Parametric Polymorphism for Software Component Architectures

Cosmin E. Oancea
Computer Science Department
The University of Western Ontario
London, Ontario, Canada
coancea@csd.uwo.ca

Stephen M. Watt
Computer Science Department
The University of Western Ontario
London, Ontario, Canada
watt@csd.uwo.ca

ABSTRACT

Parametric polymorphism has become a common feature of mainstream programming languages, but software component architectures have lagged behind and do not support it. We examine the problem of providing parametric polymorphism with components combined from different programming languages. We have investigated how to resolve different binding times and parametrization semantics in a range of representative languages and have identified a common ground that can be suitably mapped to different language bindings. We present a generic component architecture extension that provides support for parameterized components and that can be easily adapted to work on top of various software component architectures in use today (e.g., CORBA, DCOM, JNI). We have implemented and tested this architecture on top of CORBA. We also present *Generic Interface Definition Language* (GIDL), an extension to CORBA-IDL supporting generic types and we describe language bindings for C++, Java and Aldor. We explain our implementation of GIDL, consisting of a GIDL to IDL compiler and tools for generating linkage code under the language bindings. We demonstrate how this architecture can be used to access C++'s STL and Aldor's BasicMath libraries in a multi-language environment and discuss our mappings in the context of automatic library interface generation.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Modules and interfaces, Software libraries; D.2.11 [**Software Engineering**]: Languages—*interconnection*; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Languages, Design

Keywords

Antiunification, Generics, Parametric Polymorphism, Software Component Architecture, Templates

1. INTRODUCTION

This paper examines what is required to have multi-language parameterized components interoperate and how to access existing generic libraries across language boundaries. We propose a common model for parametric polymorphism that accommodates a representative range of different object semantics and binding times from various languages and use it to design a "generic" software component architecture extension that can be applied on top of most component architectures in use today.

Software component architectures provide mechanisms for software modules to be developed independently, using different programming languages, and for these components to be combined in various configurations to construct applications. To provide the richest environment, these architectures have historically attempted to capture the intersection of features of the programming languages for which they have bindings. Common programming practice has evolved a great deal, however, since the component architectures in common use today were established. Notably, parametric polymorphism has evolved from a beautiful property of research-oriented programming languages to become a standard feature of languages used in mainstream applications. The concept of multi-language, multi-platform components must similarly evolve if we wish our components to enjoy the benefits of parametric polymorphism.

Parametric polymorphism is one mechanism by which programming languages may provide support for generic programming. By associating all behavior of parameter values with the types of the parameters, it becomes possible to write generic programs. These types can either be stated explicitly as parameters to a module, or inferred, depending on the setting. Explicit parametric polymorphism has become more widely used, in practice, and has certain theoretical benefits, including termination of type inference in some higher order languages [18]. Parametric polymorphism increases the flexibility, re-usability, and expressive power of the programming environment, avoids the need for downcasting and allows a compiler to find more programming errors. There are quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics. We review a few, to give an idea of the range a general facility must be able to map onto. Our conclusion is that a mechanism to combine modules in different programming languages must be able to accommodate both *compile-time* and *run-time* instantiation of modules and both *qualified* and *unqualified* type variables. (Here, and throughout the paper, we use the term *qualified* as a synonym for *bounded quantification* [2].)

Our work was inspired by an early experiment [4], briefly presented in Section 2, where two languages with different parametric polymorphism semantics and different binding time models were made to work together. The experiment linked C++, with compile time template instantiation, and Aldor, with run time higher-order functions producing dependent types. This experiment motivated the present, more general, approach.

We have developed an extension to CORBA's Interface Definition Language (IDL) to support parameterized interfaces. We have dubbed this extended specification language *Generic IDL*, or GIDL for short. In this paper we present our implementation of GIDL, which consists of a GIDL to IDL compiler and code generators implementing C++, Java and Aldor bindings. GIDL encapsulates a common model for generics and provides efficient implementation under a wide spectrum of requirements for specific semantics and binding times of the supported languages. Our component architecture extension does *not* assume a homogeneous environment. Its design, which constitutes in our view a novel application of the type erasure technique to implement generics in a heterogeneous environment, allows it to be easily adapted to work on top of most software component architectures in use today: CORBA is just our working study case.

