Aldor Generics in Software Component Architectures

(Spine title: Aldor Generics in Software Component Architectures)
(Thesis Format: Monograph)

by

Michael Llewellyn Lloyd

Graduate Program
in
Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Masters

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
October, 2007

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Supervisor	Examiners
Dr. Stephen M. Watt	Dr. David Jeffrey
Supervisory Committee	Dr. Hanan Lutfiyya
	Dr. Eric Schost

The thesis by
Michael Llewellyn Lloyd
entitled

ALDOR GENERICS
IN SOFTWARE COMPONENT ARCHITECTURES

is accepted in partial fulfillment of the
requirements for the degree of
Masters

Date	Chair of the Thesis Examination Board

Abstract

With the introduction of the Generic Interface Definition Language(GIDL), Software Component Architectures may now use Parameterized types between languages at a high level. Using GIDL, generic programming to be used between programming modules written in different programming languages. By exploiting this mechanism, GIDL creates a flexible programming environment where, each module may be coded in a language appropriate to its function. Currently the GIDL architecture has been implemented for C++ and Java, both object-oriented languages which support compile time binding for generic types. This thesis examines an implementation of GIDL for the Aldor language, an imperative language with some aspects of functional programming, particularly suited for representing algebraic types. In particular, we experiment with Aldors implementation of *runtime* binding for generic types, and show how GIDL allows programmers to use a rich set of generic types across language barriers.

Keywords: Aldor, Generic Interface Definition Language, GIDL, Corba.

Acknowledgments

I would like to thank my supervisor, Dr. Stephen M. Watt, for his continued support and guidance.

I also wish to thank Cosmin Oancea, for his guidance and insights through the world of GIDL, and the many valuable discussions exploring the benefits of Aldor generics.

Lastly, I wish to thank my colleagues at the University of Western Ontario ORCCA lab for making the experience both challenging and fun, and for the many contributions made to this work.

Glossary

CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
GIDL	Generic Interface Definition Language
IDL	Interface Definition Language
MSIL	Microsoft Intermediate Language
OMG	Object Management Group
SCA	Software Component Architecture

Table of Contents

List of Figures

Chapter 1

Introduction

The field of computer science boasts vast array of programming languages, most geared toward a specific set or subset of problems. However, large projects often consist of several smaller problems and components. To better find solutions to such diverse problems, it is useful to use a variety of programming languages even within a single project, each suited to its own task.

Software component architectures (SCAs) allow developers to choose an appropriate language for each modular task and create interfaces between modules written in different languages. Independently developed modules and libraries may be integrated, regardless of programming language. Each aspect of a software project may be solved in a programming language to which it is well suited, then combined with other solutions to form a cohesive whole.

One of the motivations for our research is to demonstrate how languages with distinct binding times for generic types may be integrated into a software component architecture.

The term "Parametric Polymorphism" refers to the idea by which definitions for programming language types and functions may be used with different types of data, with the types given as parameters. Since its one of its early appearances in the functional language *ML* [11], parametric polymorphism, sometimes called *generic programming* or programming with *templates*, has become a useful feature in both object oriented languages such as C++ and Java, and statically typed functional programming languages.

We contend that the ability to program generically would be desirable when developing in a multi-language environment. In any case, with the resounding popularity of parametric polymorphism in object-oriented languages, it is an interesting topic to explore. One could argue that when working in multiple languages any feature that can be reasonably and easily used between languages becomes extremely useful. Though only recently introduced to Java, parameterized types have been in common use in imperative and functional languages such as ML, Aldor, C++ and many others.

The type parameters supplied to a parameterized type can have various binding times based on the implementation of the language. Some languages implement *compile time binding* for generic types, where types for each instance of the generic type are created during compilation. Some functional languages use *run-time binding* for generic types, where new types are created at run time based on the supplied parameters. One of the drawbacks with the C++ implementation of compile time generics is that generic types are not checked for type safety; the compiler can give no guarantee that the generic type will compile with all intended parameters. Figure 1.1 demonstrates how type-unsafe generic types may be created using compile time generic languages. The parameterized *List* type shown in figure 1.1, will compile provided that the type substituted for T is a class that provides a *print()* function.

Figure 1.1 Example of a Type-Unsafe Parameterized type in C++

```
template <class T>
class List {
    private:
       T car;
       List<T> cdr;
    public:
       List(T c){
            car = c;
       };
       List(T c, List<T> cd){
            car = c;
            cdr = cd;
       };
       print(){
            car.print();
            cdr.print();
       };
};
```

However, if included in a library, the class created is dependent on its usage within a program, and is may fail cryptically when used with incorrect arguments. For this reason it may be advantageous for a programmer to develop parameterized types in a language with support for *type checked generics*, which can verify validity for a declared subset of types during the first compilation.

With several languages supporting some form of parametric polymorphism, there exists an opportunity to extend software component architectures to use polymorphic types. The Generic Interface Definition Language (GIDL) project[8] at the University of Western Ontario is such an architecture. GIDL is a general Software Component Architecture (SCA) designed to provide generics. In order to provide generics for the wide variety of programming languages, GIDL defines a variety of semantic forms for parametric polymorphism in order to facilitate languages outside of current main-

stream programming.

The Aldor programming language provides an opportunity to extend GIDL to include a language with run-time generics. Aldor was originally developed by IBM as an extension language for AXIOM, a system for computer algebra. Aldor treats functions and types as first class values, allowing both to be passed as arguments or created at runtime. This uniform treatment of functions and types creates a powerful programming environment and provides an interesting challenge to map to an object oriented architecture.

This thesis will demonstrate how GIDL allows a language using runtime binding for generic types to be integrated with languages using compile time binding. In addition we will examine possibilities for related research, such as automated library translation to GIDL, and using Aldor's foreign function interface as an intermediate to expose additional languages to GIDL.

Chapter 2 will cover some of the background projects and software that are related to this thesis. We provide some details on the CORBA software component architecture, covering its basic approach and uses. We detail the GIDL project in section 2.2 and examine its motivation and goals in bringing parameterized types to software component architectures. Section 2.3 will cover the Aldor language and its particular interest to us in providing an interesting language to experiment with different types of generics.

Chapter 3 covers our initial approaches to the implementing the GIDL architecture for Aldor. We briefly review our initial experiments in providing a suitable type mapping between GIDL and Aldor. We also touch on one of the possible extensions to this work. Chapter 3.2 overviews our first major approach to providing a fully func-

tional Aldor Object Request Broker (ORB) for use with GIDL. This section details our single-compiler approach in which the GIDL IDL compiler is used to generate ORB code directly for use in the Aldor language. Finally section 3.2 examines our final design and our reasoning for adopting a two-compiler model.

Chapter 4 describes the implementation of the two-compiler method. We examine how all GIDL basic types are mapped in to the Aldor language, detailing some of the problems encountered when mapping constructed types and providing GIDL equivalent type scope. In this chapter we also detail our mapping for GIDL interfaces to Aldor category/domain pairs, the three types of GIDL generics and how we have simulated GIDL inheritance functionality. Sections 4.2.9 and 4.2.10 examine the generated stub and skeleton code. Later sections of this chapter examine some of the interesting details and problems that arose in mapping GIDL to Aldor and some techniques used to marry these two languages with very different type models.

Chapter 5 shows some test programs using our GIDL/Aldor mapping. These test cases are included to give an idea of what a GIDL/Aldor program looks like and demonstrate some of its capabilities.

Chapter 6 provides our final overview of the project, conclusions and examines some of the further possibilities for GIDL/Aldor. We also discuss some of the benefits of a SCA with support for generics.

Chapter 2

Background and Related Work

2.1 OMG CORBA

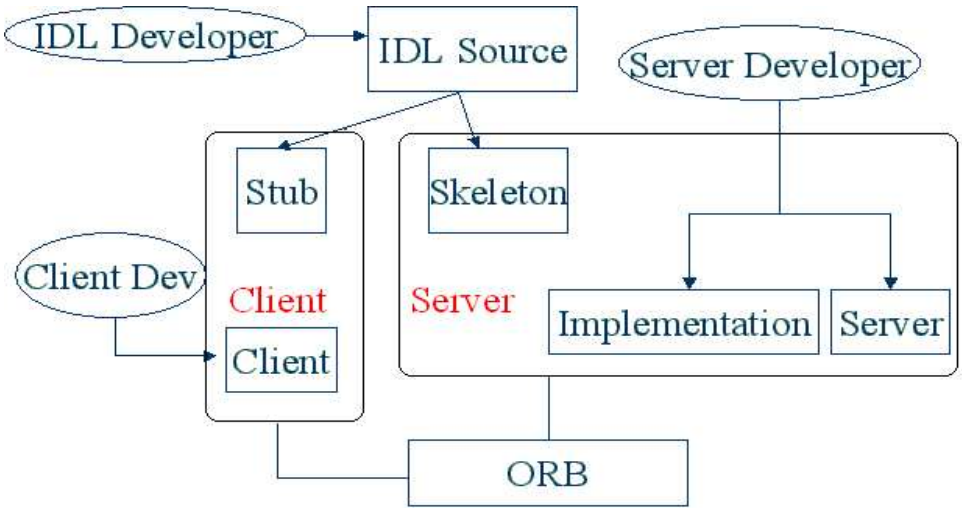
Originally developed (v1.0) in 1989 and extended in 1994 (v2.0)[1], the Common Object Request Broker Architecture (CORBA) is a well established SCA with support for more than thirty programming languages. However, the CORBA object-oriented model does not support the use of parametric polymorphism.

CORBA was designed as a language-independent software component architecture for creating distributed systems. Using an Interface Definition Language (IDL) to define abstract classes to be 'shared' between components, CORBA acts as a compatibility layer between languages and systems. This approach allows CORBA objects to be used by any platform and any system which implements a CORBA Object Request Broker (ORB).

CORBA applications are modeled on an object oriented, tiered, distributed system, usually with one component acting as a *client* which makes requests to one or

more *servers*. The *server* component may additionally act as a client to other components. For the sake of simplicity, we will only consider in detail a single client/single server model, relying on compositions of this relation to build more general architectures.

