB transport on the server as (1) a stand-alone class to handle invocations statically and (2) a member of a `ServerRequest` instance to handle invocations dynamically.

2.7 Creating New Type Representations at Run-time

Context. A CORBA service or dynamic CORBA application that needs to create a type that was not known at compile-time.

Problem. A dynamic CORBA implementation must have the ability to create TypeCodes at run-time. Section 2.5 describes how to use a TypeCode to marshal and demarshal a value of unknown type. We have not yet addressed the issue of how TypeCodes may be created dynamically, *i.e.*, outside the aegis of an IDL compiler.

Dynamic CORBA solution → TypeCodeFactory. Define a *TypeCodeFactory* that bundles TypeCode creation methods for each IDL type into a single interface. Since there are many common use cases where dynamic typing (and therefore TypeCode creation) are not required, this interface can be compiled into a separate library that can be linked optionally by the application.

Implementing the solution in TAO. The OMG CORBA specification defines TypeCode creation methods in the `CORBA::ORB` interface. Whether TypeCode creation is required or not, these methods must still be declared and the linker must be satisfied. In TAO, we want to meet these requirements in all use cases without adding unnecessary size to the ORB when TypeCode creation is not used. These forces were resolved in TAO by using the Adapter pattern [21]. Rather than delegating the TypeCode creation call to the TypeCodeFactory directly, we delegate to an adapter class instead.

`TypeCodeFactory_Adapter` is an abstract class included in the compilation of the ORB. It contains pure virtual methods corresponding to each TypeCode creation method in `CORBA::ORB`. The TypeCodeFactory library contains a concrete class derived from `TypeCodeFactory_Adapter` that makes the actual call on the `TypeCodeFactory` interface. An instantiation of the Component Configurator pattern [22] is then used to load CORBA dynamic typing libraries at run-time. Figure 8 shows schematically how the component

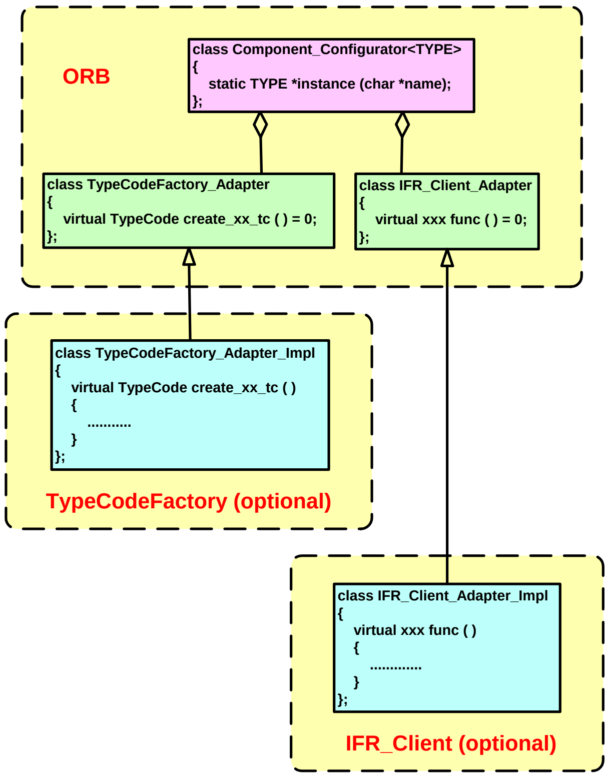

ORB
class Component_Configurator<TYPE>
{
static TYPE *instance (char *name);
};
class TypeCodeFactory_Adapter
{
virtual TypeCode create_xx_tc ( ) = 0;
};
class IFR_Client_Adapter
{
virtual xxx func ( ) = 0;
};
class TypeCodeFactory_Adapter_Impl
{
virtual TypeCode create_xx_tc ( )
{
....
}
};
TypeCodeFactory (optional)
class IFR_Client_Adapter_Impl
{
virtual xxx func ( )
{
....
}
};
IFR_Client (optional)

Figure 8: Component Configurator and Adapter

configurator works with an adapter by parametrizing it, and how the adapter makes it possible to add functionality to the ORB, yet only penalize applications that do not use this functionality with the addition of a few pure virtual functions.

The component configurator and adapter, as well as an abstract factory [21] work together to:

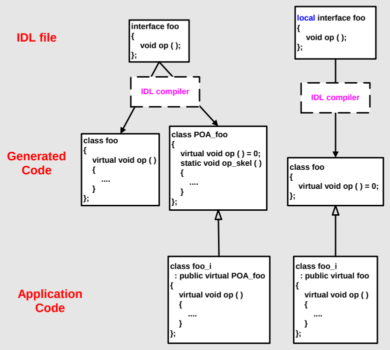
- Encapsulate TypeCode creation so it can be used by the ORB or other tools and optionally linked by the application
- Break the ORB dependencies on the TypeCode creation methods to avoid penalizing applications that do not require that functionality with additional ORB footprint and
- Load dynamic typing libraries on demand.

Although we have avoided a memory footprint penalty for applications that do not require a TypeCodeFactory, we would still like to minimize the added footprint for applications which do require it. By making the TypeCodeFactory interface local in the OMG IDL specification, the size of the generated code for TypeCodeFactory is greatly reduced, as shown in Sidebar 4.

Sidebar 4: Adding Locality Constraints

A Dynamic Any is intended to be a temporary entity, used for the sole purpose of composing or decomposing a standard Any when no static type information is available [23]. As such, it need not support remote creation, remote operation calls, export to a process other than the one in which it was created, creation of a stringified Interoperable Object Reference (IOR) or any of the common operations it inherits from the `CORBA::Object` interface. The CORBA specification prohibits Dynamic Anys from doing all these things, and specifies the exceptions to be raised if any of them are attempted [20]. Moreover, TypeCodeFactory is not useful to a remote ORB, which can call its own TypeCode creation methods. Consequently, the TypeCodeFactory and Dynamic Any classes have been implemented as `local` interfaces.

If an IDL interface is declared with the extra keyword `local`, none of its operations can be accessed remotely. In such cases, the IDL compiler does not generate a server class, and the application's implementation class inherits instead directly from the client-side generated class, as shown in the following figure.



```
interface foo
{
   void op ( );
};
```

```
local interface foo
{
   void op ( );
};
```

```
class foo
{
   virtual void op ( )
   {
      ....
   }
};
```

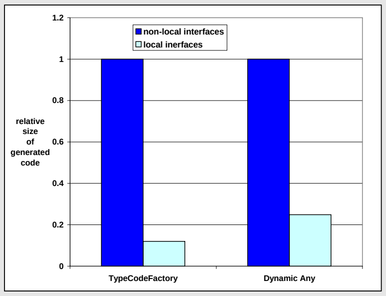
```
class POA_foo
{
   virtual void op ( ) = 0;
   static void op_skel ( )
   {
      ....
   }
};
```

```
class foo
{
   virtual void op ( ) = 0;
};
```

```
class foo_i
  : public virtual POA_foo
{
   virtual void op ( )
   {
      ....
   }
};
```

```
class foo_i
  : public virtual foo
{
   virtual void op ( )
   {
      ....
   }
};
```

Using `local` interfaces for Dynamic Any in TAO resulted in a significant footprint reduction in the generated code, compared to what would be generated by the IDL compiler for non-local interfaces, as shown in the following figure.



2.8 Composing and Decomposing Values of Unknown Type

Context. A dynamic CORBA application that needs to (1) assign a value to a variable of a type unknown at compile-time or (2) examine the contents of such an existing value.

Problem. When CORBA middleware uses static typing, there are overloaded Any insertion and extraction operators available for each known type. These operators insert or extract values all at once, *e.g.*, the insertion operator for a `struct` inserts the values of each member automatically and in order since the member insertion operators have already been defined and generated by the IDL compiler. With dynamic typing, however, the insertion or extraction of the value of an aggregate type must be recursively decomposed into basic types so the basic operators defined in the ORB can be used since generated specialized operators may not be available.

To support dynamic typing, incremental insertion or extraction is required. The logic of incremental Any insertion and extraction is the same as that for interpretive marshaling and demarshaling (see Section 4.5) and for some ORB implementations the same code may even be reused. Requiring an application to use such code adds to accidental complexity, however, and makes the application development process more tedious and error-prone. An application using dynamic typing should therefore be able to compose and decompose Any values incrementally, without being exposed to low-level ORB internals or implementation details.

Dynamic CORBA solution → Dynamic Any. Define a *Dynamic Any* hierarchy of types using the Facade pattern [21] to provide a consistent and portable interface for dynamic management of Any values while hiding the underlying ORB details from applications. An empty Dynamic Any may be created for composition by passing the appropriate TypeCode to an instance of *DynAnyFactory*, while for decomposition, the appropriate Any is passed to the factory.