To test the effectiveness of our model for generics, we have investigated how to use GIDL as a vehicle to access *two generic libraries* beyond their original language boundaries. The first library experiment implements a server incorporating part of the *C++ Standard Template Library* (STL) functionality. We have not re-written the STL: our implementation uses the STL as a black box, wrapping it in a manner that can easily be automated. We find that GIDL is perhaps more suitable than C++ to express the STL "orthogonality" semantics. Our specification is self-explanatory and self-contained, in the sense that it does not need free language annotation to explain type safety constraints. The second library experiment explores the high-level conceptual ideas involved in mapping the semantics of the Aldor BasicMath library to a GIDL specification. We see that Aldor's functional model of polymorphism can be mapped naturally into GIDL.

We see the main contributions of this paper as:

- recognition of parametric polymorphism as important to support in a multi-language environment,
- identification of polymorphism semantics suitable for use in this setting,
- the definition of an interface language, GIDL, with mappings for three very different target languages and
- a report on our implementation experience.

The remainder of this paper is organized as follows. Section 2 compares our generic model with different parametric polymorphism models in various programming languages. It also presents the current mainstream software component architectures and argues the generality of our proposed design. Section 3 describes GIDL's semantics and the GIDL to IDL translation. Section 4 presents the high level ideas used in the mapping of the generic type qualifications. Section 5 introduces the general architecture of the GIDL base application. Sections 6 describes the GIDL bindings of the supported languages (C++, Java, Aldor). Section 7 examines the effectiveness of the GIDL generic model, by exposing parts of *two* existing generic libraries to a multi-language environment via GIDL. Finally, Section 8 concludes the paper.

2. BACKGROUND AND MOTIVATION

This section lays out the background and context for our work. We first explain the original motivation and the design point we desire to satisfy. We then lay out the context of the work, summarizing the parametric polymorphism semantics of a few languages to give an idea of the range a general facility must be able to map on to. Finally, we briefly review several software component architectures in use today to understand how they can be extended with parametric polymorphism.

2.1 Motivation and Design Point

The initial motivation for our work arose building a linkage between Aldor and C++ in the context of a European project for symbolic-numeric computation. The two main background items brought into the project were (1) a complex, heavily template-based C++ library, PoSSo, for the exact solution of multivariate polynomial equations over various coefficient fields, and (2) an optimizing compiler for a higher-order programming language, Aldor, used in computer algebra. One of the specific objectives of the project was to allow Aldor programs to make use of the PoSSo library. From this very practical problem arose the interesting challenge to make two languages with very different binding time models and parametric polymorphism semantics work together (C++ with compile-time template class instantiation and Aldor with run-time higher-order functions producing dependent types). A detailed account of this work is given elsewhere [4, 5]. This experiment established that we could overcome the C++/ Aldor semantic gap and motivated our search for a systematic solution for parametric polymorphism for components, encompassing more languages in a simpler way.

Since the semantics of generics are different in various programming languages, we have been forced to identify a common ground that can be suitably mapped to these languages efficiently. GIDL supports mappings to Java, C++ and Aldor, thus handling a wide spectrum of parametric polymorphism and binding time semantics. We are not aware of any current mainstream languages that would pose substantially new issues. For example, it would be straightforward to provide C# bindings once its template support is finalized.

Our generic model supports a type of bounded polymorphism, in which restrictions can be placed on type variables in terms of both inheritance relations (*extension-based qualification*), and expected functionality (*export-based qualification*). The distinction between the two types of qualification is the standard one between named and structural subtyping. The latter will provide a natural mapping for programming languages that allow type variables to be bounded by a list of exports, and will be useful in cleanly describing the semantics of orthogonally designed libraries (see C++'s STL, Section 7.1), or in mapping functional types to GIDL (see Section 7.2). Both qualifications are implemented in an uniform manner over the targeted languages (C++, Java and Aldor), with almost no run-time overhead introduced by the generics mechanism.

For our implementation of the generic model, we chose an erasure technique (as in **Java/GJ** [25, 24]), rather than syntax expansion (as in **C++**) or type-valued parameters (as in **Aldor** [35]). The generic type information is "erased" to types that are understood by the underlying component architecture, our mapping to the targeted languages (Java, C++, Aldor) being responsible for recovering the lost (generic type) information. The application of the type erasure technique [25] for implementing generics has allowed us to design a *generic software component architecture extension* that can work on top of most component architectures in use today (different CORBA implementations, DCOM, JNI), modulo modifications in the targeted language stub code generation phase. In addition, this design enforces the backward compatibility with the non-generic applications written for the underlying component architecture and with applications written in programming languages with no support for parametric polymorphism. GIDL is also comprehensive with respect to binding times, requiring the particular language binding (C++, Java, Aldor) to determine appropriate implementations (see Section 6).