Figure 2.1 Overview of CORBA Framework



The key to understanding CORBA is in understanding how IDL can be used to communicate across programming languages. A CORBA application is constructed of five main conceptual components; the Client, Stub, ORB, Skeleton, and Server. The ORB provides the core functionality for sending and handling requests between the client and server. The Stub is generated by the IDL compiler and may differ between CORBA implementations. The role of the stub is to provide a transparent interface between the user's program and any remote or CORBA controlled objects being used. The exported functions defined in IDL become exports of the generated domain, with similar type signatures. When called from a client program, the stub functions *marshal* parameters given for transmission to a server.

Similar to the Stub, the Skeleton file is generated from the IDL file for each application. The role of the Skeleton is to receive function call requests from the Stub and call the corresponding function implemented by the user.

The Client and Server files are written solely by the developer and constitute the program being developed. Typically the Client embodies the logic of the program with one or more of the implementation modules being implemented on the Server.

A CORBA mapping for a programming language consists of a library of *CORBA types*, programming types representing the basic types and programming objects defined by the CORBA standard, a definition for how IDL interfaces and definitions may be represented in the target language, and (optionally) a compiler which follows the definitions and generates Stub and Skeleton code in the target language. The OMG CORBA standard itself defines a set of basic types, programming objects, and the inter-operable object reference (IOR) and GIOP data representations. CORBA defines value ranges and sizes for basic types such as 32, 64 bit integer, float, double, string, character, octet, etc, and variable length types such as array, and sequence. Standard accessor functions are defined for the *Portable Object Adaptor*(POA) which is used by the ORB to manage *servants* (user written objects) to be accessed remotely. In addition, the CORBA standard contains specification for *naming services*, CORBA objects which may be used to locate and return other CORBA objects.

It is not the intention of this thesis to create a complete CORBA language mapping for Aldor. Instead, we hope to demonstrate how the GIDL SCA may be mapped to a language with support for run-time generic types. With this goal in mind, only the basic CORBA functionality to create inter-language generic programming has been included in this project. Many OMG CORBA features, such as naming services and

extensive memory management systems, are left to the CORBA developers of the underlying CORBA ORB GIDL-ALDOR interacts with.

2.2 GIDL

With parametric polymorphism slowly being introduce into mainstream programming languages such as C++ and Java, the possibility for an SCA with support for generic languages presented itself. The FRISCO project experimented with using generics between Aldor and C++. The motivation for FRISCO was Aldor, providing a rich data type structure for mathematical programming with support for generic types, and the C++ PoSSo library, a fast C++ library for computer algebra making heavy use of templates. The FRISCO project provided many insights into mapping object-oriented data types to Aldor, and provided evidence that the use of generics in inter-language programming would be a valuable feature.

Motivated by the FRISCO project, the Generic Interface Definition Language was developed to provided a solid SCA with support for generic types. Generic Interface Definition Language (GIDL)[8] is an extension to OMG IDL. GIDL allows for the definition of parameterized interfaces which may then be translated into equivalent data-types in any implemented language. Currently, mappings for GIDL exist for C++ and Java, allowing polymorphic types to be used between these languages[8]. A GIDL encapsulation of the C++ Standard Template Library has been developed by Cosmin Oancea (University of Western Ontario, ORCCA) as validation of GIDLs capabilities.

GIDL is an implementation-independent framework to provide generics over an existing Software Component Architecture. In order to provide for the varying models

for generics in different languages, GIDL defines three types of generics;

non-restricted types are types accept any type as a parameter.

extend-restricted types are given a *restricting type* in their definition. Only types belonging to the restricting types inheritance tree may be bound. Inheritance follows standard object oriented sub-typing rules.

implement-restricted type are more permissive than extend-restricted types, in that types that fully provide the exports of the restricting type may be bound. Bound types need not belong to the inheritance tree of the restricting type. Only exact matching of each export signature is accepted.

GIDL provides an extension for CORBA-IDL to allow definition of generic *interfaces*. At this time, generic functions are not included as CORBA-IDL has no support for static methods, and some generic languages, such as ML, do not support it. GIDL generic interfaces are based loosely on those found in C++ or Java and follow similar scoping and inheritance rules. Parameterized types may inherit or be inherited, however a type $List<B>$ does not extend $List<A>$ even if B extends A. In addition to allowing three different kinds of generic type parameters, GIDL interfaces may also include *non-type class template* parameters. These parameters allow a generic type to be supplied a constant value, such as an array size, on instantiation.

GIDLs model for generics was based on four primary rationale [8]:

- The type of any expression should be context independent.
- The model for generics should allow the restrictions placed on generic types to provide easy extensibility and readability.

- Mappings should allow programming languages with support for parameterized types to make use of them in a natural way.
- GIDLs model should allow for backward compatibility with CORBA applications that do not have support for generic types. In addition, client applications written in languages with no support for generic should still be able to inter-operate with server applications that do.

Apart from allowing generic types, the GIDL compiler makes no modification to the OMG IDL language, allowing standard OMG IDL to be used with GIDL. GIDL parameterized types remain backwards compatible with OMG-CORBA IDL and inter-operable.

In order to accomplish backward compatibility with OMG-CORBA mappings, GIDL utilizes a *type-erasure* technique similar to that found in Java 5. Function parameters dependent on the generic type are converted to a general IDL type prior to transmission. The transmitted type loses some of its type information during transfer. The transmitted argument then has its type rebuilt in an implementation-dependent manner when received. Using this technique GIDL provides polymorphic exports without modification of the underlying SCAs transmission protocol.

GIDL-generated types act as wrappers for CORBA-IDL generated types. When a GIDL method is invoked on the client, the generated GIDL method *erases* the parameterized type information and calls the corresponding generated IDL method on the downcast parameters.

Type-erasure for non-restricted types is accomplished by replacing all instances of the scope name with the *any* type. This ensures that any possible type will be accepted and allows for the type to be *re-cast* by the GIDL server after transmission.

Extend-restricted types are recast to the highest level type accepted by the generic type, allowing OMG-IDL to enforce strong type restrictions on the erased type without restricting functionality.

In order to provide type-erasure for *implement-restricted*, GIDL reduces these types to *extend-restricted* types using a *most general generic unifier (MGGU)*. The *MGGU* is a constructed interface making use of only *non-restricted* generic types, and provides the full list of exports described in the given *to be implemented* type. All GIDL types that fully implement the exports declared in the *MGGU* (and therefore the base restricting type) are made to extend the *MGGU*. This reduces *implement-restricted* generics to a form of the *extend restricted type*, which is then type-erased as described.

2.3 Aldor

The Aldor programming language is a strongly-typed imperative programming language that uses functions and types as first class run-time values. Originally developed by IBM as an extension to Axiom[5], Aldor has strong support for symbolic and numeric computation and *type safe* generic programming.

Current trends in computer languages increasingly favor methods of dynamic programming. With the introduction of templates to C++ and Java, languages must take care to continue to provide safety and program variability. In this trade off between dynamic flexibility and compile time verification the Aldor language sits comfortably by allowing strong typing as well as the use of dependent types and the treatment of functions as first class values. Dependant types provide the functionality of templates of other languages.

Aldor creates domains (types) dynamically at runtime, providing a flexible environment for generic types well suited for the runtime generics required in GIDL. As such, Aldor is a good candidate for experimentation with generic types in software component architectures. The Aldor *foreign function interface* allows the execution of C functions from within Aldor, which permits the use of an existing CORBA C implementation to be used instead of implementing the OMG CORBA standard strictly in Aldor.

Objects in Aldor may be represented by a category/domain pair [2]. Categories in Aldor are similar to *interfaces* in Java or *abstract classes* in C++. The purpose of the category is to export public function definitions for the domain as well as *inheritance* information.

Domains provide the implementation of functions. Domains may implement a default category to automatically export all functions within the domain. This simply provides a way to organize and separate groups of functions, categories and domains, similar to a C++ namespace. A domain may also provide an implementation of a specific category, in which case it implements functions defined as exports of the category. This provides a two-tier type structure. In addition, domains may provide a representation type, which determines how a value belonging to the domain is represented in memory as well as which functions are internally available to the domain when instantiated.

Chapter 3

Problem and Approaches

3.1 Overview of ORBit

This section provides a brief overview of the C language CORBA ORB *ORBit* and highlights details of its implementation that are important to this thesis.

According to its documentation, ORBit is a CORBA 2.4-compliant Object Request Broker implemented in C and distributed freely under the Lesser Gnu Public License (LGPL). ORBit is included in most Linux distributions and is featured as a component of the GNU Object Model Environment (GNOME). ORBit was chosen as our underlying ORB implementation when experimenting with GIDL due to its wide availability, open source code, and C language implementation that can be is easily accessible through Aldor's foreign function interface.

ORBit client structures are fairly straightforward and their implementation does not significantly affect the Aldor mapping. Interface definitions are mapped to a named structure that holds the IOR used to identify the object on a remote server,

and a set of accessor functions in accordance to OMG CORBA C language naming standards. The implementation of these accessor functions varies between ORBit versions, however. In our implementation, these functions are either replaced (as in the Single-Level approach) or called directly (Two-Level approach).

3.2 Initial Experiments

In order to show that a GIDL mapping for Aldor could be successfully built using an existing C CORBA implementation we conducted a series of experiments using a direct straightforward approach. The goal was to develop a working client/server side CORBA compliant mapping without support for generics, in order to determine whether an existing implementation could be used directly and experiment with various methods for mapping Aldor types to GIDL. We created thin wrapper domains for all conceptual types used in the underlying ORBit library. These types contained little or no information about their representation, save for a pointer to the data, which could be stored and passed to the underlying ORBit function calls as required. Using this mechanism, client Aldor client programs could call Corba exported functions through an existing C language ORB using the C Foreign function interface and thinly wrapped types. This experiment proved that there was no need to implement a fully functional CORBA ORB in Aldor.