Using Dynamic Anys, we can now expand our online music e-commerce example to include TypeCode creation and management of the Any values, both before and after the DII request, without recourse to the generated operators. First, we initialize an ORB and create TypeCodes for the exception and the typedef.

```
CORBA::ORB_var orb =
  CORBA::ORB_init (/* suitable args */);
CORBA::StructMemberSeq members;
members.length (0);
CORBA::TypeCode_var _tc_NotCarried =
  orb->create_exception_tc (
      "IDL:Warehouse/NotCarried:1.0",
      "NotCarried",
      members
    );
CORBA::TypeCode_var _tc_sales_rank =
  orb->create_alias_tc (
      "IDL:Warehouse/sales_rank:1.0",
      "sales_rank",
      CORBA::_tc_ulong
    );
```

Next we begin creation of the TypeCode for the complex return type. First, we create the TypeCode for the member type.

```
members.length (2);
members[0].name = CORBA::string_dup ("price");
members[0].type =
  CORBA::TypeCode::_duplicate (
      CORBA::_tc_float
    );
members[0].type_def = CORBA::IDLType::_nil ();
members[1].name =
  CORBA::string_dup ("in_stock");
members[1].type =
  CORBA::TypeCode::_duplicate (
      CORBA::_tc_boolean
    );
members[1].type_def = CORBA::IDLType::_nil ();
CORBA::TypeCode_var _tc_format_info =
  orb->create_struct_tc (
      "IDL:Warehouse/format_info:1.0",
      "format_info",
      members
    );
```

Now we use the member TypeCode to create the Typecode for the entire return type.

```
members[0].name = CORBA::string_dup ("cd");
members[0].type =
  CORBA::TypeCode::_duplicate (
      _tc_format_info.in ()
    );
members[1].name =
  CORBA::string_dup ("cassette");
members[1].type =
  CORBA::TypeCode:_duplicate (
      _tc_format_info.in ()
    );
CORBA::TypeCode_var _tc_title_info =
  orb->create_struct_tc (
      "IDL:Warehouse/title_info:1.0"
      "title_info"
      members
    );
```

As in the previous version, we must nest get the target object reference, create the DII request, and set the return type.

```
CORBA::Object_var target_obj = ...
  // get object reference from command line,
  // file, web page, etc.
CORBA::Request_var req =
  target_obj->_request ("GetInfo");
req->set_return_type (_tc_title_info.in ());
```

Instead of using Any insertion operators to add arguments to the request, however, we use Dynamic Anys.

```
obj =
  orb->resolve_initial_references (
      "DynAnyFactory"
    );
DynamicAny::DynAnyFactory_var factory =
  DynamicAny::DynAnyFactory::_narrow (
      obj.in ()
    );
DynamicAny::DynAny_var da =
  factory->create_dyn_any_from_type_code (
      CORBA::_tc_string
    );
da->insert_string ("The Beatles");
CORBA::Any_var string_any = da->to_any ();
req->arguments ()->add_value ("artist",
                              string_any.in (),
                              CORBA::ARG_IN);
```

Once a Dynamic Any is created, the type it contains cannot change, but it can be assigned another value of the same type.

```
da->insert_string ("Abbey Road");
string_any = da->to_any ();
req->arguments ()->add_value ("title",
                              string_any.in ()
                              CORBA:ARG_IN);
```

A Dynamic Any created by the DynAnyFactory must be destroyed when it has served its purpose.

```
da->destroy ();
```

Since an `out` argument is not assigned a value before invocation, we can add it using only the TypeCode.

```
CORBA::Any alias_any (_tc_sales_rank.in ());
req->arguments ()->add_value ("rank",
                              alias_any,
                              CORBA::ARG_OUT);
```

As before, we add the exception to the request and invoke it.

```
req->exceptions ()->add (_tc_NotCarried.in ());
req->invoke ();
```

Now we extract the return value, but this time we do it incrementally using Dynamic Anys. First we create a Dynamic Any from the Any return value of the request.

```
DynamicAny::DynAny_var da_retval =
  factory->create_dyn_any (
      req->return_value ()
    );
DynamicAny::DynStruct_var ds_retval =
  DynamicAny::DynStruct::_narrow (
      da_retval.in ()
    );
```

Then we extract the first member and access its values.

```
DynamicAny::DynAny_var da_member =
  ds_retval->current_component ();
DynamicAny::DynStruct_var ds_member =
  DynamicAny::DynStruct::_narrow (
      da_member.in ()
    );
CORBA::Float cd_price =
  ds_member->get_float ();
CORBA::Boolean cd_in_stock = 0;
if (ds_member->next ()) {
  cd_in_stock = ds_member->get_boolean ();
}
```

Finally, we advance to the second member and repeat the process.

```
CORBA::Float cassette_price = 0;
CORBA::Boolean cassette_in_stock = 0;
if (ds_retval->next ()) {
  da_member = ds_retval->current_component ();
  ds_member =
    DynamicAny::DynStruct::_narrow (
        da_member.in ()
      );
  cassette_price = ds_member->get_float ();
  cassette_in_stock =
    ds_member->get_boolean ();
}
da_retval->destroy ();
```

We also use a Dynamic Any to extract the value of the `out` argument.

```
DynamicAny::DynAny_var da_out =
  factory->create_dyn_any (
      *req->arguments ()->item (2)->value ()
    );
CORBA::ULong sales_rank = da_out->get_ulong ();
da_out->destroy ();
```

Implementing the solution in TAO. The TAO implementation of Dynamic Any uses the Composite pattern [21] to facilitate incremental composition and decomposition of Dynamic Anys. This implementation requires that any member of a Dynamic Any containing a non-basic type itself be a Dynamic Any. As with TypeCodeFactory, the TAO Dynamic Any implementation is contained in a separate library that can be optionally compiled and linked.

As shown in Figure 2, some time after the original implementation of Dynamic Any in TAO, a large number of changes in the CORBA specification of Dynamic Any necessitated an extensive reimplementation. At this time, we reduced the size of the Dynamic Any library considerably by modifying the inheritance structure. The upper diagram in Figure 9 shows the original inheritance structure of the implementation classes, while the bottom diagram shows the modified version, which uses an intermediate non-instantiated class to contain common code. In Figure 10, we see that the size of the Dynamic Any library was reduced by over 40% as a result of the modified class inheritance structure. The size of the Dynamic Any library was further reduced by adding locality constraints to the IDL interfaces, as shown in Sidebar 4 on page 8.

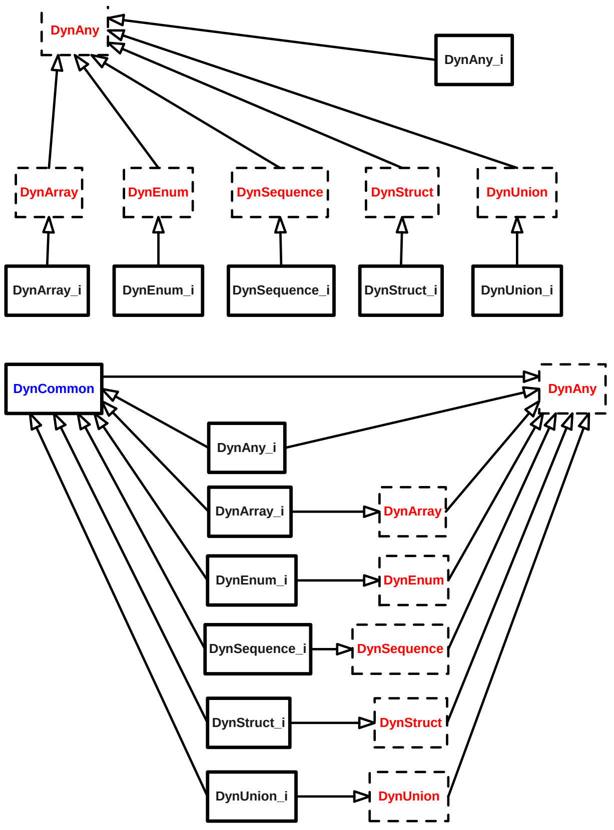


Figure 9: Dynamic Any Inheritance Alternatives

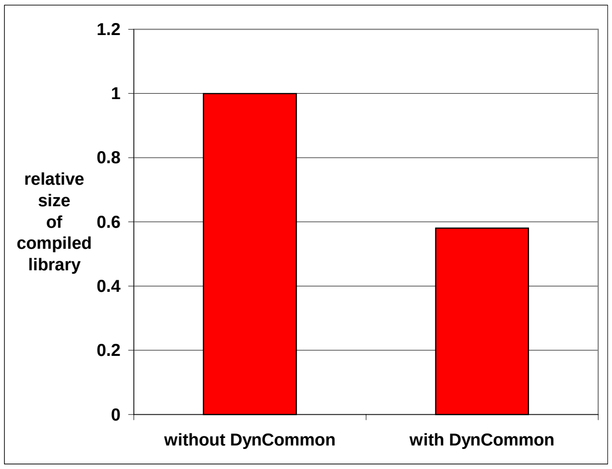


Figure 10: Dynamic Any Footprint Reduction

2.9 Implementing Run-time Type Discovery

Context. An application that needs information about the operations of some object reference.

Problem. Section 2.8 showed how the TypeCodeFactory can be used to create a TypeCode for any legal IDL type, and also showed how Dynamic Anys can be used to compose or decompose a regular Any that corresponds to that type. Likewise, Section 2.5 showed how DII can be used to create and execute an invocation of any signature legal for an IDL operation. It is rarely useful, however, to create types, typed values, or operation signatures without guidance from any external information. Moreover, as useful as Anys are as generic containers, they create a potential problem by allowing the possibility of bypassing a compiler's built-in type checking and enforcement mechanisms.