2.2 Parametric Polymorphism

Theoretical work on the type theory and semantics of parametric polymorphism goes back at least to the work of Girard in 1972 on the context of proof theory in logic [10]. He was the first to formulate the now well known *System F* type system that supports universal types. A little later, in 1974, Reynolds developed independently a type system with essentially the same power and named it *polymorphic lambda calculus* [29, 30]. The ML-style of *let polymorphism* was first introduced by Milner in 1978 [19]. It constitutes a less general form of parametric polymorphism then that of *System F*, but has other advantages such as it effectively employs type reconstruction to compute the *principal type*: the most general type possible for every expression and declaration [6]. This problem is known to be undecidable for *System F*.

Bounded quantification is a means to type-check functions that operate uniformly over all subtypes of a given type. The first language to support a type-safe bounded quantification appears to have been CLU [16](1981). Cardelli and Wegner formulated in 1985 $SystemF_{<:}$ [2] that combines polymorphism and subtyping, increasing both the expressive power of the system and its complexity. Extensions of the $SystemF_{<:}$ relax the context scope rules: *F-bounded quantification* [1] allows the type variable to appear in its bound (as in $\lambda X <: \{a\colon Bool, b\colon X\}\,.\,t$). The Generic Java type system [25] permits mutual recursion between type variables via their upper bounds. Parametric polymorphism is also a common feature of higher-order systems with dependent types; these issues were explored early on in the context of the Russell language [8].

There are now quite a few popular programming languages with support for parametric polymorphism, albeit with differing semantics. The remainder of this section briefly reviews a few of them.

In **Ada** [15], a generic subprogram or package is defined by a generic declaration, containing a generic part (which may include the definition of generic formal parameters) and by a generic subprogram/package. Generic type definitions may be array, access or private type definitions. Within the specification and body of the generic program unit, the operations available on values of a generic formal type are those associated with the corresponding generic type definition, together with any given by generic formal subprograms. That is, when a template is established (instantiated), all names occurring within it must be identified in the context of the generic declaration.

In **Modula 3** [21], generics are confined to the module level: generic procedures and types do not exist in isolation. A generic module is a template in which some of the imported interfaces are regarded as formal parameters, to be bound to actual interfaces via type instantiation. In Modula 3 there is no separate type-checking associated with generics, but instead, the implementations will expand the generics and type-check the result. This is a non-homogeneous approach: the source code is reused, but the compiled code is different for different instances.

The templates in **C++**, much like in Modula 3, are expanded at the compile time of the client; the same template call may or may not generate a compile time error, depending on the instantiation and on the context. However, the generic parameters can be substituted by any C++ type —it is not confined to be a class type as in Modula 3.

Java version 1.5 introduces a generic type mechanism inspired by the **Generic Java** (**GJ**) extension. As the Java Virtual Machine has no support for parametric polymorphism, the GJ extension [24] needs to compile away polymorphism through translation strategies. The **GJ** [25] type system is based on a combination of Hindley-Milner type inference [6], F-bounded quantification[1] and type-classes [13]. It uses a homogeneous implementation approach based on a type erasure mechanism that preserves both backward (through row types) and forward (through retrofitting) compatibility. The main drawback of the approach is that some operations involving type variables are forbidden: one cannot apply the `new` operator on a generic type or on an array type whose signature involves a generic type. Also, the Java objects carry at run-time only the *erased* type information, the reflective mechanisms may be used. Other proposed extensions for parametric polymorphism in Java (e.g. NextGen [3]) preserve the run-time information of the type variables and impose fewer restrictions than GJ, but feature a weaker compatibility with legacy code.

C# [36] semantics for generics are similar to those of Java. The implementation, however, is not through an *erasure* technique. Instead, the .NET 2.0 Beta Common Language Runtime (CLR) provides support for bounded parametric polymorphism. To implement its F-bounded quantification, the CLR uses a combination of a homogeneous approach (for reference type instantiations) and macro-expansion (for basic type instantiations).