Aldor provides the ability to extend existing types to belong to new categories and provide additional functionality. Using this *post-facto extension* feature, we sought to extend existing Aldor library types to include the required functionality to be used with GIDL. This approach provided many hypothetical advantages over re-implementing the existing types, chiefly the extended types would be able to be

transparently used between pure `libAldor` functions and remote GIDL functions without the need to re-cast to an appropriate GIDL type. Unfortunately difficulties in successfully implementing a large number of extended types prevented this feature from being fully explored within our time frame and the simpler, more direct approach of re-implementation was adapted.

3.3 The Single-Level Approach

3.3.1 Overview

Our initial approach taken when building a GIDL Aldor SCA was to build directly upon an existing C implementation for CORBA. This approach allowed us to work around the significant overhead that exists when using the high level interface provided by most ORB implementations. Allowing Aldor to directly call and use the underlying library code, we are able to avoid unnecessary or redundant type manipulation.

CORBA GIOP (Generic Inter Object Protocol) is a specification language used to describe how data is structured when communicating between CORBA modules. In order to allow GIDL Aldor programs to compile and run without an additional third party IDL compiler (see section 3.4), generated Aldor stub and skeleton functions included a significant amount of code used to handle GIOP communication. In addition, every basic GIDL type mapped to Aldor required several additional functions to allow for this communication. *Marshaling* functions were required to translate each type to a GIOP compliant format and provide correct byte alignment within the message buffer supplied by the underlying ORBit library. *Demarshaling* functions

for each type read from the receive buffer and reconstruct the type. GIOP does not specify a byte order for its communication, but instead relies on the receiving module to perform any byte re-ordering that may be required for the architecture. Each *demarshaling* function was required to potentially re-order received data.

Each type requires a *Type Code* structure used to describe the type for a number of internal functions and when converting to and from the *Any* type. Type Code structures are specified by the OMG CORBA standard, and implemented in the Aldor mapping by a *getTypeCode* function within each domain which returned the specific structure for each basic type.

Figure 3.1 GIDL definition for List

```
interface Comparator<S>{

   boolean compare(in S x, in S y);

};

interface List<S; C:) Comparator<S> >{

    List<S; C> sort( in C c);
    List<S; C> cons( in S x);

};
```

3.3.2 Details of the Single-Level Approach

The goal of the single-level approach was to take advantage of the open source nature of the ORBit CORBA object request broker. By calling ORBit functions directly from the Aldor code generated by GIDL, we could eliminate the need to use the ORBit

compiler completely. This created a simplified, one step compilation to produce an Aldor-GIDL program and allowed for some simplification of the mapping, since marshaling could be done directly from Aldor and is therefore not restricted by the C language mapping.

Figure 3.2 Stub *cons* Method Generated from List.idl

```
cons( __this:%, __a0GIDL:S,
      __env:CorbaEnvironment):(List__ref(S, C))==
{
      __cnx:GIOPConnection == getConnection(__this);
      (__sndBuff:SendBuffer, __requestId:CorbaLongLong) ==
                                          use(__cnx,"cons",%,__this);
      if nil?(__sndBuff) then {
          setSystemException(__env, ex__CORBA__COMM__FAILURE,
                             CORBA__COMPLETED__NO);
          throw CorbaSystemException;
      }

      marshal(__sndBuff, __a0GIDL);
      write(__sndBuff);
      unuse(__sndBuff);
      __recBuff:ReceiveBuffer == use(__cnx, __requestId, true);
      if nil?(__recBuff) then {
             setSystemException(__env, ex__CORBA__COMM__FAILURE,
                                CORBA__COMPLETED__MAYBE);
             throw CorbaSystemException;
      }
      __ret:List__ref(S, C) == demarshal(__recBuff);
      unuse(__recBuff);
      return (__ret);
};
```

Figure 3.2 shows the client stub code generated for the *cons* function from *List.idl* (Figure 3.1). The first three lines retrieve the connection information associated with this object using *getConnection*, an Aldor wrapper for an imported ORBit C function. The *use* function creates a new *SendBuffer* type to use for marshaling parameters

and creates the GIOP header for the request. In addition to the function name, the GIOP request header will also contain information such as the endian orientation of the machine, GIOP version, and several other parameters used on the server side, most of which will be parsed out when the request is received by the server and is not relevant to GIDL.

Following a check for a *nil* or invalid *SendBuffer*, the functions arguments are converted to GIOP format and copied into the buffer using the *marshal* method for each type in the mapping. In the case of unrestricted template types, the type is first *coerce*d to the *Any* type before marshaling.

Lastly, the *ReceiveBuffer* is created to hold the reply from the server, and the return types are *demarshaled* and returned by the function.

The *portable object adaptor* holds pointers to active *POA servants*. Each *POA servant* is responsible for dispatching the requested function on the user's object and marshaling the result back to the client. The details of the POA and POA manager are handled through ORBit, however since the POA servant type for each interface defined in IDL is generated by the ORBit IDL compiler, we generate a similar structure from the GIDL compiler. The ORBit POA servant type is represented in C as three nested structures which we will designate POA, VEPV, and EPV. The POA for each object is used by ORBit to handle requests for that object's functions and provide a reference to the user's object. The POA object is represented by a category/domain pair. The POA category for the object should extend a basic *CorbaSkel* category which provides exports for functions common to all POA objects, such as *getSkel*. The generated POA category must also provide exports for the object's method dispatch function(s). The POA domain for the object must correctly implement all functions

defined in the POA category and must also provide the representative structure for the POA object.

The top level POA structure serves as a container for the VEPV structure as well as the *classid* structure, containing the interface name, and two pointers to functions, one required to initialize a local reference to the object and the other to the C-viewable *getskel* function used to retrieve the methods of the object by name. This POA *type* exports all the functions that are required to connect the underlying ORBit libraries with the user's code. This includes the *getskel* function, which is used internally to return required function pointers to the ORBit libraries, the *init* function, which is used by the user to create an instance of the POA type, and a demarshaling function for each function defined for this interface in the IDL, used to retrieve parameter values for the function call from the client.

The VEPV structure contains a pointer to the *base_epv* structure, a structure used internally by ORBit, and the EPV structure.

The EPV structure is the most important of these structures for this paper. The EPV in ORBit implementation contains a *private* pointer, which may be used by the programmer to store private variables for the type, followed by a list of pointers to the user's implementation of each method of the object. In order to handle generic types, this structure has been modified to contain an EPV2 structure for each of the functions defined in the IDL. An EPV2 structure holds a pointer to the Aldor closure representing the user's implementation of the function, a pointer to the specific Aldor *demarshaling* function for the function, and a pointer to the user's domain instantiation for the type. This allows for the Aldor language to call the correct instance of the demarshaling function when using generic types. It also allows Aldor

maintain type information after the *demarshaling* function is returned from the *getskel* method, which must return a Pointer type.

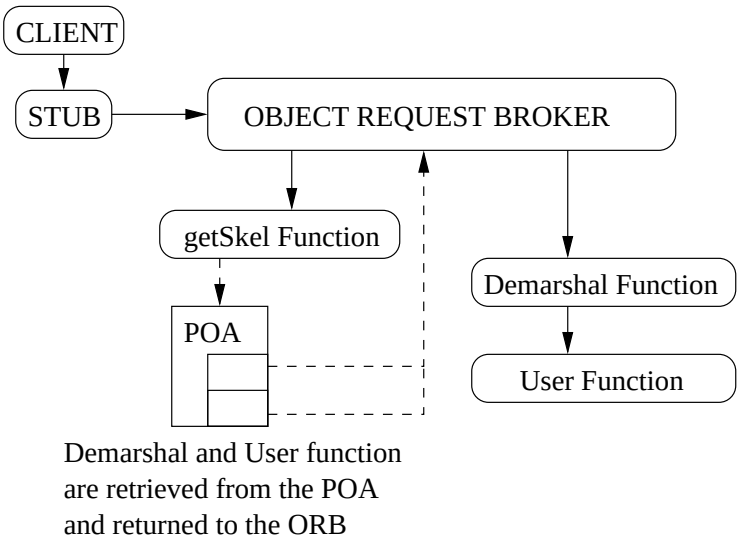
For each interface defined in the GIDL, a category/domain pair is generated for each of the above POA types. The category for each type exports the generic tie-ins *apply*, *set!* and *bracket* both for internal use and for advanced user's who may wish to modify the structures themselves. In addition, the top-level POA type exports the *init*, *getskel* and demarshaling functions, making them available outside the domain. In order to understand the functionality of these structures, one must understand the basic flow of how a CORBA request in handled.

When the client stub makes a method execution request, it includes a reference for the desired object converted to a string (*"stringified"*), the method's name, and the parameters on which the method is to be executed. The Object Request Broker uses the stringified reference to retrieve the POA structure corresponding to the requested object from the interface repository [1], and executes the *getskel* function (contained within the POA structure).

The *getskel* function compares the method name passed by the client stub to its list of methods. If the method name is not found, an exception is returned to the client stub. If the method is found, the function returns a pair of pointers. The first references the *demarshaling* function used for the requested method, which is later called to rebuild arguments from their GIOP representation. The second pointer references a structure containing the user's instantiation of the object and the EVP2 structure for the requested function, both of which are retrieved from within the EPV structure.

After calling the *getskel* function, the ORBit server then calls the *demarshalling*

Figure 3.3 Overview of Request Call Flow



function for the requested method using the function pointer returned from *getskel*. It passes the structure returned by *getskel* as an argument to the *demarshaling* function.

The *demarshaling* function provides the main functionality of the skeleton. This function is responsible for retrieving the requested methods parameter values from the input buffer, casting the function pointer passed within the EPV2 structure to the correct type and executing the resulting function on the retrieved parameter values. When the user's function returns, the return values are converted to Internet Inter-Orb Protocol (IIOP) compatible format and placed on the output buffer to be returned to the client.