To solve these problems, some way of obtaining external type information is required. There are several possible ways to get such information, ranging from a translation table that uses rules to map one type system to another, from an event channel that carries type information about its clients (such as the CORBA Notification Service), or from a repository [23]. Such a repository should be persistent, updateable, and located at a well-known address.

Dynamic CORBA solution → A Repository For Type Information. Define an *Interface Repository* (IFR) service that provides CORBA type information at run-time. The location of a CORBA IFR can be resolved by the ORB in a manner similar to other CORBA services, *e.g.*, via a Naming Service or Trading Service. It stores its information in the form of *Interface Repository Objects* (IR Objects) that derive directly or indirectly from the abstract base class `CORBA::IRObject`, which is derived from `CORBA::Object`. The operations defined in the various concrete IR Object classes are the means by which the repository can be queried and updated. The repository itself is also an IR Object.

Using the CORBA Interface Repository, we can now update our online music e-commerce example to be completely free of reliance on outside knowledge, other than that obtained from the Interface Repository. After initializing the ORB, we resolve the IFR just as we did with DynAnyFactory.

```
CORBA::Object_var repo_obj =
  orb->resolve_initial_references (
      "InterfaceRepository"
    );
CORBA::Repository_var repo =
  CORBA::Repository::_narrow (repo_obj.in ());
```

We then get the repository id of the target object, and use it to look up the target object definition in the repository.[1]

[1]Naturally, CORBA programmers should always check if the result of a `_narrow()` operation is 0, but we leave such things out in this example in the interest of brevity.

```
const char *repo_id =
  target_obj->_interface_repository_id ();
CORBA::Contained_var contained =
  repo->lookup_id (repo_id);
CORBA::InterfaceDef_var obj_def =
  CORBA::InterfaceDef::_narrow (
      contained.in ()
    );
```

Alternatively, we can try to get the interface definition from the target object directly by replacing the above lines with

```
CORBA::InterfaceDef_var obj_def =
  target_obj->_get_interface ();
```

Next we want to get a list of the interface's operations, so we call for its contents and limit the resulting list to operations only.

```
CORBA::ContainedSeq_var operations =
  obj_def->contents (CORBA::dk_Operation,
                     0);
```

The second argument to `contents` indicates that we are not excluding inherited operations in our list. Now that we have a list of our target object's operations, we must have some criteria to tell us which operation is the one we want. We assume that such criteria are in place in the application, and iterate over the operation list.

```
CORBA::ULong length = operations->length ();
CORBA::OperationDef_var target_op;
for (CORBA::ULong i = 0; i < length; ++i) {
  if (/* criteria are satisfied */) {
    target_op =
      CORBA::OperationDef::_narrow (
          operations[i].in ()
        );
    break;
  }
}
```

Now we can create a DII request, using the operation's name.

```
CORBA::String_var op_name =
  target_op->name ();
CORBA::Request_var req =
  target_obj->_request (op_name.in ());
```

We can also set the return type and populate the request's exception list. The Interface Repository creates the TypeCode by calling the TypeCodeFactory.

```
CORBA::TypeCode_var rettype =
  target_op->result ();
req->set_return_type (rettype.in ());
CORBA::ExceptionDefSeq_var op_excepts =
  target_op->exceptions ();
length = op_excepts.length ();
CORBA::TypeCode_var except_type;
for (CORBA::ULong j = 0; j < length; ++j) {
  except_type = op_excepts[i].type ();
  req->exceptions ()->add (except_type.in ());
}
```

Next we query the operation definition for the parameter list.

```
CORBA::ParDescriptionSeq_var op_params =
  target_op->params ();
```

The type `ParDescriptionSeq` is a sequence of structs, each containing the parameter's description, *e.g.*, its name, type, and direction. From this information, we can create a Dynamic Any for each argument and assign the value as we saw in the previous version of our example. We can find out more about the operation's return type as follows:

```
CORBA::IDLType_var ret_def =
  target_op->result_def ();
```

Likewise, we can further query `ret_def` for its structure to assist in incrementally extracting the member values of the complex return type in our example. Note that the code example is now free of the arbitrary, hard-coded values and actions that it contained in the previous two versions.

Figure 11 shows a UML sequence diagram of a distributed scenario consisting of a client, a server, an interface repository, and an interface repository loader (with an associated IDL file), all remote from each other. Although this scenario is only one of several possible configurations and action sequences, it serves to show how all the dynamic typing capabilities work together. The object name boxes at the top of each lifeline are color coded, with objects of like color being necessarily collocated. TypeCodes and Anys are also objects in this scenario, but since no calls are made on any of their methods, they are not given lifelines of their own.

Note that the client creates the DII request while the interface repository is being updated. It is shown this way in the figure to save space, and in an actual execution would cause no problem, as long as the update is completed before the client queries the repository. In the interest of brevity, the complete set of calls to the Interface Repository and to Dynamic Anys has been collapsed, and the extraction of the return value has been omitted.

Implementing the solution in TAO. The TAO IFR implementation is a large and complex piece of software, with the source code generated from the OMG IDL specification alone totaling over 100,000 lines. Below, we present the design challenges encountered in the design and implementation of the TAO IFR and explain how these challenges were met.

• **IFR design challenges.** Like TypeCodes and Dynamic Anys, the Interface Repository must be able to deal with composite types that may be deeply nested and complex. The operations that retrieve items from the repository and create new entries in the repository both have CORBA objects as their end products. The repository may be required to have a long lifetime, longer than that of an ORB or an application process. The design of the Interface Repository therefore requires the resolution of the following forces:

- Efficient storage and retrieval of complex nested types
- OO database characteristics
- Persistent storage option

Since IR Objects contain information about the definition of other objects, they may be viewed as meta-objects that contain the information in their state. The Memento pattern was used to externalize and record this state without violating encapsulation [21]. This pattern also allows restoration of the meta-object from its recorded state, reversibility of transactions, and the storage of incremental changes in state. The underlying container of the repository uses hash tables for efficient retrieval of the meta-object state, and a memory map for persistence. The globally unique repository id that every IDL named type declaration has can be used, if is known, as a key for storage in a special index section for increased lookup efficiency.

• **IFR underlying container data structures.** To mimic the nested structure of the data contained by Interface Repository, its underlying container takes the form of a tree-like hierarchy of hash tables. Each table can contain both values (stored as integers, strings or binary chunks of specified size), and other tables (stored as string key names) called *sections*, each of which is the root of a subtree, as shown in Figure 12.

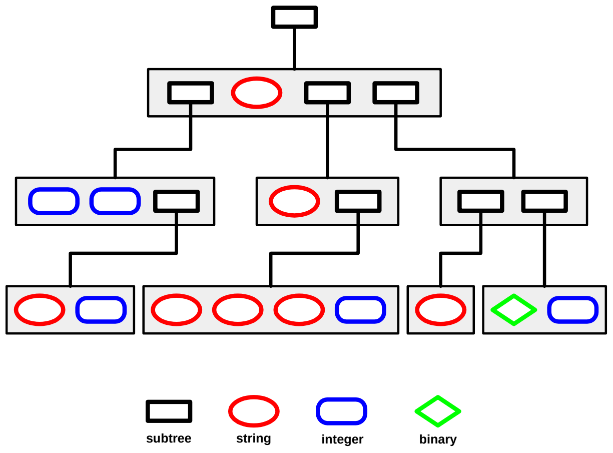


Figure 12: Underlying Container Structure

• **Request dispatching.** The CORBA Portable Object Adapter (POA) is a server-side entity that matches requests to servants. There may be several POAs in a server process, each with one or more servants that are registered with it, and each created with zero or more policy values that govern facets of its behavior. When the Interface Repository service starts up, a POA is created to manage it. This POA is created with the PERSISTENT policy, which enables the use of a backing store