Other programming languages provide parametric polymorphism through higher order functions. **ML** provides functors [17] operating on module structures as its form of parametric polymorphism. In addition, the supported *let polymorphism* [19] allows a single part of a ML program to be used with different types in nested *let* scopes.

Aldor [33, 34, 35] is a functional language, with a higher order type system with dependent types and type categories. Aldor has been used in the area of Computer Algebra, where an expressive type system is required to capture the relationships among abstract mathematical objects. Parametric polymorphism is provided by type-producing functions that accept and produce types belonging to declared type categories at run time. In Aldor, type variables may be qualified by means of named category-subtyping, or by means of a list of exports.

We note that programming languages with separate compilation for generic modules and dynamic binding time (Java, Aldor) usually provide support for parametric polymorphism with qualification. This allows them to statically type-check the generic code. We also note that they also usually employ a homogeneous approach to implementation. Other programming languages (C++ and Modula 3) rely on their static binding time to implement parametric polymorphism. In these cases, the type-checking has to wait until the generic type is instantiated, thus the implementation approach is usually a non-homogeneous one.

We conclude that a mechanism to combine modules in different programming languages must be able to accommodate both *compile-time* and *run-time* instantiation of modules and both *qualified* and *unqualified* type variables. Our approach has been to design a qualification-based generic type model to accommodate programming languages that support it, and to enforce these qualifications in our mappings for the programming language which do not. Our model also allows generic types to be unqualified, in which case any GIDL type is a valid candidate for instantiation.

2.3 Software Component Architectures

Among the mainstream software component architectures today, we note CORBA, DCOM, JNI and the more recent .NET architecture. This section briefly introduces these technologies, with the exception of CORBA, which we describe in Section 5.1.

The Distributed Component Object Model (DCOM) is a set of Microsoft concepts and program interfaces in which client program objects can request services from server program objects written in various languages on networked computers. Every COM object has an associated interface, written in an intermediate language. The only way to access a COM object is through its interface, which once published, is immutable. This is similar to CORBA, which uses IDL to describe the component interfaces.

The *Java Native Interface* [31] (JNI) is a native programming interface that allows Java code running inside a JVM to interoperate with applications and libraries written in C, C++ and assembly code. JNI imposes no restriction on the implementation of the underlying JVM: a native application should work with all JVMs supporting JNI. A Java class that contains some *native* methods, when compiled under JNI with the `javah` utility, produces a C/C++ stub header file, containing the signature of its native methods. The JNI defines mapping types for basic types (`jint, jfloat,` ...), reference types (`jobect`), plus its specializations for array types, `Class`, `String`, and `Throwable` (`jarray`, `jintArray`, *etc*, `jclass`, `jstring`, `jthrowable`). The correctness of an invocation in either direction is the programmer's responsibility and an incorrect invocation may result in arbitrary undefined behavior or in a JVM exception. JNI does not support parameterized components: all parameterization is erased and is invisible to the JVM.

The Common Language Runtime (CLR), introduced by Microsoft in its .NET framework, aims to provide a common type system and intermediate language (CIL) to facilitate interoperability between programs written in several programming languages [14]. Starting with version 2.0 Beta, .NET's CIL provides support for bounded parametric polymorphism, as proposed by Kennedy and Syme [14]. It is not clear to us if the generics semantics are unified across .NET's supported languages. (What will happen if one tries to instantiate a C# parameterized class making use of bounded type parameters on some invalid types from within *C++*?) The approach of this paper is at a higher level than .NET, in the sense that it does not require a homogeneous multi-language environment. That is, it does not require a common intermediate language to which all the supported languages must compile.

Our mechanism for generics can be adapted in a straightforward way to add genericity to various software component architectures, as discussed in Section 5, while preserving the backward compatibility with non-generic applications. This is a direct consequence of the type-erasure mechanism that implements our generic model.

We are unaware of other effort, besides ours, aiming at endowing software component architectures with parametric polymorphism features in a non-homogeneous environment.