3.3.3 Difficulties in Passing Function Pointers

The passing of function pointers is necessary as the ORBit libraries have no knowledge of the function structure names which are to represent the interface defined in IDL,

hence linking cannot be done at compile time. Only by passing function and structure pointers to the ORBit library functions at run-time can the skeleton and user functions be executed. This is an extremely useful aspect of the ORBit implementation, it does however introduce a problem when passing pointers to Aldor functions into C space.

One of the difficulties in this approach was producing an effective method to pass function pointers between the generated Aldor skeleton and the ORBit C library functions. In order for the C library functions to call the Aldor skeleton functions, pointers to those functions must be contained within structures and passed as arguments to the ORBit library initialization functions. This allows the ORBit library to store and later retrieve and call functions which were not available when the library was compiled. Although Aldor is capable of calling functions implemented in C, and of exporting its own functions so that they may be linked to C programs using the Aldor foreign function interface [5], passing functions as arguments between the two languages presented problems.

Aldor is a functional language, and as such each Aldor function is represented by a closure structure containing a pointer to the actual function and a pointer to the environment in which the function was created. In addition the actual function takes the environment structure as its first argument. These two properties of Aldor prevent the straight passing of function pointers into C. In addition, when an Aldor function is exported to C, the Aldor compiler generates two separate function versions, one which can be linked from the C program and the other which is referenced from the within the Aldor program. Any attempt to reference the exported function from within the Aldor program returns a pointer to the ALDOR version of the function, which is unusable in C.

In order to circumvent this problem and allow the passing of the correct function pointer to the ORBit library, a special set of C functions were created. Any Aldor function that must be called from within the ORBit library functions, namely the *getskel* function for the interface and the *dispatch* functions for each method declared in IDL, must be declared at the top level and exported to C. In addition, a special C function is generated for each of the Aldor functions. These C functions take no arguments and return an untyped pointer to the exported Aldor function. The C functions are then imported into the initialization function of each POA domain so their values can be stored in the *POA::classid* structure and passed to the ORBit library.

This method has an added effect when applied to parameterized types. Only a single function with a unique name can be exported to C. To allow for parameterized polymorphisms, we provide an additional mechanism to determine the correct *demarshaling* function, based on the object on which the method is requested. For this purpose the Aldor function pointer (which is a closure structure and so can only be executed from within Aldor) for the correct dispatch method is stored in the EPV2 structure. Since the types for the dispatch functions are known when the domain is created, these function pointers refer to the correct version of the dispatch function for the bound type. The duty of the *top level dispatch* function for the method is to retrieve the closure for the correct functionS from the EPV2 structure and execute it.

The drawback to this method is that it forces the GIDL/Aldor compiler to generate an additional C module which must be compiled and linked to the generated ALDOR code. It does however allow the Aldor skeleton to make use of run-time generic types without modification of the IIOP protocol or the ORBit library functions.

3.4 The Two-Level Approach

While the single level approach provided the desired functionality, it failed in one of the goals of GIDL: it was not portable. While the specification for the exported functions of a Portable Object Adapter are well defined, the underlying library functions differ between CORBA implementations. In order to address this issue, we chose an approach closer to that taken by the original GIDL project.

In the two-level approach, the GIDL source is used to generate both stub and skeleton code in the desired target language, and a OMG IDL file containing properly substituted IDL types in place of any parameterized definitions declared in GIDL[4]. The OMG IDL file is then compiled into the target language using a third party compiler. The GIDL generated stub and skeleton interface provide type erasure and reconstruction for parameterized types, and forward call requests through the stub and skeleton code generated from the IDL compiler. The two-level approach ensures portability between underlying CORBA implementation since all connecting points between the two levels are standardized by OMG CORBA.

In the case of the our Aldor GIDL compiler, we use Aldor's foreign function interface to connect with generated C language code. This is necessary since there is currently no OMG CORBA ORB written for the Aldor language.

Chapter 4

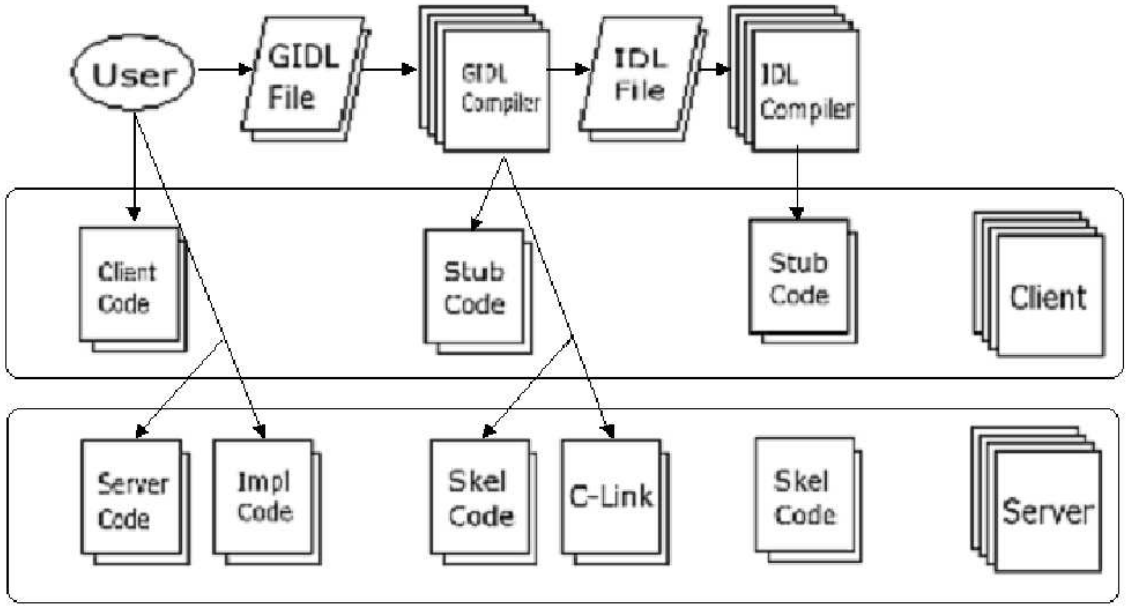
A Closer Look at the Two-Level Approach

In this section we take a closer look at the implementation details of the two-level approach. We examine the GIDL to Aldor mapping for basic and constructed types, our mapping for interfaces, and details on how type relationships are modeled. Later in this section we give a brief overview of the execution flow of a method invocation, and detail how the server side generated code operates.

4.1 File Structure

The file structure of the Two-Level Code Generation approach is based largely on that of the GIDL project. GIDL definitions are compiled by the GIDL compiler to produce stub and skeleton and linking code in the target language, and an additional IDL file containing unparameterized definitions of interfaces. In the case of our Aldor mapping, the IDL definitions are then compiled using a standardized C CORBA

Figure 4.1 Overview of GIDL for Aldor files



compiler to produce stub and skeleton code in the C programming language. The C language and Aldor stubs are then compiled and linked with a client program written by the user and the Aldor type mapping library to produce the client executable. C language and Aldor skeleton files are compiled and linked with the user written server file, the Aldor type mapping libraries, and an additional *linking* module provides compatible function signatures between Aldor and C. The result is an executable Aldor *client* program which may use the GIDL defined types written and compiled in any GIDL language, and an Aldor server which supplies functionality for the GIDL defined types to a *client* written in any GIDL implemented language.

The Generated IDL definitions are functionally equivalent to the GIDL definitions. GIDL to IDL translation is accomplished by first removing parameterization from types, then replacing *non-restricted* and *implement-restricted* type variables with the

Figure 4.2 Parameterized GIDL Definition for List

```
interface Comparator<S>{

  boolean compare(in S x, in S y);

};

interface List<S; C:) Comparator<S> >{

  List<S; C> sort( in C c);
  List<S; C> cons( in S x);

};
```

Figure 4.3 OMG IDL File Generated from List

```
interface Comparator
{
        boolean  compare ( in any  x , in any  y ) ;
} ;


interface List
{
        List   sort ( in Object    c ) ;
        List   cons ( in any  x ) ;
} ;
```

CorbaAny type. *Extend-restricted* types are substituted with the unparameterized type representing the GIDL interface they are supposed to extend[4]. The result is an OMG IDL file providing equivalent exports to those declared in the GIDL definition. Consider Figure 4.2, in this case we have two parameterized interfaces, *List* and *Comparator*. *List* defines a Lisp-like list of elements of the *non-restricted* type *S*, which may be sorted by a boolean function defined in any type belonging to the class

hierarchy of *C* which *implements* the exports of *Comparator* of *S*. The resulting IDL file can be seen in Figure 4.3.

4.2 Type Mapping

This section contains an overview of how GIDL types are mapped to Aldor domains. We also examine how the generated Aldor stub/skeleton code translates GIDL in order to provide equivalent type scoping and inheritance rules within Aldor.

4.2.1 Basic Types

Basic types are mapped by creating a new domain/type using an existing equivalent Aldor base type (from the *axllib* library) as its representation.

Integer Types

OMG IDL provides standardized integer types based on possible value ranges [1]. The Aldor abstract machine supports integers of different widths, but not all of them have full high-level functionality. In Aldor, we represent each Integer type with Aldor type/domain extending the *most functional* equivalent type supplied by Aldor. Thus, for *long* we generate a new Aldor type extending the *SingleInteger* domain. The *short* integer type has no equivalent rich domain in Aldor, so the *Sint$Machine* type is used.

Floating-Point Types

Floating point types are defined by the *IEEE Standard for Binary Floating-Point Arithmetic*. We represent these types using new Aldor domains extending the *Sin-*

gleFloat and *DoubleFloat*[1] domains for single and double precision floating point numbers, respectively.

Character Types

The GIDL character type in Aldor uses the *Character* Aldor type for its representation. This type may be *coerced* to and from either its representation type from the Aldor library, or a C style machine type. Coercion to a machine type is done transparently when required to pass character types to underlying C functions.

Boolean and Octet

Similar to the Character type, Boolean and Octet types in GIDL Aldor use Aldor library types for their representation. These types may be cast between their corresponding machine types or their representative types via the *coerce* function.