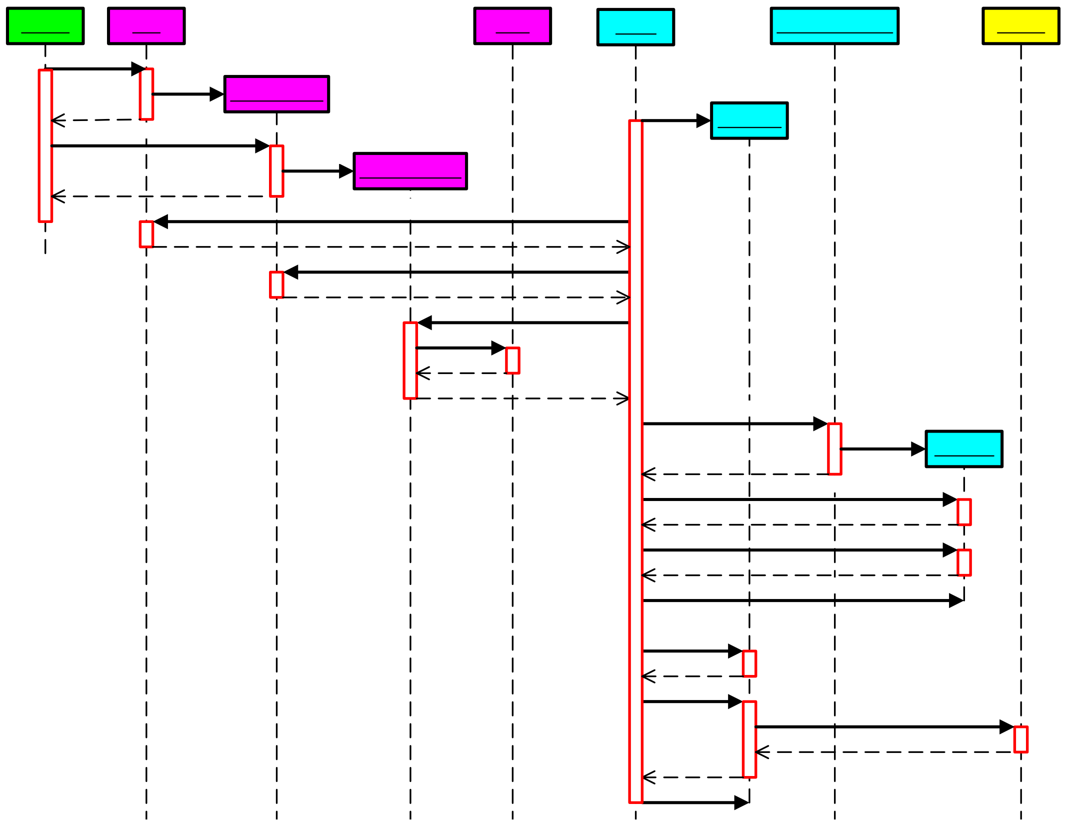


Figure 11: Client Request Using Dynamic Typing

to aid in restoration of state in the event of a server crash. Another POA is created to manage the contents of the repository. It also has the PERSISTENT policy, as well as three others:

- USER_ID—When an object is registered with its POA, the POA requires the ObjectId portion of the IOR to be externally supplied instead of being created by the ORB. We will see how this ObjectId is used to create servants and to locate an entry in the repository.
- USE_SERVANT_MANAGER—The POA does not dispatch to a servant already in its object table, but instead creates a servant. The servant may be created once for the lifetime of the POA (this option is called *Servant Manager*) or created and destroyed for each request (this options is called *Servant Locator*).
- NON_RETAIN—Selects the Servant Locator option above.

• **Servant creation and lifecycle** The Servant Locator option creates a servant for each method invocation, by subclassing `PortableServer::ServantLocator` and overriding its methods `preinvoke` and `postinvoke`. Below is a simplified version of the override in our implementation.

```
class IFR_ServantLocator
  : public PortableServer::ServantLocator
{
public:
  virtual PortableServer::Servant preinvoke (
      const PortableServer::ObjectId &oid,
      PortableServer::POA_ptr poa
    )
  {
    CORBA::String_var s =
      PortableServer::ObjectId_to_string (oid);
    ACE_Configuration_Section_Key root_key =
      this->repo_->root_key ();
    ACE_Configuration_Section_Key servant_key;
    ACE_Configuration *config =
      this->repo_->config ();
    config->expand_path (root_key,
                         s.in (),
                         servant_key);
    IFR_Servant_Factory *factory =
      this->repo_->servant_factory ();
    PortableServer::Servant servant =
      factory->create_servant (servant_key,
                               poa);
    return servant;
  }
  virtual void postinvoke (
      PortableServer::Servant servant
    )
  {
    delete servant;
  }
private:
  TAO_Repository_i *repo_;
};
```

The ObjectId of a repository entry is composed from its IDL scoped name, using backslashes for separators. In our online music e-commerce example, the IR Object corresponding to the operation `GetInfo()` would have an ObjectId formed by the following:

```
PortableServer::ObjectId *oid =
  PortableServer::string_to_ObjectId (
      "Warehouse\GetInfo"
    );
```

There is a one-to-one mapping between an entry's scoped name and the path name from the repository root to the entry's location, and we can convert in either direction using a string like the one above as an intermediate. The repository container method `expand_path` takes the string and returns the internal key corresponding to the entry's location. Appendix A contains more extensive coverage of the Interface Repository's internal structure and layout.

After we have the entry key, we pass it to a servant factory, which reads the `CORBA::DefinitionKind` value (see Appendix A) stored in the section corresponding to the key. The servant factory then creates an instance of the appropriate class based on the `DefinitionKind`, passing in the key as state. The resulting servant now represents the repository entry and is ready for the upcall.

• **IR object creation and lifecycle** A reference to an IR Object may be obtained by creating a new entry in the repository (in which case a reference to the new entry will be returned to the caller) or by querying an existing IR Object reference. The IR Object reference obtained by either method will be valid until the Interface Repository service is shut down or until the repository contents are modified in a way that changes the path in the repository tree from the root to the entry corresponding to the IR Object. If this happens, the path string constituting the ObjectId is no longer valid, and a fresh object reference must be obtained.

2.10 Managing Run-time Type Information

Context. Any domain where an Interface Repository will be used to provide interface definition information.

Problem. An Interface Repository must have associated with it one or more mechanisms to add, remove, and update its contents. Although there may be many sources for the information that resides in an Interface Repository, the most common source of input is conversion from IDL declarations. An Interface Repository loader that accomplishes such conversions would need to parse these declarations in the manner of an IDL compiler, although its subsequent actions would be quite different. When processing IDL declarations, such a mechanism should reuse the parsing engine of the IDL compiler.

Dynamic CORBA solution → IFR Loader. Create an *IFR loader* that translates the contents of an IDL file into Interface Repository entries.

Implementing the solution in TAO. The TAO IFR loader shares an IDL parsing engine with the TAO IDL compiler. To reuse the IDL compiler's parsing engine, it was first necessary to modify the IDL compiler itself, which had previously been compiled and linked as a monolithically program. To improve its flexibility, it was refactored [24] into three components:

- **Front-end (FE) library**, which is resuable software for validing grammar and syntax, and for building the AST.
- **Back-end (BE) library**, which is a pluggable library for specialized functions, *e.g.*, code generation or Interface Repository management.
- **Top-level executable**, which is a thin layer for preprocessing, parsing command line arguments, initialization, and execution. This component can vary with the back-end library since it must be aware of the command-line arguments recognized by the particular back-end that is used.

Figure 13 shows how the IDL compiler has been refactored and then extended to serve multiple purposes. Like the TAO IDL compiler, the back-end library of the TAO Interface Repository loader uses the Visitor pattern [21] to traverse the AST and perform its actions through the usual scheme of double dispatching with virtual methods. The specific action taken depends on the most derived type of both the AST node and the visitor.

By default the loader creates entries in the repository corresponding to declarations in the processed IDL file, but it may also be set by command-line argument to remove the entries, if they are found in the repository. Managing the contents of the interface repository in this way enhances the repository's built-in error checking, which is defined by CORBA exceptions specified for various error cases, but is by no means a foolproof guard against IDL errors and inconsistencies.

The IDL parser can detect errors within a single IDL file early, before any remote calls on the repository are made. Using the create/destroy portion of the CORBA Interface Repository API in an ad hoc manner, on the other hand, provides no such early detection of IDL errors, and may necessitate a transaction reversal in order to keep the repository in a correct

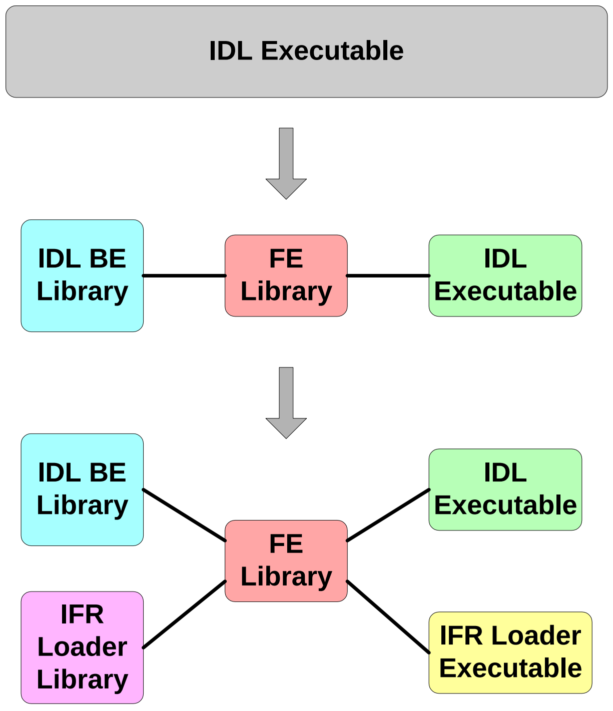


Figure 13: IDL Compiler Modularization

and consistent state. In any event, if such a transaction reversal does become necessary, the entry-removing option of the Interface Repository loader makes it a simple matter to back out of a transaction at the file level.

3 Evaluation and Lessons Learned

This section evaluates the dynamic CORBA capabilities provided by TAO and summarizes lessons learned from our experiences.