3. GENERIC IDL

CORBA–IDL [26] is a declarative language used to describe the interfaces that client objects call and object implementations provide, separating the specification and the implementation aspects of a module. It defines basic types (`short`, `byte`, `float`, `double`, `string`, ...), structured types (`struct`, `sequence`, `array`) and provides signatures for interface types, fully specifying each operation's parameters. Thus, a specification written in CORBA–IDL encapsulates the information needed in order to develop clients using the specified services. These services may be provided by local or remote objects and are in principle transparent to the client program. CORBA–IDL's usefulness for language-independent specification has lead to its use outside of its initial CORBA setting. For example, the World Wide Web Consortium provides IDL definitions for its document object models for XML, SVG, MathML, etc.

In this section we present the syntax and semantics of Generic Interface Definition Language (GIDL), our extension to CORBA–IDL that supports parametric polymorphism. We have developed a corresponding GIDL compiler, consisting of about 33,500 lines of code in 133 Java classes.

We emphasize from the outset that we do not aim at writing a compliant OMG–CORBA extension; for example we have not as yet modified the CORBA interface repository to handle generic types. We have focused on adding parametric polymorphism at the static IDL level of CORBA so the ideas involved in our design can be applied in a straightforward manner to extend other software component architectures. Reflective features and type repositories are architecture specific and thus not the subject of this paper. However, these type (interface) repositories mirror the IDL specification and therefore similar ideas can be employed to enhance them with support for parametric polymorphism.

3.1 Rationale of the Design

We summarize the main principles that guided the design of our GIDL extension. We required that the GIDL's model for generics should:

- be general enough to allow similar extension for various software component architectures, and preserve the backward compatibility with non-generic applications
- have the property that the type of an expression be context independent (i.e. be determined solely by the type of its constituents),

- be powerful enough to make specifications written in GIDL clear, precise and easily extensible, allowing qualifications to be placed on generic types,
- allow mappings to languages supporting parametric polymorphism in a natural way, within a small overhead cost.

In the light of the above objectives we have constructed a generic model for GIDL in some ways similar to that of Java (GJ [32]). We use a homogeneous implementation approach, based on a type erasure technique which ensures the backward compatibility with the non-generic applications written for the underlying software component architecture. Briefly, the GIDL compiler generates an IDL specification file by erasing the generic type information, and generates wrapper code in the desired programming language (C++, Java, Aldor) to retrieve the erased information.

3.2 Parametric Polymorphism Semantics

GIDL defines a generalized model of parametric polymorphism that allows us to support a range of languages through various mappings. One consequence is that GIDL is neutral to whether the type parameters are created statically or dynamically; this depends on the targeted language. From a type-system point of view, GIDL supports F-bounded quantifications [1] based on named and structural subtyping. Type variables can be restricted to explicitly extend a given interface, or to implicitly implement all the functionality (methods) of a given interface. The latter addresses the code extensibility and re-usability issue, allowing the programmer to design a clean and precise specification, and to avoid unnatural inheritance relations between interfaces. (This is useful, for example, in rendering the correct semantics of orthogonal-based libraries as the C++ STL.) Furthermore, there are languages like Aldor that can allow type variables to be bounded simply by a list of exports, without demanding a subclassing relationship. This type of restriction is discussed further in Section 4.

The following example introduces the varieties of parametric polymorphism supported by GIDL. Suppose we want to write a very simple GIDL interface describing a priority queue, as in Figure 1.

The interface `PriorQueue1` specifies a priority queue of objects whose types have to be the `PriorElem` interface or to explicitly extend it (be a subtype of it). We call this an *extension-based qualification.* A type instantiation of an *extension-based qualified* generic type will is valid only if it actually inherits from the qualifier, in our case `PriorElem`.

The `PriorQueue2` interface accepts as valid candidates for the generic type all the interfaces that implicitly, fully implement all the operations present in the definition of the `PriorElem` interface. We call this an *export-based qualification.* Note that this definition requires exact matching of method signatures, and does not accommodate functional subtyping (contravariant parameter types, covariant return type).