Constructed Types

Constructed types consist of the *struct* and *union* types. The *struct* type is similar to that found in C or C++, providing a single type with accessible named fields. In Aldor, the IDL *struct* type is represented as an equivalent *Record* type. However, several problems occur in this mapping. Internally declared structure types must be declared as exports of the enclosing scope (see Types and Scoping). The *Record* domain does not have a corresponding category. This causes difficulty when declaring

[1] Aldor in fact represents the *Double Float* type as a 'pointer to a IEEE double precision floating point number. This wrapping is transparent when working within an Aldor program, but can cause confusion when passing DoubleFloat arguments through the Foreign function interface, or when doing low level memory manipulation.

the *Record* type itself as an export. In order to correct this, every *struct* type declared in IDL creates a *helper* category/domain pair with a unique identifier at the top lexical scope. The *helper* Category provides exports for *apply*, *set!* and *bracket* while the domain acts as a thin wrapper for *Record*. This gives IDL structure types a well defined mapping to the Aldor language.

Structures are mapped to an equivalent Record() type in ALDOR. Structures maintain their position in the scope hierarchy in ALDOR, this is accomplished by creating domain and category generating functions in the outer scope with unique names and calling them from the correct scope.

This mapping allows programmers familiar with object oriented languages such as C++ which allow type definitions within classes or functions, to use defined structures in a familiar and natural way. Defining the structures uniquely at the top level eliminates difficulties encountered when attempting to define new structures within interfaces. Also, since the type returned by the Record type from *axlib.as* does not have a category which may be extended or referenced, the Record type could not be properly exported from within a category without the first defining a new category at the top level. In the case of structures within interfaces, the translator will generate the appropriate definitions for the structure within the interface category. The assignment will be generated in the interface domain. All CorbaStructure types export the basic functions required for marshaling, conversion to the CorbaAny type, and the generic tie-ins:

```
apply:(%, 'id') -> T;
set!(%, 'id', T) -> T;
```

For each aggregate type T in the structure, and:

```
dispose!:(%) ->(); (from CorbaRootCategory);
bracket:(Tuple Type);
```

for destruction and creation of the structures. Note that % is an ALDOR symbol representing the type currently being defined.

4.2.2 Types and Scoping

OMG IDL name scoping is very similar to that found in languages such as C++ and Java. Nested lexical scopes are formed within the IDL file by module, interface, structure, union, operation and exceptions declarations.

Aldor provides similar scoping declarations, nested lexical scopes in Aldor are introduced by the following expressions:

- *E* where `Definitions`
- `+->`
- `with`
- `add`
- `for` *i* `in ...`
- `Applications, e.g. Record(i: Integer == 12) [5]`

Similar scope rules are achieved through mapping IDL scope defining definitions to equivalent Aldor scoping expressions. For instance, modules and interfaces are easily mapped to the *add* domain creation expression, IDL structures are mapped to *Record* types, etc. While this straightforward approach provides similar scope rules at a high level, several differences appear upon implementation.

Internally Scoped Type Definitions

Type definitions that are contained within an IDL scope definition, such as an interface, must be treated as an *export* of the mapped Aldor definition. Since Aldor types belong to *categories* or views, every type declared within a domain must be accessible from outside its immediate scope. In addition, since when mapping inherited interfaces to Aldor we duplicate inherited types and methods within the child domain, scope names within all parent interfaces must be accessible to allow for proper type checking.

Figure 4.4 IDL File using an Internal Structure Definition

```
interface A{
  struct a { long f1; short f2; };
  void op1(in a p1);
};

interface B:A {
  void op2(in a p1);
};
```

Figure 4.5 Aldor Definitions Showing an Internal Structure

```
define ACategory:Category ==  with {
    a:aA____GIDL__Category;

    op1:( __this:%, __a0GIDL:a, __env:CorbaEnvironment) -> ();
};

define BCategory:Category == Join(ACategory) with {
    op2:( __this:%, __a0GIDL:a, __env:CorbaEnvironment) -> ();
};
```

The Aldor is not an object-oriented language. As such, in order to give correct

representation of inherited operations in domains, exports from parent interfaces are duplicated within the child domain. In the case of internal types inherited from the parent, the duplicated type is an *alias* of its parent type. Had the actual internal type been re-created, then *a$A* and *a$B* (from Figure 4.4, 4.5) would be identical but distinct types in Aldor.

4.2.3 Type Definitions

Type definitions given by `typedef` in IDL are mapped to assignments in Aldor. The actual ALDOR code that is generated by the translator depends on the location of the typedef in the IDL header and the type being defined. If the typedef statement is located within an interface and is a typedef of an interface then two lines are generated within the interface category, one defining the new typedef'ed category and another defining the new typedef'ed domain, the assignments for these definitions are included in default statements in the category as well as in the domain of that interface. If the typedef statement is of an interface but is outside of an interface then the 'in category' definitions are omitted and only the assignments are generated. Note: Definitions of an interface within itself are replaced with the % symbol (as in above case). Basic types (such as CorbaLong, CorbaShort, CorbaString, etc) are similar but do not have a category definition. Constructed types are assigned to a call to the CorbaAlias Domain (see Arrays and Sequences). This mapping allows for typedef'ed types and constructed types to maintain the same scope as they did in the IDL definition and allows them to be referred to in the program by their path names or if in the same scope by just their identifier names.

The declarations of `typedef`s in the generated code is purely for the user's con-

venience. In order to prevent multiple definitions of the same function when working with inherited functions, all type defined types are replaced with the original type value during code generation. This prevents a situation where a parent interface would define a function using a typedef'ed type, while its child redefines the function using the original type, causing two versions of the same function to be exported by the child interface.

4.2.4 The Any Type

CorbaAnys are represented as a Record (structure) containing a pointer to a TypeCode, a pointer to a type, and a boolean value. Conversion to a CorbaAny type from another type first retrieves the TypeCode for the type using the *getTypeCode()* function exported from that type. Conversion from a CorbaAny to a type first checks the TypeCode stored in the CorbaAny against the TypeCode for the target type. If the types do not match a 'CorbaBadTypeCastException' is thrown. All conversions to and from the CorbaAny type are done through the *coerce* function in each type.

4.2.5 Interfaces

Interfaces in IDL are used to declare types and their exports which will be accessible between client and server. Interfaces may provide operational exports, constructed types, constants and exceptions within their scope. Interfaces may also inherit exports from previously declared interfaces. Properties of inherited objects follow similar inheritance rules to that found in C++. Child objects inherit all exports of their parents, and are considered to be valid substitutions for their parents in any argument.

In order to create a new type with supplied exports for each interface in Aldor,we

must create a category/domain pair to represent each interface declared in IDL. For the client side, each interface is represented by a *reference* domain that provides marshaling functionality for each of the declared exports. This *marshaling* function converts all received arguments to *IIOP* format and transmits the result along with the export signature and identifier of the calling type to the server. In order to properly represent the type, a category is also created to declare the exports of the interface. All inheritance relationships between interfaces are handled at the category level, each category includes exports from its parents through the *Join* operation. This structure allows for IDL inheritance rules within Aldor without compromising the type safety of the Aldor language.

4.2.6 Enumerated Types

Enumerated types (as defined by GIDL) are similar to those found in languages such as C++/C/Java. Enumerated types allow programmers to create a set of named values, each of which represent an unspecified ordinal value and allow relational operations to be performed between them. Enumerations in IDL do not however allow for a specific ordinal value to be assigned to the types.

4.2.7 Parameterized Types

The GIDL [8] extension defines three distinct types of parameterized interfaces. These interfaces are handled as follows:

Implement-restricted templates allow any type satisfies all exports of the qualifier to be supplied as a parameter. In Aldor we create a category/domain pair

using unnamed category with the qualifier exports. Using an unnamed category allows the Aldor compiler to verify valid type satisfaction based on the type structure, instead of type checking by name. In addition to the exports declared in the GIDL declaration of the qualifier type, the unnamed category must also export functionality for basic CORBA operations (i.e. those exports declared in CorbaRootCategory). To enforce this, the unnamed category extends CorbaBasicCategory ensuring the supplied type belongs to the basic set of Corba compliant types. As such, for any implement restricted parameter *A* in *interface B<A:-C>*, a valid type for *A* must supply all exports declared in *C* and belong to the Category, or a sub-category of, *CorbaRootCategory*.

Extend-restricted templates allow for type that directly or indirectly inherits from the restricting interface to be supplied as a parameter to the type. Since Aldor inheritance is simulated at the category level, parameterized interfaces require a domain that belongs to or inherits from the restricting category.

Non-restricted templates allow any type belonging to the CORBA/GIDL type hierarchy to be supplied as a parameter to the interface. These types most closely resemble unrestricted templates in C++. In Aldor, these types may accept any type belonging to the *CorbaRootCategory* Category as a parameter.

Since the GIDL specification for parameterized types uses a *type erasure* technique[8], whereby type information of parameterized types passed as arguments are removed during transmission, all parameterized types are marshaled as the *CORBA-Any* type. Lost type information is regained using the generated client/server code after transmission. Details of this process are examined in section 4.2.10.

4.2.8 Providing Inheritance

One of the fundamental principles of GIDL and other object-oriented architectures is the concept of inheritance relationships. In GIDL, any interface may inherit from zero or more interfaces, provided that the resulting inheritance tree is acyclic. Such *child* interfaces inherit all exports of its *parent* interfaces and are considered to be valid substitutions for any interface higher in the inheritance tree for the purposes of type checking.

Since Aldor is not an object-oriented language, we must model these inheritance relations artificially within an *aspect-oriented* environment. As previously mentioned, GIDL interfaces may be represented by category domain pairs. Categories provide all valid exports of both interface and parents through the *Join* category operation, which also provides part of the substitution functionality we require. The domain provides definition of the interface type and *belongs* to its corresponding category. Since domains belong not only to their own category, but every category included in the *Join* operation, each domain type *belongs* to every category representing its parent interfaces. Using this property, child for parent substitutions are allowed by supplying the type, quantified as belonging to a specific category representing an interface, followed by a variable of the supplied type. From this it is easy to see that only valid types belonging to the inheritance tree rooted at the required interface may be supplied to a function, and only a valid variable of the supplied type may be used by a function. This technique allows for sufficient inheritance emulation on the client, allowing interface method to accept the same range of types as defined by GIDL.