3.1 Evaluation of Dynamic Typing Capabilities

Section 2 presented a wide range of dynamic typing capabilities and demonstrated their use. Based on our experience implementing and using these capabilities, we have observed the possibilities, tradeoffs, and limitations of dynamic typing capabilities described below.

Interdependency. The order in which the various dynamic typing capabilities have been presented in this paper can also be viewed as a progression of dependency. For example, an Any cannot exist without a TypeCode, and all the other dynamic typing capabilities depend on these two. In some cases, the dependency is not absolute, but nevertheless imposes significant limitations. Dynamic Anys do not depend on DII (or NamedValue and NVList, on which DII itself depends), but their use will be circumscribed, since the end product of Dynamic Any composition (an Any) can then be used only in a static invocation. If an application can make a static invocation with an Any in the argument list, it is likely that the generated insertion and extraction operators for that Any are present as well, which would probably eliminate the need for Dynamic Anys in the first place. Likewise, the Interface Repository does not depend on DII, but the results of IFR queries are of limited use unless they are targeted for the assembly of a dynamic invocation.

Hybrid Applications. Although the examples and comparisons in this paper have presented applications as either wholly statically types or wholly dynamically typed, in a real application this need not be the case. An application may use dynamic typing in only some areas, or it may use a subset of dynamic typing capabilities, subject to the dependencies mentioned previously. An application may also have some “out-of-band” source of type information at run-time. This source may replace one or more dynamic typing capabilities.

Overhead vs. Generality. An application is most adaptable and flexible when it uses the complete set of dynamic typing capabilities. As we have seen in Figure 14, this choice may nearly double the size of the middleware that must be linked by the application, compared to one that uses static typing. We have also seen from the example code in Sections 2.5 and 2.8 how the size of the application code itself may grow, as well as the indirection of function calls. Although minimizing size and indirection are worthwhile, there are many cases where their sacrifice is an acceptable tradeoff, when adaptability and flexibility are the paramount concerns.

Portability. The dynamic typing capabilities mentioned in this paper have been part of the CORBA specification for some time. Any CORBA-compliant ORB will provide a complete set, and applications that run on multiple ORB implementations should find few if any restrictions in choice of ORB vendor.

3.2 Lessons Learned

Below, we present the key lessons learned from our experience in the design and use of dynamic typing capabilities in CORBA.

ORB Footprint Management. Not every ORB or application will require all, or even any, of the dynamic typing capabilities. In such cases, we wish to avoid penalizing an application with extra footprint for capabilities it does not use. The Interface Repository service can be compiled and run in a separate process, as is the case with other common CORBA services, but it would be advantageous for the ORB to know about the client-side portion of the Interface Repository, in cases where the ORB has no IFR of its own running but wishes to connect to a remote IFR. We can have the best of both worlds by separating the client-side and server-side portions of the IFR, and compiling the client-side portion in a separate library, as with TypeCodeFactory and Dynamic Any. An application can optionally link only to this library, while the IFR service must always link to it.

However, there are dependencies between the ORB and the client-side portion of the IFR that must be circumvented before this separation of libraries can be achieved. As shown in Figure 8, the solution for TypeCodeFactory has also been applied to the IFR client library, with an abstract/concrete set of adapter classes and a Component Configurator to make sure that only pure virtual functions are added to the ORB if the application does not need to use an Interface Repository.

Figure 14 shows the relative sizes of all the dynamic typ-

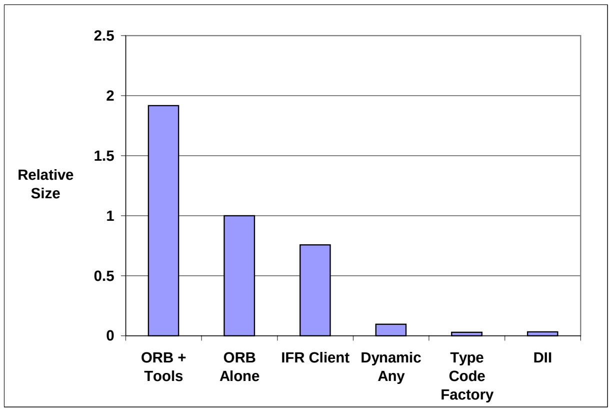


Figure 14: Relative Sizes of ORB and Dynamic Typing Capabilities

ing capabilities that have been mentioned so far, along with the size of the ORB both with and without them. The size of the ORB without any dynamic typing capabilities has been normalized to 1.0. The graph shows that the size of the ORB nearly doubles when all the dynamic typing capabilities are linked in. By selecting only the subset of capabilities needed, developers can keep the size of their applications to a minimum.

Software Reuse. It is widely accepted that software reuse results from the use of design patterns. However, software reuse may also be accomplished outside the application of any design pattern. For example, the modularization of the TAO IDL compiler described in Section 2.10 enabled the reuse of the code which parses IDL and constructs an AST. This code now resides in its own library, and is available for use with code-generating backends targeting other languages, such as Java or XML.

ORB Subsetting. DII/DSI, TypeCodeFactory, Dynamic Any and the Interface Repository have all been implemented in TAO to be compiled into separate libraries. This practice has been repeated with other components of TAO, and is a more flexible alternative to the Minimum CORBA specification [20]. Subsetting work in TAO is ongoing, and is further described in Section 5.2.

4 Related Work

This section compares and contrasts our work with representative related work. Sections 4.1 and 4.2 each describes non-CORBA research that is a precursor to elements of the OMG specification related to dynamic typing. Section 4.3 compares some features of the Interface Repository to a similar entity in Microsoft COM. Section 4.4 describes a scripting language that uses the dynamic typing capabilities mentioned in this paper. Section 4.5 compares the different marshaling mechanisms used with static and dynamic typing. Finally, Section 4.6 places dynamic typing in a larger context, as one of a group of meta-programming mechanisms.

4.1 Dynamic Type and Value Representation

The idea of combining an object's value along with a type description in a generic type container has been studied for many years. For example, in 1989, Abadi et al [25] introduced a data type called *Dynamic* as an aid to type determination at run-time. This data type was a two-tuple, consisting of a binary representation of the data object and a representation of its type. The Dynamic design was prescient of the CORBA Any.

4.2 Self-Describing Objects

The virtual method `get_interface()` defined in `CORBA::Object` enables any CORBA object to be queried about its own definition. The object does not physically contain the description information, but instead resolves an interface repository (if one is running that has registered with the same ORB to which the object's POA is registered) and passes in its repository id for lookup.

In [26], Muckelbauer and Russo describe how they developed the *Renaissance Interface Description Language* (RIDL). An RIDL specification is translated into a target language, such as C++, and then used by an object to describe itself when its implicitly supported `signature()` method is called. The `CORBA::Object::get_interface()` method is much more flexible. Since `get_interface()` accesses an interface repository, a `CORBA::InterfaceDef` IR Object is returned, which may in turn be queried to retrieve information about the interface's operations, attributes, typedefs, or other IDL types the interface may contain, as well as a list of ancestors. The query may also be made with options that mask the type of contents reported or exclude inherited operations.

4.3 Microsoft COM Automation

Automation is a feature of Microsoft COM [27] that makes it easier for macro and interpretive languages (usually Visual Basic or Visual Basic Script) to access COM components [28]. To make the access useful, COM Automation provides run-time type information in a *type library*. A type library can be created by calling the COM Automation method `CreateTypLib` and using the resulting `ICreateTypeLib` interface to populate the type library. However, a type library is more often created from an IDL file by the Microsoft IDL (MIDL) compiler, using Microsoft extensions to the Distributed Computing Environment (DCE) specification [29].

The Microsoft COM type library is similar in purpose to the CORBA Interface Repository, and is likewise populated with IDL declarations. However, a type library may also uses Microsoft's extensions to OMG IDL. These extensions include the *library* keyword, which can label sections of an IDL file for compilation into the type library (unlabeled sections are excluded) and a provision for including help strings in an IDL file.

Microsoft COM type libraries are more limited than the CORBA Interface Repository in many ways. For example, each COM type library is populated from the contents of a single IDL file, and may not be combined into a larger repository that can be used as a warehouse of type information. In addition, the TAO Interface Repository loader has a command-line option that causes the contents of the IDL file to be removed from the repository. This option, when used in conjunction with the normal adding mode, provides a safe way to update the IFR with a new version of an IDL file.

4.4 CorbaScript

CorbaScript is an object-oriented interpretive language that can be used to create generic applications whose behavior can be determined at run-time [30]. It uses Dynamic Anys, DII/DSI, and the Interface Repository to construct CORBA clients and servers directly from IDL decarations, without need for compiled stub and skeleton code. Integrating CorbaScript with TAO proved to be an excellent regression test to detect inconsistencies and incompatibilities with TAO's Interface Repository and Dynamic Any implementation.

4.5 Compiled vs. Interpretive Marshaling

TAO sends requests statically using a technique called *compiled marshaling* [31]. Compiled marshaling is performed via overloaded operators that are generated by TAO's IDL compiler for each new IDL type defined in an application. Other code generated by the IDL compiler uses these operators to marshal and demarshal operation parameters automatically. Since both client and server have compiled this generated code, there is no need for the application to do any type investigation at run-time, at either endpoint.