To illustrate, at line 27 in our example, the type checker will accept the `Test<Foo_extend, Foo_export>` scoped-name, because the interface `Foo_extend` inherits from `PriorElem`, and the `Foo_export` interface implements the whole functionality of the `PriorElem` interface. Line 28 will generate a type error since `Foo_export` does not inherit from `PriorElem`, and therefore violates the extension based qualification of the `A: PriorElem` generic type.

```
module GenericStructures {
  interface PriorElem {
    short getPriority();
    short compareTo(in Object r);
  };

  interface Foo_extend : PriorElem { /* ..... */ };
  interface Foo_export{
    short getPriority();
    short compareTo(in Object r);
    //... Assume Foo_export is not in a isA
    //    logical relation with PriorElem so
    //    we did not want to extend from it
  };

  interface PriorQueue1<A: PriorElem> {
    void    enqueue(in A a);   A       dequeue();
    boolean empty();           short   size();
  };
  interface PriorQueue2<A:-PriorElem> {
    void    enqueue(in A a);   A       dequeue();
    boolean empty();           short   size();
  };

  interface Test<A: PriorElem, B:- PriorElem>{
    Test<Foo_extend,Foo_export> op1();//line 27 - OK
    Test<Foo_export,Foo_export> op2();//line 28 - ERROR
    Test<Foo_extend,Foo_extend> op2();//line 29 - OK
  };
  // ...
};
```

Figure 1: Different generic type qualifications

A type instantiation of an *export-based qualified* generic type is valid only if it is found to implement the whole qualifier's functionality. In this example, a call such as `PriorQueue2<Interf>` is valid only if `Interf` contains the operations

```
  short getPriority()
  short compareTo(in Object r)
```

This check is not trivial, as shown below:

```
  interface Elem {
     Elem op(in string str, in Object o);
  };
  interface TElem<A, B> {
     A op(in B b, in Object o);
  };
  interface Test<A:-Elem>{  };
```

Both `Elem` and `TElem<Elem, string>` are valid candidates for the generic type `A` in the definition of the `Test` interface, but this is not true for `TElem<Object, string>` for example, because `Object` is not a subtype of `Elem` and `op` is required to return an `Elem`.

GIDL also supports *unqualified* generic types, similar to templates in C++ (e.g. `PriorityQueue3<A>`). This allows the instantiation to be any GIDL type. GIDL also supports type parameterized methods, common to all three mapped languages (e.g. as inner template function). The GIDL-level type checking and the language bindings necessary to implement this feature are similar to those for parametric polymorphism at the interface type level. However, a delicate problem arises when ensuring the correct invocation of such a method. Due to their static implementation of parametric

```
//...

<forward_dcl> ::= ["abstract"] "interface" identifier
                    ["<" <template_dcl> ">"] ;

<template_dcl> ::= <template_dcl_unit>
         | <template_dcl> "," <template_dcl_unit> ;

<template_dcl_unit> ::= <identifier> [{":"|":-"}
                            <scoped_name>] ;

<template_call> ::= <template_call_unit>
         |   <template_call> "," <template_call_unit> ;

<template_call_unit> ::= <const_type> ;

<scoped_name> ::= ["::"] <identifier>
                    ["<" <template_call> ">"]
         |        <scoped_name> "::" <identifier>
                    ["<" <template_call> ">"] ;

//...
```

Figure 2: Modifications to the IDL grammar

polymorphism, both C++ and Java expect the method-level generics to be instantiated at the call site. In our case, the code is split between the caller and callee and separately compiled, thus the server has no way of knowing the type parameter instantiations. To handle this, we pass extra reflective-parameters that encapsulate the type-information of the generic type instantiations. The server-side generates code for a small method, which invokes the parameterized method on properly instantiated type-parameters, *just-in-time* compiles it and *links* it to the application. The *generated method* is finally called to complete the *original* invocation. The *generated methods* corresponding to different instantiations of the exposed type parameters can be cached for later reuse. A similar mechanism can be found in [23].

3.3 Grammar and Consistency Checks

To provide syntax for parametric forms, we have modified the OMG IDL grammar as shown in Figure 2. Specifically, we have modified the derivation rule for the `scoped_name` nonterminal so that we can manipulate template types inside the GIDL specification (we can have sequence, arrays, structures, unions, interfaces, regular-values, etc. making use of generic types).