Since the server is required to provide actual functionality for inherited interfaces, the implementation on the server side is more complicated. In order to fully explain

how inheritance is achieved, we must first examine the difference between *reference types* and *actual types*. *Reference Types* are the same reference objects used on the client side to provide access to an object located on a server. These types provide no implementation other than to initiate a remote method call on the referenced object, as such, these objects may be type cast into another reference object with the same exports without ill effect, as the actual object being referenced does not change. *Actual Types* are the actual implementation types for a defined interface. These types exist only on the server. When a method call is executed, the reference is used by the server to retrieve the *Portable Object Adaptor* which contains the *actual* domain implementation. The *POA* is then used determine the method being called, demarshal the methods arguments, and call the method on the *actual* instance. All demarshaled arguments are guaranteed to be either *reference types* or basic types. Hence, the only time the actual type of a Domain is seen is when it is executing a function that belongs to it.

Aldor type casts erase all knowledge of an object's original domain. If *domain A* and *domain B* both provide an export *op1*, then an instance of *domain A* may be cast to *domain B*, however attempting to call *op1* on the newly cast function will execute the code found in *domain B*. This behavior is contrary to what would be expected in a GIDL environment between related interfaces.

Since all arguments are reference types, any argument may be type cast to its parent type without violating type safety. A method requiring an argument of *reference type A* may be supplied a *reference type B*, and safely treat it as *reference type A* within the function. Since stringified reference of the *reference* type remains unchanged, all method calls made to the object will be invoked on the original *type B* implementation. We can now be assured that even if a child reference interface is

supplied in place of a parent when calling a method, the correct functions from the child interface will be called.

4.2.9 The Stub Implementation

The purpose of the GIDL stub code is to act as an interface to the IDL generated stub and to provide the type structure necessary to use the GIDL defined type naturally within ALDOR programs. To accomplish this we create a *reference type* for each interface declared in GIDL. Reference types are artificial Category/Domain pairs used to represent a remote type/object within the *local* program. Within Aldor, these types are wrappers for the stub code found in the underlying SCA.

Stub classes in CORBA are representations for an *IOR* (Inter-operable Object Reference). An *IOR* is a unique string identifying an actual object located on a server. Within an aldor program the IOR forms the underlying representation for a complete domain type, which provides exports for methods declared in its GIDL definition.

4.2.10 The Skeleton Implementation

The GIDL compiler would produce two sets of POA (Portable Object Adapter) code for each interface defined in GIDL. If the GIDL interface was a parameterized type, then one of the pair of POA types generated would share the same parameters as the interface definition, the second is *type-erased* and has no parameters but has an identical memory representation as the parameterized POA. The *type-erased* POA contains a pointer to its parameterized type, stored as a value. When any function implementation is called belonging to the *type-erased* POA, the parameterized POA

type is retrieved and the *POA argument* is re-cast and the function re-called. This shows us one of the major advantages to working with aldor, in that since types may be treated as values any structured type may contain a type field and therefore be re-created after type-erasure.

In ORBit, when a client request was received, the ORB uses the transmitted IOR to retrieve a *type-erased* POA containing the *servant* (user implemented) type. The ORB then calls a *dispatch* function belonging to the POA. The *dispatch* function is responsible for parsing the invoked function name (passed as a string) and returning a function pair, a function used to demarshal the function arguments, and the user implemented function (*_impl*). The *_impl* arguments are then demarshaled and the function invoked.

In Aldor, we use the generated *Clink* file to overwrite the *_impl_POA* structure generated by ORBit. Instead the GIDL compiler produces an identical structure with an additional pointer field used to store the parameterized type value of the POA, and a pointer field to store the user implementation of the domain, which may be arbitrary. Within the aldor *skel* file, the *_impl_POA* structure is recreated for each interface as a domain with exports for initialization and all methods declared in the GIDL definition. If the GIDL interface is generic then the *_impl_POA* type is parameterized according to our mapping for parameterized types, also an additional *type-erased POA domain* is created. Within the Aldor POA domains, the call to *init* calls the generated ORBit *init* function, and stores the POA type as a value within the structure. This type field is used after type erasure to restore the generic type information lost between client and server.

Implementation functions for the generic *_impl_POA* type are written by the user.

Functionality may be supplied at this level, or these functions may be made to call equivalent functions belonging to an underlying type, stored in *_impl_POA.obj*. User functionality is provided at the POA level so that structures such as the CORBA ORB and POA are available in the event that they are required for the implementation, such as the creation and return of new CORBA controlled objects.

The connection between the ORBit and Aldor skeleton is provided by the Clink file. Each Aldor *_impl* function is exported to C via the foreign function interface. In the case of generic types, the *_impl* functions from the *type-erased* domain are exported. Clink provides one function for every export, with a function signature conforming with the CORBA C language mapping standard, which invokes the exported Aldor methods. This additional level of redirection is required since Aldor functions are exported to C with a specific type signature which differs from that required by C CORBA mappings. Attempting to construct Aldor functions which export directly to the required C function signature would have severely limited the GIDL to Aldor function mapping and would have resulted in a more cumbersome solution.

Chapter 5

Example Programs

In this section we introduce some basic examples of Aldor/Corba programs. These examples are intended to show the basic structure described in previous chapters. Section 5.1 introduces a simple text program in which strings sent from an Aldor client are printed from a server implemented in C++.

Section 5.2 discusses an Aldor server for the an implement-restricted parameterized linked list. The purpose of these examples is not to show the complete working code, but rather to give an overview of concepts described in previous chapters.

5.1 Echo

The *echoString* example should be familiar to anyone who has read any CORBA tutorial book. One of the simplest possible examples, we use *Echo* to introduce the basic operations of Aldor/GIDL.

The Echo program is similar to the classic 'Hello World' program introduced in Kernighan and Ritchie's "The C Programming Language", but with a slight twist.

Echo will pass the string from an Aldor client to a C++ implemented server for printing.

5.1.1 Aldor

Figure 5.1 Generated Client Stub for Echo.idl

```
define EchoCategory:Category == with {
   echo__string:( __this:%, __a0GIDL:CorbaString,
                  __env:CorbaEnvironment) -> (CorbaString);
};

define Echo__refCategory:Category == Join ( EchoCategory,
                  CorbaObjectCategory) with {};

Echo__ref:Echo__refCategory == CorbaObject add {
    Rep == CorbaObject;

    echo__string( __this:%, __a0GIDL:CorbaString,
                  __env:CorbaEnvironment):(CorbaString) == {
        import {
           echo__echo__string:( %, CorbaString, CorbaEnvironment)
                  -> CorbaString
               } from Foreign C;
           echo__echo__string(__this, __a0GIDL, __env);
    };
};
```

The generated code for *Echo* is fairly simple. Figure 5.1 shows part of the generated stub code. The GIDL compiler produces a small category declaring exports, and a *__ref* category used to define exports for the *reference* type.

The *echo__string* function imports the generated ORBit stub function shown in Figure 5.2 and calls it directly.

Figure 5.2 shows the ORBit stub function generated by the ORBit2 IDL compiler.

Figure 5.2 ORBit2 Generated Stub Code for Echo.idl

```
void echo_echo_string(echo _obj, const CORBA_char *f1,
                      CORBA_Environment *ev){
    gpointer _args[1];
    _args[0] = (gpointer &f1);
    ORBit_c_stub_invoke (_obj, &echo__iinterface.methods,
                         0, NULL, _args, NULL, ev, echo__classid,
                         G_STRUCT_OFFSET(POA_echo__epv, echostring),
                         (ORBITSmallSkeleton)
                          _ORBIT_skel_small_echo_echostring
                         );
}
```

Notice the call to ORBit_c_stub_invoke, the ORBit2 function which will marshal the given arguments and transmit the request for the server. Unlike ORBit (version 1.5) which generated the marshaling code within the stub functions, ORBit2 makes extensive use of structures generated at IDL compile time, which are used to store data about each IDL defined interface. Structures such as *echo_iinterface* are referenced to determine at runtime each arguments type and size when marshaling. In ORBit, these attributes would be determined at compile time and a more specialized function would have been generated. This difference between CORBA ORBs produced by the same vendor illustrate one of the primary reasons for the two-stage compiler approach.

Figure 5.3 shows the generated *POA* category for *Echo*. These functions provide wrappers for the *POA* generated by the ORBit2 IDL compiler.

Figure 5.4 shows the client implementation for Echo.idl. This code shows the main body of the client program and is written entirely by an end user, no generated code is included. The Aldor Echo client is divided into blocks (Figure 5.4, Figure 5.5 and 5.6) for easier discussion, each block is denoted by an Aldor comment similar to "Code Block #1 Begins" and is ended with the comment "Code Block #2 Begins".

Figure 5.3 Part of the Skeleton for Echo.idl

```
define POA__EchoCategory:Category == with {
    activate:(CorbaPortableServerPOA,
              CorbaObjectID,
              %, CorbaEnvironment) -> ();
    servantToReference:(CorbaPortableServerPOA, %,
                        CorbaEnvironment) -> Echo__ref;
    New:(Echo) -> %;
    init:( Echo, CorbaEnvironment) -> %;
    ...
};
```

Figure 5.4 Implementation of the Aldor Echo Client (Block #1)

```
#include "echo-stub.as"
--Code Block #1 Begins
import from CorbaEnvironment, CorbaORB;

env:CorbaEnvironment == init();
orb:CorbaORB == init(env);

--Code Block #1 Ends
```

This section is described in more detail to give an understanding of how an end user fits into the CORBA model, and also a limited understanding of Aldor syntax. Readers are encouraged to consult the Aldor.org web site [5] for more information on the Aldor language and syntax.