The dynamic alternative to compiled marshaling is called *interpretive marshaling*. In interpretive marshaling, the application uses methods provided by the ORB to traverse the tree-like structure of a TypeCode at run-time, doing so recursively until basic IDL types are discovered, at which point the ORB-defined operators for the basic types can be used to marshal or demarshal the values. This process corresponds to the IDL compiler's generation of specialized operators, except that the IDL compiler traverses part of the Abstract Syntax Tree (AST), while the generic ORB code traverses the TypeCode.

Comparison of compiled and interpretive marshaling reveals the classic time/space tradeoff. With compiled marshaling, we save the time used for tree traversal by moving it to compile-time and delegating it to the IDL compiler, and the price we pay is the extra generated code, which grows with each new declared data type. Interpretive marshaling, by doing the traversal at run-time, can use one-size-fits-all code that is centrally located in the ORB and available to any number of applications [18].

Besides the time/space tradeoff, there is another difference between compiled and interpretive marshaling, and thus also between static and dynamic typing – an important difference that is the primary motivation for the work described in this paper. An application that uses only compiled marshaling must also use static typing, and cannot therefore handle any operations that are not defined at compile-time. Conversely, an application that uses the ORB's general-purpose code for interpretive marshaling may still limit itself to static typing, but it need not do so. It may instead use the dynamic typing capabilities described in Section 2 to handle an unlimited set of interfaces, parameter types, and operation signatures. Such an application can adapt, be open-ended, and even evolve over time.

4.6 Middleware Meta-Programming Mechanisms

The dynamic typing capabilities described in this paper form part of a larger group of middleware-based meta-programming mechanisms that improve the adaptability of distributed applications. Examples of these meta-programming mechanisms include [17]:

- **Smart proxies**, which modify interface behavior without requiring application re-implementation.
- **Portable Interceptors**, which are a standard CORBA feature that can be used at specified points in the end-to-end invocation path.
- **Pluggable protocols**, which are used to decouple an ORB's transport protocols from its higher-level components.
- **Bridges**, which are used to connect ORBs residing in different domains or systems running different middleware technologies.

These meta-programming mechanisms can be used by both static and dynamic CORBA applications. We therefore do not focus on them in this paper since our emphasis is on dynamic CORBA features.

5 Future Work

Although the work presented in this paper covers a timespan of over three years, there are many more R&D issues that remain to be addressed, both in the area of dynamic CORBA in general and in the area of dynamic typing in particular. This section presents an overview of these topics, and, where possible, some speculation about fruitful approaches.

5.1 Performance Optimization

As seen in Section 2.9, queries or updates to the TAO Interface Repository presently use the Servant Locator option to create temporary servants for each call. Such a scheme is an efficient way to encapsulate a logical OO database entry on the fly. It requires that servants be created by a factory, however, where heap allocation must be used, thereby degrading performance.

Since the set of operations of all IR Object classes is static, the locations of method implementations in memory could be stored in a table, and the operations dispatched using a perfect hashing scheme [32]. Using this method of operation dispatch, one generic servant (whose lifetime parallels that of the repository) could handle all queries and updates, eliminating the need for heap or even stack allocations for servant creation with each call.

As discussed in Section 4.5, interpretive marshaling in an ORB does not perform as well as compiled marshaling in generated code. Interpretive marshaling is slower in part due to the use of certain accessor methods in the TypeCode and Any interfaces, which must return copies of their member data. The caller is then responsible for destroying the copy. In an application, this type of copying makes sense. However, copying is rarely warranted with interpretive marshaling since it is entirely internal to the ORB. Alternate methods that do not copy member data when accessed could be created and used internally within the ORB.

5.2 Footprint Reduction

The dynamic typing tools whose design and implementation have been described in this paper have all been created as self-contained libraries, making the additional compilation time and the additional footprint from linking completely optional. However, the most basic CORBA dynamic typing capabilities, such as TypeCodes, Anys, and the ORB's interpretive marshaling code, have not yet been separated in this way. Restructuring of ORB code to minimize dependencies plus the use of the Component Configurator [22] and Adapter [21] patterns, as described in Sections 2.7 would allow further subsetting of the ORB, along with reduced footprint and increased configurability.

5.3 Interface Repository Scalability

Scalability of the TAO Interface Repository could be increased in both directions if the repository container class were modified to allow a size to be passed to the constructor of the underlying hash tables. Currently a default size is used for all hash table construction. Another approach to increased scalability would be to use real-time hash functions and tables [33], which amortize their resizing.

5.4 Interface Repository Federation

Although not possible in the current implementation, it is conceivable for an ORB to have access to multiple interface repositories. Federation of repositories could have one of several motivations, each with its own benefits.

- Reduction of id clashes, since repository ids must be unique only within a single repository.
- Specialized contents for each repository.
- Different security restrictions for each repository.

5.5 Emerging Technologies

5.5.1 The CORBA Component Model

Several areas of research related to the CORBA Component Model (CCM) [34] have the potential for cross-fertilization with dynamic typing and dynamic CORBA. They are described in the paragraphs below.

Component Feature Discovery. Through the `Navigations`, `Receptacles`, and `Events` interfaces, clients may discover the facets, receptacles, and event sources/sinks supported by a component. CORBA's dynamic typing capabilities could facilitate a self-describing capability in a component. Alternatively, they could work with the CCM Component Implementation Definition Language (CIDL) compiler to generate a table of such information to be included in the component implementation.

Component Configuration. A component-aware Interface Repository could maintain a dynamic database of information to be used interactively by the CCM configuration interfaces `Configurator`, `StandardConfigurator`, and `HomeConfiguraion` during the configuration and deployment phases of distributed component application.

IDL Extensions. The *import* keyword mandated by CCM will require the IDL compiler to use dynamic typing for its support. This support could be realized by software that essentially does the reverse of TAO's Interface Repository loader, *i.e.*, generates IDL declarations from repository entries. An interface repository would also be an ideal place for applications to discover component port information.

5.5.2 Model-Integrated Computing

Model-Integrated Computing (MIC) extends the scope and use of models with model analyzers and model interpreters so they can be used in every phase of system development [35].

IFR and Configuration Manager. In a component-based approach to MIC, a component-aware Interface Repository could function as a component database for use by the MIC Configuration Manager [36] in component assembly. This collaboration would give a system dynamic re-configuration cabability based on current run-time information.

Component Compatiblity. A Configuration Manager or other MIC tool may wish to know if two components are compatible before connecting them. This decision might be made on the basis of the components' ancestry, *i.e.*, whether or not it inherits from a given IDL interface. This can be discovered by calling the `is_a()` method that is inherited by all children of `CORBA::Object`. However, with no compiled stub code to use, the MIC tool would have to compose a DII request, even though the name and signature of the operation is known,

6 Concluding Remarks

The widespread transition to component models as the paradigm of choice for distributed application development is increasing the interest in dynamically typed middleware, which is an integral part of component-based application development [37]. To support changing requirements and conditions late in application lifecycles, *i.e.*, during deployment and at run-time, DOC middleware must therefore continue to evolve to support these new requirements. This paper (1) presents the key design challenges faced when adding dynamic typing capabilities to CORBA middleware and (2) describes how these design challenges were resolved via the systematic application of patterns and object-oriented design techniques.

Despite the widespread acceptance of DOC middleware, such as CORBA, COM+, and J2EE, developers are still faced with a great deal of diversity and heterogeneity among service domains when trying to achieve widespread deployment and use of their applications. For this reason, there is a growing interest in dynamic applications that can be open-ended and adaptive. By examining an extensive number of examples, we have shown that dynamic typing enables applications to possess these qualities, and therefore can work effectively in domains where application using statically typed middleware alone cannot.

By extending previous work on dynamic CORBA in TAO, we have designed and implemented a complete set of dynamic typing capabilities for TAO. By judicious application of patterns, software reuse, refactoring, and library subsetting, we have shown that extensive functionality can be added to an ORB implementation without incurring excessive penalties in performance overhead or memory footprint for application that do not use the additional capability. All the dynamic CORBA capabilities described in this paper are available in the TAO ORB, which can be downloaded from `deuce.doc.wustl.edu/Download.html`.

7 Acknowledgements

The four years I have spent at the Center for Distributed Object Computing in the Department of Computer Science at Washington University have been immensely rewarding and enjoyable. I am indebted to a number of people who have shared with me in abundance their patience, generosity, good humor, wisdom, and experience, and I would like to acknowledge each one individually.

I would first like to express my thanks to Dr. Douglas C. Schmidt, who is responsible for my joining the DOC Group, for my job as a DOC Group staff researcher, and for my becoming a graduate student. It was because of his challenging encouragement and attention that I chose to pursue a career in Computer Science.

I would also like to thank the faculty and staff of the Department of Computer Science at Washington University for making my return to university life propitious and stimulating. I am especially grateful to the members of my master's project committee: Dr. Ron K. Cytron, Dr. Christopher D. Gill, and Mr. Fred Kuhns, from each of whom I have benefitted enormously due to my association with them in both their professional and academic capacities. I would also like to express my appreciation to Dr. Sally Goldman of the Department of Computer Science at Washington University, and Mr. Paul Feldker of the Department of Physics at St. Louis Community College, who rekindled the fascination with science and mathematics I first experienced as a child.