Code Block #1 (Figure 5.4) shows our initialization calls for the CORBA ORB and CORBA Environment wrappers. The CORBA environment type is passed to all CORBA functions as the final argument and provides context information to the underlying ORB.

Code Block #2 (Figure 5.5) first redefines the Echo_ref type introduced in Figure

Figure 5.5 Implementation of the Aldor Echo Client (Block #2)

```
--Code Block #2 Begins

Echo ==> Echo__ref;
file:FileName == filename("./echo.ior");
fp:TextReader == reader(file);

ior:CorbaString == readline!(fp)::CorbaString;
print <<$String "using " << ior << newline << newline;

try {
    echoclient:Echo == stringToObject(orb , ior, env);

    if (nil? echoclient) then
       never;

} catch E in {
   print <<$String "FAIL" << newline;
};
-- Code Block #2 Ends
```

5.1 to Echo using an Aldor macro definition(==>). The server generated IOR referencing the Echo object on the server is read from the "echo.ior" file and a new Echo_ref domain is created using the *stringToObject* function included from the CorbaObject domain (Echo_ref's parent). Notice that although defined in CorbaObject, the type of the return from *stringToObject* is an Echo_ref domain, illustrating one of the nicer aspects of Aldor static polymorphism. Under the hood, *stringToObject* makes a call to the underlying CORBA ORB to contact the server, which is encoded within the IOR, and validate that our Echo object exists. Upon a successful return, *echoclient* represents a validated reference to an Echo object to used as if it were a local domain.

Code Block #3 (Figure 5.6) shows the body of the Echo program, simply reading text from standard input and calling the echo_string function (shown in Figure 5.1).

Figure 5.6 Implementation of the Aldor Echo Client (Block #3)

```
-- Code Block #3 Begins
while (x:CorbaString := rightTrim(readline! stdin, newline))
       ~= "." repeat{
   try {
       print <<$String "client: "
             << echo__string(echoclient, x, env)
             << newline;
   } catch E in {

       if hasException?(env) then {
           print <<$String "an exception has occurred"
             << getException(env)
             << newline;
           never;
       };
    }
-- Code Block #3 Ends
};
```

The given argument (a CorbaString), is passed to the ORBit generated stub function shown in Figure 5.2 and marshaled to the server.

5.1.2 C++

The Echo Server

The Echo.idl server implementation is fairly straight forward with respect to its functionality. The following section presents the MICO C++ ORB code for the Echo server, this code is included for completeness with our Aldor Echo client, and is not intended to be a full description of C++ Corba. Describing the grit of CORBA is well beyond the scope of this thesis.

Figure 5.7 shows the user implementation of the *Echo* type. The *echo_string*

Figure 5.7 C++ Server Side Implementation of Echo

```
namespace GIDLImplem
{
  class Echo_Impl : public virtual POA_GIDL::Echo,
                    public virtual ::PortableServer
                                   ::RefCountServantBase
  {
      public:
          Echo_Impl(){}
          virtual GIDL::String_GIDL
                  echo_stringGIDL(GIDL::String_GIDL a1_GIDL)
              throw (CORBA::SystemException)
          {
              cout << a1_GIDL.getValue() << endl;
              return a1_GIDL;
          }
  }
}
```

function is coded similar to any other C++ member function, and simply prints its argument to standard output.

5.2 An Implement-Restricted List Example

This section gives an overview of the Aldor client and server implementation for an parameterized List type in Aldor. The List interface is the same as defined in 4.2, using an unrestricted type for storage, and a implement-restricted template to restrict the Comparator type, the GIDL definition is presented in Figure 5.8. The result is a sortable linked list implementation that can be supplied a comparator function from any GIDL enabled language.

Unfortunately the generated source code is quite long to be included as an example

Figure 5.8 GIDL Definition for a Parameterized Linked List

```
interface Comparator<S>{
  boolean compare(in S x, in S y);
};

interface List<S; C:) Comparator<S> >{
    List<S; C> sort( in C c);
    List<S; C> cons( in S x);
    C getC();
};
```

(spanning several pages) and is presented only as an overview of the capabilities of GIDL and Aldor.

Figure 5.9 Client Side Implementation for a Comparator Domain

```
Comp(S:CorbaLongCategory):ComparatorCategory(S) ==
        Comparator(S) add {

   compare(__this:%, x:S, y:S, __env:CorbaEnvironment)
        :CorbaBoolean == {
      import from S;
      return (x > y)::CorbaBoolean;
   }
}
```

Figure 5.10 The Aldor Client for a Parameterized Linked List

```
--Code Block #2
--start a server to handle request to a 'Comp' Object.
poa:CorbaPortableServerPOA ==
          resolveInitialReferences(orb, "RootPOA", env);

import from Pointer;
if (nil? poa) then
    if (hasException? env) then {
        import from CorbaException;
    }
print <<$String "POA init OK" << newline;

myPOA:POAComparator(CorbaLong, Comp(CorbaLong)) == init(
         0$SingleInteger pretend Comp(CorbaLong)
         , env);

mangr:CorbaPortableServerManager == getPOAManager(poa, env);
activate(mangr, env);

objid:CorbaObjectID := [ 0, 5 , "list"];
activate(poa, objid , myPOA, env);

comp__serv:Comparator__ref(CorbaLong) ==
             servantToReference(poa, myPOA, env);
-- End of Block #2
```

Figure 5.11 shows the basic client body using a the List type defined in Figure 5.8. For brevity, "Code Block #1" is omitted and remains identical to that described in Figure 5.4. Figure 5.10 shows a small section omitted from Figure 5.11 in place of "Code Block #2". This block creates a CORBA server to supply an implementation for a Comparator domain. This allows our client to act as a server for a user implemented Comparator function (wrapped in a domain) implemented in Figure 5.9. A reference domain named comp_serv is created and passed as an argument to the List server in from the sort function in Figure 5.11. Notice that no IOR is generated

Figure 5.11 The Aldor Client for a Parameterized Linked List

```
import from CorbaEnvironment, CorbaORB;

-- Code Block #1
...
-- End Code Block #1

list ==> List__ref(CorbaLong, Comparator__ref(CorbaLong));
import from list;

ior:CorbaString == readline!(fp)::CorbaString;
print <<$String "using " << ior << newline << newline;

-- Code Block #2
-- Described in separate figure
-- End of Block #2

--run the program
try {
   listclient:list := stringToObject(orb , ior, env);

   if (nil? listclient) then
       never;

} catch E in {
   print <<$String "FAIL" << newline;
};
```

for the Comparator type, the server receives a complete reference through comp_serv, routing function calls made to that domain back to the client. The remaining logic is shown in Figure 5.12, and simply creates a list using *cons* and calls sort, passing in the domain wrapped comparison function.

In implementation of the List domain presented in Figure 5.14. Several interesting aspects of the Aldor Corba are included in Figure 5.14, the first showing the implementation of implement-restricted template in Aldor. The second parameter of List

Figure 5.12 The Aldor Client Body for a Parameterized Linked List

```
try {
   import from CorbaLong;
   for i in 1..20 repeat {
     print <<$String "adding " << i* 31 mod 19
           <<$String " to list"
           <<newline;
     listclient := cons(listclient,
                        (i * 31 mod 19)::CorbaLong,
                         env);
   };

  sort(listclient, comp__serv, env);

} catch E in {
  import from CorbaException;
  if hasException?(env) then
    {
     print <<$String "an exception has occured"
           << getException(env)  << newline;
     never;
    };
}
```

does not restrict the domain C to a defined Category, but rather an unnamed category build from the ComparatorCategoryMacro defined in Figure 5.13. This allows any domain which implements the functions declared in ComparatorCategory to be used , without requiring the extension of ComparatorCategory. Essentially, any Corba GIDL declared reference type declaring a compare function over S may be used.

The Aldor library defined List type is used as the representation for this domain. Using the *rep* and *per* macros to convert or type to its representation and GIDL List type respectively, we are able to effectively wrap the List library type and export it through GIDL. In order to sort our wrapped list, we create a function closure f, which

Figure 5.13 Definition of the CompareCategoryMacro

```
ComparatorCategoryMACRO(S) ==> with {
   -- sequences/arrays

   compare:( __this:%,
             __a0GIDL:S,
             __a1GIDL:S,
             __env:CorbaEnvironment) -> (CorbaBoolean);
   marshal:(SendBuffer, %) -> ();
   demarshal:(ReceiveBuffer) -> %;
   marshaltemplate:(SendBuffer, %) -> ();
   demarshaltemplate:(ReceiveBuffer) -> %;
   objectToReference:(%,
             CorbaPortableServerPOA,
             CorbaEnvironment) -> CorbaObject;
   coerce:(CorbaAny) -> %;
   coerce:(%) -> CorbaAny;
};
```

calls the compare function (implemented on the client) using __a0GIDL, and __env from the enclosing scope. This allows a true binary comparator function over S to be passed to the underlying representation List type, showing the advantages gained through the use of Aldor.

Figure 5.14 Server Side Implementation of a Sortable List Domain

```
List(S:CorbaRootCategory,
     C: with ComparatorCategoryMACRO(S)):ListCategory(S, C) == add {

   Rep == Record(__poa:CorbaPortableServerPOA, l:List(S));
   cons( __this:%, __a0GIDL:S, __env:CorbaEnvironment):
         (List__ref(S, C)) == {

        import from Rep;
        l:% ==  per [ (rep __this).__poa,
                      cons( __a0GIDL, (rep __this).l)$List(S)];

        --lift the List(S) instance to a corba controlled __ref type
        a:List__ref(S,C) == objectToReference(l,
                                              (rep __this).__poa,
                                              __env);

        return a;
   };

   sort( __this:%, __a0GIDL:C, __env:CorbaEnvironment):
         (List__ref(S, C)) == {

        f(x:S, y:S):Boolean == {
            compare(__a0GIDL, x, y, __env)::Boolean;
        };

        l:% == per [(rep __this).__poa, sort!( f, (rep __this).l)];
        objectToReference( l, (rep __this).__poa, __env);
   };

};
```

Chapter 6

Conclusions

6.1 Advantages of a Semantic Matching between Template and Functional Generic types

With the introduction of GIDL for Aldor, programmers are able to use Aldor's intuitive type system for mathematical objects from any programming environment implementing GIDL. One distinct advantage to this approach is that C++ programmers can utilize generic libraries that can be pre-compiled and compile time-check for type safety within Aldor. Since Aldor compiles and type-checks generic types during compilation and generates instantiated types at runtime, Aldor provides a much *safer* generic programming environment than the C++ substitution approach to generic libraries. This makes finding bugs in a program far easier and more reliable since programmers need not sift through errors in library code generated by a simple type mismatch or undefined symbol. In addition, Aldor programmers may use GIDL to take advantage of the numerous libraries written for C++, making full use of generic

types while still programming within Aldors intuitive type system.

6.2 Why the Two-Level Compile Method Was Best

The OMG CORBA standards do not specify the internal workings of the CORBA ORB. Instead the standards specify sets of exports for the POA, function signatures for IDL defined methods in each language, and the GIOP language and protocol used to transmit between client and server. Since the details of the ORB are left to the implementer, attempting to build an extension such as GIDL by directly linking with an existing ORB cause severe problems when attempting to change ORB implementations or versions. The two-level approach allows the level of portability we were attempting to achieve in this project. By generating an accessor layer of Aldor code to work on top of an existing C language SCA, we can ensure that we use only OMG standard function calls and in doing so ensure that our implementation may be used with any Standard ORB implementation with minimal modification. In addition, by simply changing the code generated in the C-Link file, which acts as a compatibility layer between the generated Aldor code and underlying SCA, it should be possible to use our implementation with similar SCA such as SOAP, or DCOM.

At the time of this writing, most major SCAs still in use do not support parameterized types. The one exception is the Microsoft .Net framework. Unlike other software component architectures, such as CORBA and SOAP, which define interfaces which may be used between components, the .Net framework instead compiles all implemented languages down to the same intermediate language (MSIL) which runs on a common language runtime environment (CLR). This approach allows .Net to unify multiple languages without requiring the end-user programmer to worry about

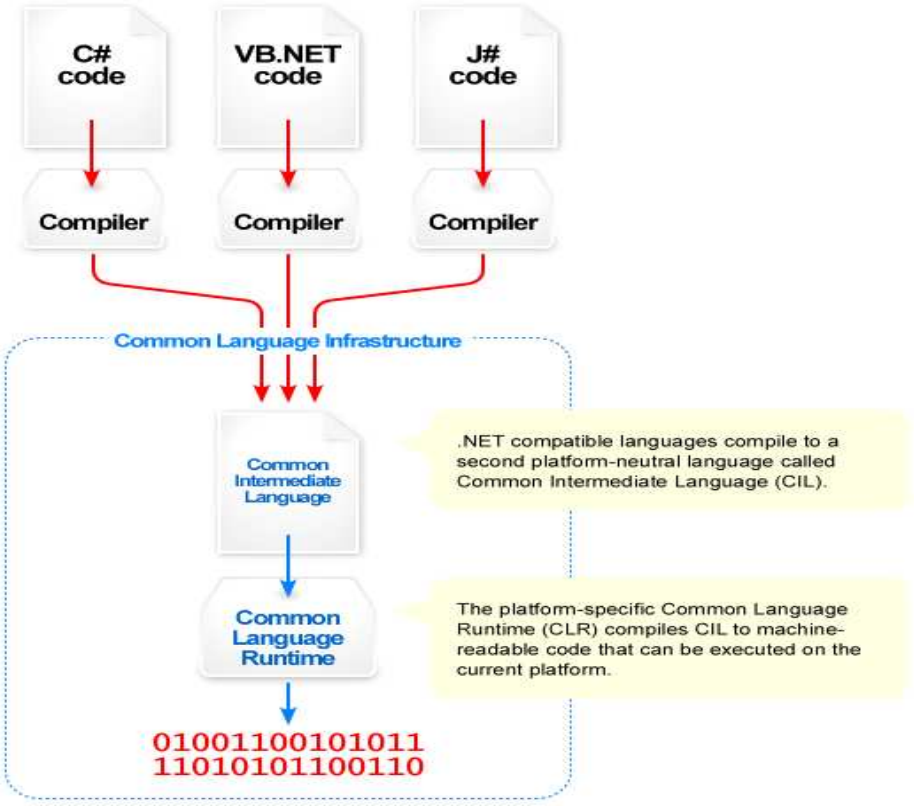
infrastructure details such as marshaling or client/server implementation. However one could reason that the disadvantage is the requirement that new compilers must be created in order to accommodate new languages.

As documented in [12], the proposed .Net support for generics implements generic types directly within the Common Runtime Library, allowing languages with support for parameterized types to translate easily to generic MSIL types. This approach provides more native support for generics than simply attempting to 'compile-away' parameterized types.

6.3 A Comparison of GIDL and .Net

To contrast to .Net, GIDL behaves similarly to previous Software Component Architectures by allowing programmers to define a set of shared objects using an IDL. Instead of compiling program components down to a shared intermediate language, the GIDL compiler produces marshaling code for source languages which may then be compiled using a third party compiler. GIDL users are not required to use a specific language compiler but may use any compiler for the implemented language they desire. In addition, languages with no support for generics may still use generic types imported from generic languages through GIDL, it is unclear if this feature is available through .NET.

Figure 6.1 Overview of .Net Framework CLR *(Source: http://en.wikipedia.org, Valid November 22, 2007)*



6.4 Inter-Operation of Compile-time and Runtime Generics

Compile time generics, such as those found in C++, have several advantages over the runtime generics presented in a statically typed language such as Aldor. C++ for example allows generic types to be defined using unrestricted template parameters, which can accept any type. Such template parameters are substituted at compile time, and using lazy instantiation, need only provide the properties required to compile functions which are called within the compiling program. This allows for types to be used in a template that would break several functions provided those functions are

never called. Additionally, these generic types can be extremely small when compiled, as only types and functions which are used will appear in the compiled program. For these reasons, as well as the prevalence of C++ libraries and functions already using compile-time generics, interacting with these languages from other generic languages.

Run-time generics also have their advantages, namely that new types can be created and used as template parameters without needing to recompile the generic type. For strongly typed languages such as Aldor, the added advantage comes when creating pre-compiled libraries exporting parameterized types. Since all type checking is done when the library is compiled, programs can use the exported types without the added compilation overhead or cryptic compilation errors that can result from using incorrect parameters (such as types that fail to implement a particular interface). This allows libraries written in these languages to more clearly and openly expose there generic types.

By using both together we can take advantage of both, for instance, by fully exporting a C++ template library through GIDL to Aldor, we can use Aldor generics to type check all potential arguments and operations at compile time, based on the restrictions defined in GIDL. When done correctly, the GIDL functions as a screening layer, validating types within the Aldor client. Of course this would still require the server side library to be recompiled when adding a new template parameter. However, with the use of implement restricted templates, a single reference type of the desired interface may be supplied as a template parameter. Thus allowing new types to be used from the client side without compilation, provided of course that all functions within the generic type respect the defined interface. Alternatively, using Aldor function closures, Aldor library types can be pre-compiled and easily exported to languages such as GIDL. Even types accepting functions as parameters (such as

the example in Figure 5.14) can be exported without requiring excessive work. Using GIDL, languages with distinct binding times for generics can inter-operate, creating a rich and powerful programming environment.

References

[1] Object Model Group (OMG). *Common Object Request Broker Architecture – OMG IDL Syntax and Semantics*. Revision 2.4 (October 2000), OMG Specification, 2000.

[2] Oancea, Watt. *Generic IDL: Parametric Polymorphism for Software Component Architectures* (2001).

[3] Henning, Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Boston, New York, Toronto (1999).

[4] Oancea, Watt. *Parametric Polymorphism for Software Component Architectures*. University of Western Ontario (2005).

[5] *Aldor User Guide*, http://www.aldor.org/aldoruserguide/, 2003. (Valid on December 22, 2005).

[6] *FRISCO project*. ESPRIT Fourth Framework project. LtR 21.024.

[7] *ORBit Corba 2.2 compliant Object Request Broker*. http://www.gnome.org/projects/ORBit2/. (Valid on October 22, 2007).

[8] Chicha, Lloyd, Oancea, Watt. *Parametric Polymorphism for Computer Algebra Software Components*. Ontario Research Center for Computer Algebra – University of Western Ontario, 2004.

[9] *Java 2 Standard Edition, version 1.4.2*. http://java.sun.com/1.4.2/index.jsp (Valid on December 22, 2005).

[10] Grabmeir, Kaltofen, Weispfenning (Eds.). *Computer Algebra Handbook*. Springer-Verlag Berlin Heidelberg New York (2000).

[11] Sethi. *Programming Languages, Concepts & Constructs 2nd Edition*. Addison-Wesley Longman Inc. AT&T (1996).

[12] Kennedy, Syme. *Design and Implementation of Generics for the .Net Common Language Runtime*.Microsoft Research, Cambridge, UK. Conference on Programming Language Design and Implementation (PLDI) (2001).

[13] Vandevoorde, Josuttis. *C++ Templates: The Complete Guide.*Addison-Wesley Pearson Education Inc. (2003).

[14] Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley AT&T Labs, Murray Hill, New Jersey. (1997).

Vita

Name:	Michael Lloyd
Post-secondary Education and Degrees	University of Western Ontario London Ontario, Canada 1999-2003 B.Sc
Honours and Awards:	Natural Sciences and Engineering Research Council (NSERC) Undergraduate Summer Research Student Award (2003).
Related Work Experience:	Teaching Assistant University of Western Ontario. London, Ontario 2003-2004 Research Assistant University of Western Ontario. London, Ontario 2003-2006

Publications:

Y. Chicha, M. Lloyd, C. E. Oancea, and S. M. Watt,
"Parametric Polymorphism for Computer Algebra Software Components,"
6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNASC'04, Sept. 2004, Universidad de Santander, Spain.