I am indebted to past and present members of the DOC Group, for their friendship, collaboration, and perpetual willingness to share time, knowledge, and experience. These members include, but are not limited to, Mr. Nanbor Wang, Mr. Balachandran Natarajan, Mr. Ossama Othman, Mr. Carlos O'Ryan, Dr. Irfan Pyarali, Dr. Aniruddha Gokhale, and Mr. Krishnakumar Balasubramanian.

Finally, I wish to express my eternal gratitude to my mother, Elaine Parsons, who has given me a lifetime of support, praise, and encouragement.

A Appendix

This section contains the details of how the Memento pattern was used to externalize the state of the meta-objects (IR Objects) that describe IDL declarations in the Interface Repository. The figures below show the underlying structures in the Interface Repository container for entries corresponding to the various IR Object types. As explained in Section 2, internal IDs of nodes in the tree of hash tables can be of four types - string, unsigned long, binary chunk (of specified length) or the root of another subtree.

Figure 15 shows how each of these node types is represented

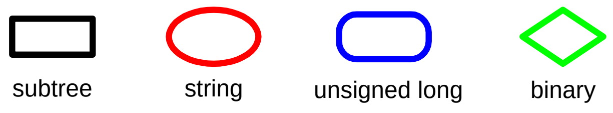


Figure 15: Node Type Legend

in subsequent figures. The labels shown on each node are the string names that constitute the external id in the hash table. Many of these string names are converted from the hexadecimal representation of unsigned long integers. This was done to preserve the order of declaration when iterating over the subsections of a "defns" section, since the available hash table iterators do not guarantee to reproduce this order. Nodes labeled "<xxxxxxxx>" represent some unsigned long value in hex converted to a string. Unsigned long values labeled "count" contain the number of subsection entries in the table.

IR Objects corresponding to typed IDL declarations inherit from one or both of the abstract IR Object classes `Container` and `Contained`. Figures 16 and 17 show the node structure common to all IR Object types inheriting from

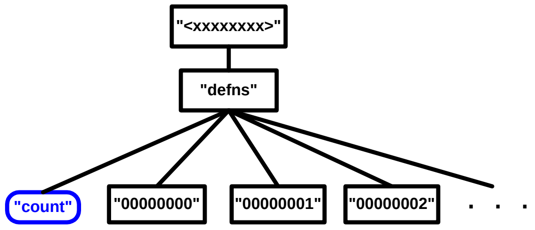


Figure 16: Container

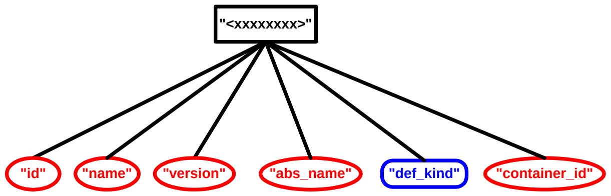


Figure 17: Contained

these base classes. Nodes for specific IR Object types shown in subsequent figures will exist in addition to those for any base classes that apply from these two figures. For a complete description of the inheritance structure of IR Objects, see the CORBA specification [20].

Figure 18 shows the root of the tree and its children. Subsequent figures show expansions of these child nodes.

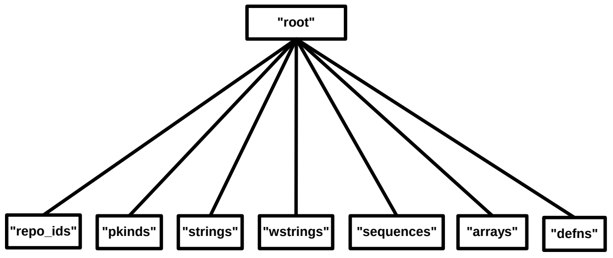


Figure 18: Repository Root

The subtree shown in Figure 19 is an index section where repository ids are mapped to strings which are backslash-separated segments representing the path from the root to the

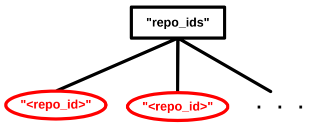


Figure 19: Repository ID Index Section

entry corresponding to the repository id. The path string can be passed to a method in the Interface Repository container class that returns the external id of the item, if it is in the repository.

Figure 20 shows the expansion of the section that contains

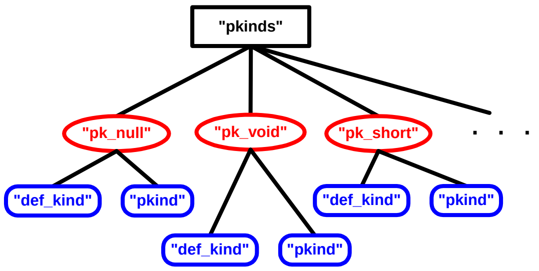


Figure 20: PrimitiveDef Section

IR Objects of type PrimitiveDef, which correspond to the basic IDL data types.

Figures 21 and 22 show the subtrees that contain entries for

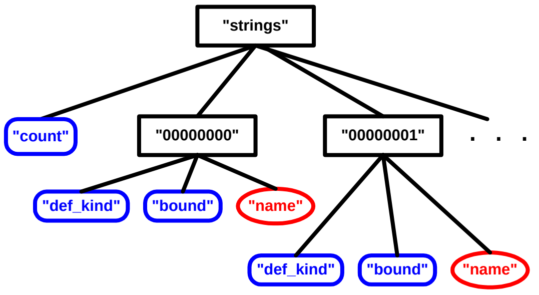


Figure 21: StringDef Section

IDL string and wide strings, respectively. The structure of both subtrees is identical, since IDL strings and wide strings are not named types, so only the bound is needed to distinguish one string or wstring from another.

Figures 23 and 24 show the subtrees that contain IDL declarations of sequences and arrays, respectively. Like IDL strings and wide strings, IDL sequences and arrays are not named types, so their declarations are collected in these top-level sections. The rightmost child of the root node in Figure 18 corresponds to the "defns" node seen in Figure 16, since the repository is itself an IR Object that inherits from Container.

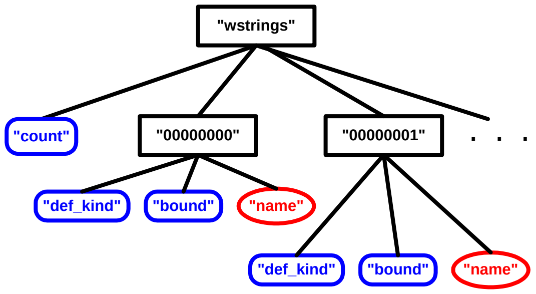


Figure 22: WstringDef Section

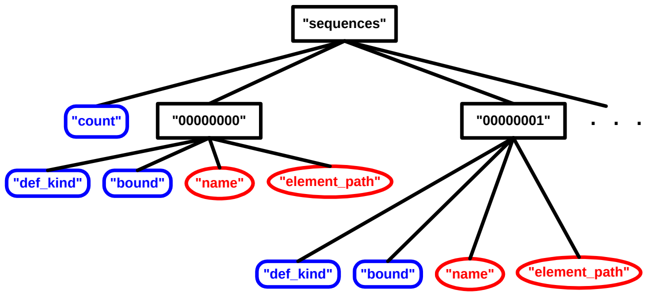


Figure 23: SequenceDef Section

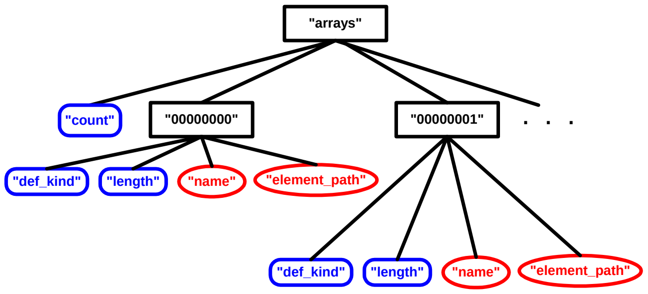


Figure 24: ArrayDef Section

The remaining figures in this section show the entry structures for named IDL types. Figure 25 shows the structure

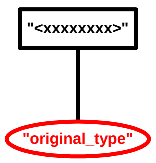


Figure 25: AliasDef

for AliasDef, the IR Object type that corresponds to an IDL typedef declaration. We see that this structure is quite simple, requiring only a path to the original type, located elsewhere in the repository. Figure 26 shows the node structure corresponding to an IDL interface's attribute declaration. The

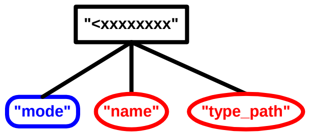


Figure 26: AttributeDef

"mode" value tells us whether or not the attribute has been declared as read-only, the "type_path" string holds the path to the repository entry for the attribute's type, and the "name" string contains the attribute's local name in the enclosing interface declaration.

Figure 27 shows the node structure for ConstantDef. The size of the stored binary data is determined by the constant's

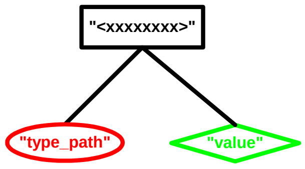


Figure 27: ConstantDef

type, which must be looked up at retrieval time in order to extract the constant's value. The path value in "type_path" will point either to an entry in one of the top-level sections "pkinds", "strings" or "wstrings", or to an EnumDef entry, whose structure is shown in Figure 28. In an EnumDef entry, a "count" value and hex-to-string subsections are used to preserve the order of the enum values when a repository query iterates over them to create a list.

In Figures 29 and 30, we see that the entry structures are identical, the only difference in the two being in IDL syntax, where an exception may contain no members while a struct may not. In the repository, both StructDefs and ExceptionDefs

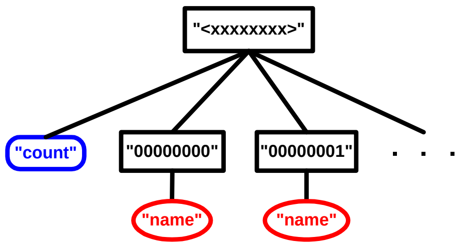


Figure 28: EnumDef

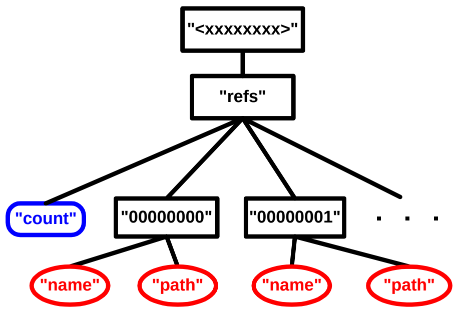


Figure 29: ExceptionDef

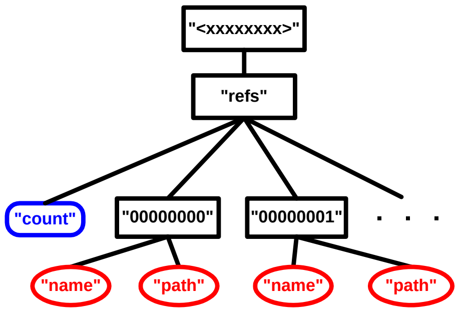


Figure 30: StructDef

have a ”refs” section, which in turn contains a ”count” value and hex-to-string subsections, again to preserve the member order in the event of iteration by a repository query. An ExceptionDef in the repository representing an empty IDL exception would have a ”count” value of 0 under the ”refs” section, and nothing else.

Figure 31 shows how the entry structure for UnionDef is

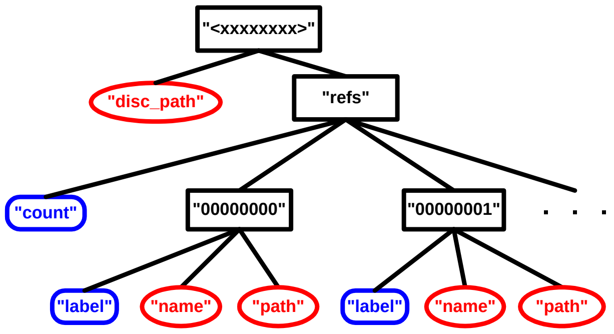


Figure 31: UnionDef

similar to that for ExceptionDef and StructDef, with the addition of a path to the entry for the discriminator type, and a label value for each member. If a member happens also to be declared inside the enclosing union, struct or exception, the ”path” string for the member will point to an entry under the ”defns” section of the member's enclosing scope, since UnionDef, StructDef and ExceptionDef all inherit from Container. As explained above, nodes common to all IR Object inheriting from Container and/or Contained are not shown in every figure to keep them as uncluttered as possible. The node structure for InterfaceDef, shown in Figure 32, has a section contain-

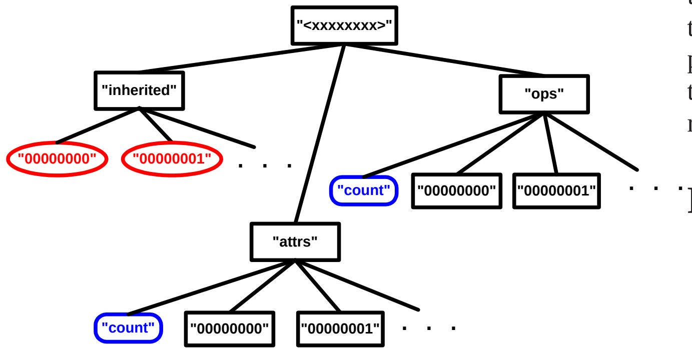


Figure 32: InterfaceDef

ing a list of parent interfaces, and a section each for attributes and operations, each of which has a ”count” value and hex-to-string subsections to preserve the order of declaration.

Figure 33 shows that a repository entry for an IDL native declaration contains no additional information, which is to

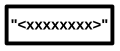


Figure 33: NativeDef

be expected since the corresponding IDL declaration contains none. On the other hand, the node structure for an OperationDef, shown in Figure 34 is the most complex. There is a ”mode” value, which tells us if the IDL operation is declared

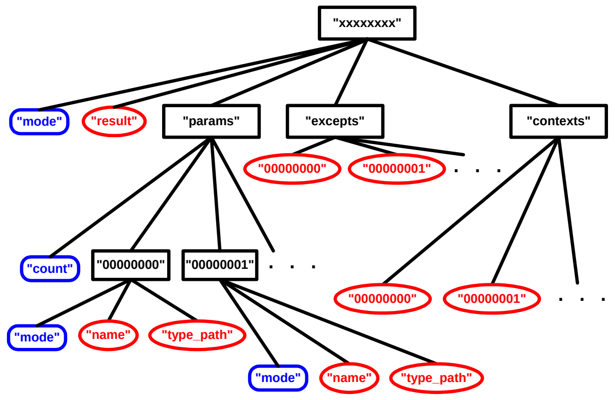


Figure 34: OperationDef

as oneway, and a ”result” value which contains the path to the repository entry corresponding to the operation's return type. The ”contexts” section contains the string names of the operation's contexts, if any, and the ”exceptions” section contains the paths to ExceptionDef entries in the repository for any exceptions the operation may raise. Finally, the ”params” section has a ”count” value and hex-to-string subsections to preserve the order of parameter declaration. Each parameter subsection is in turn composed of a ”mode” value, which labels the parameter as `in`. `inout`, or `out`, a string value containing the parameter's name, and a string containing the path to the repository entry corresponding to the parameter's type.

References

[1] R. E. Schantz and D. C. Schmidt, “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications,” in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2002.

[2] D. C. Schmidt and S. Vinoski, “Dynamic CORBA, Part 1: The Dynamic Invocation Interface,” *C/C++ Users Journal*, July 2002.

[3] Object Management Group, *Python 1.2 RTF Report*, OMG Document ptc/02-06-05 ed., June 2002.

[4] Object Management Group, *IDLscript RTF Report*, OMG Document ptc/01-08-29 ed., June 2001.

[5] Jim Coplien, *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

[6] Stan Lippman, *C++ Primer, 2^{nd} Edition*. Addison-Wesley, 1991.

[7] Kenneth C. Louden, *Programming Languages – Principles and Practice*. PWS, 1993.

[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] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, Oct. 1997.

[10] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[11] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[12] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[13] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[14] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[15] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA," *Concurrency and Computing: Practice and Experience*, vol. 13, no. 2, pp. 507–541, 2001.

[16] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.

[17] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware," *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, Oct. 2001.

[18] 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.

[19] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, Nov. 1996.

[20] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, Dec. 2001.

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

[22] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[23] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.

[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring - Improving the Design of Existing Code*. Reading, Massachusetts: Addison-Wesley, 1999.

[25] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," in 16^{th} *ACM Symposium on Principles of Programming Languages*, pp. 213–277, ACM Press, 1989.

[26] P. A. MuckelBauer and V. F. Russo, "The Renaissance Distributed Object System," Department of Computer Science, Technical Report TR.93-022, Purdue University, 1993.

[27] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1998.

[28] D. Rogerson, *Inside COM*. Redmond, WA: Microsoft Press, 1997.

[29] W. Rosenberry, D. Kenney, and G. Fischer, *Understanding DCE*. O'Reilly and Associates, Inc., 1992.

[30] P. Merle, C. Gransart, J. Roos, and J. Geib, "CorbaScript: A Dedicated CORBA Scripting Language," in *CHEP'98 Computing in High Energy Physics*, (Chicago, IL), 1998.

[31] A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in *Proceedings of INFOCOM '99*, Mar. 1999.

[32] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, Apr. 1990.

[33] S. Friedman, N. Leidenfrost, B. C. Brodie, and R. K. Cytron, "Hashtables for embedded and real-time systems," in *Proceedings of the IEEE Workshop on Real-Time Embedded Systems*, 2001.

[34] Object Management Group, *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03 ed., Nov. 2001.

[35] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.

[36] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad, "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems," Tech. Rep. ISIS-99-01, Vanderbilt University, 2000.

[37] G. T. Heineman and B. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001.