We discuss a few details, with examples referring to the GIDL specification in Figure 3. We define the visibility scope of a generic type parameter to be throughout the interface in which it is defined. Following the same approach as in Generic Java [25, 32], we consider the subtyping to be invariant for parameterized types. For example, even if `Elem` is a subtype of `Object`, `Comp<Elem>` is not a subtype of `Comp<Object>`. In Figure 3, the type-checking of the `Comparator<Comp<B>, Comp<A>>` type (with mutual-recursive bounds) shall fail. This is because `Comp<B>` should extend `Comp<Comp<A>>` and, since the subtyping is invariant for parameterized types, this implies that `B` and `Comp<A>` are precisely the same type, which is not true. Using a similar reasoning, one will find that the `Comparator< Double, Float>` type is well-formed.

```
interface Base<C> {
  typedef struct BaseStruct {
    Base<C> field;
  };
};

interface Comp<A> : Base<A>{
  void op1(in BaseStruct s);
};

interface Double : Comp<Float> {...};
interface Float : Comp<Double> {...};

interface Comparator<A : Comp<B>, B : Comp<A> > {
  Base<B>::BaseStruct op2();

  Comparator<Comp<B>, Comp<A> > op3(); //Error
  Comparator<Double, Float> op4();     //OK
};
```

Figure 3: Scopes and type-checking

We turn now to the validity of the `op1`/`op2` operations of the `Comp`/`Comparator` interfaces. The `op1` method takes a parameter of type `BaseStruct`. The latter makes use of the generic type `C` and is defined inside the `Base` interface, which is a superclass of `Comp`. It follows that `BaseStruct` is also in the scope of `Comp`, its signature in this context, determined by up-traversing the inheritance tree of `Comp`, being `Base<A>::BaseStruct`. In the case of the `op2` method, all the information is stored inside the `scoped_name` of the returned type: `Base<B>::BaseStruct`.

We explicitly note that the *extension-based qualification* is stronger that the *export-based qualification.* For example, the GIDL specification below should generate a compile error.

```
interface Test0<C:Type1> { ... };
interface Test1<A:-Type1> : Test0<A> { ... };
```

This is because the type variable `A` in the `Test1<A>` scoped name is not required to extend `Type1`, as requested by the `Test0` definition, but only to implicitly implement its functionality.

3.4 Well-Formedness Type Rules

This section discusses the issues that arise from the combination of both named and structural subtyping in the definition of the qualification semantics. Figure 4 shows some of the type rules for well-formedness and subtyping in the presence of qualified type variables. We do not discuss the *unqualified* generic type, as its formal integration does not pose any challenges.

In this discussion, the metavariable X ranges over type variables; T, R and P range over types; N and O range over types other than type variables (non-variable types). I and m range over interface and method names respectively, while M ranges over method signatures. We write $\overline{X}$ as a shorthand for X_1,...,X_n and $\overline{X}\overline{\triangleleft}\overline{N}$ as a shorthand for $X_1 \triangleleft_1 N_1, ..., X_n \triangleleft_n N_n$. The length of the sequence $\overline{X}$ is $\#\overline{X}$ and we assume that the sequences of type variables contain no duplicate names. An interface table IT is a mapping from interface names to interface declarations. A type environment Δ is a finite mapping from type variables to pairs of bounds and qualification relation, written $\overline{X}\overline{\triangleleft}\overline{N}$ where $\triangleleft_i$ is

(Well-formed types ":" and ":-" qualifications)

$$\frac{\begin{array}{l} IT(I) = interface\ \ I < \overline{X} \lhd \overline{N} >: \overline{O}\{...\} \quad \lhd_i \in \{\ :\ ,\ :\text{-}\ \} \\ \Delta \vdash \overline{T} \qquad\qquad \Delta \vdash T_i\ \ \triangledown_i\ \ [\overline{T}/\overline{X}]N_i \quad \forall i \in \{1, .., \#\overline{X}\} \\ where\ \triangledown_i =\ \ <:\ \ if\ \lhd_i =\ \ :\ \ and\ \triangledown_i =\ \ <:\text{-}\ \ if\ \lhd_i =\ \ :\text{-} \end{array}}{\Delta \vdash I < \overline{T} >}$$

(Named subtyping "<:")

$$\frac{IT(I) = interface\ \ I < \overline{X} \lhd\ \overline{N} >: \overline{O}\{...\}\ \ \lhd_i \in \{\ :\ ,\ :\text{-}\ \}}{\Delta \vdash I < \overline{T} > \quad <: \quad [\overline{T}/\overline{X}]O_i \qquad \forall i \in \{1, .., \#\overline{O}\}}$$

(Structural subtyping "<:-")

$$\frac{\begin{array}{l} Methods(O_1) = \{M_{11}, .., M_{1k}\}\ \ Methods(O_2) = \{M_{21}, .., M_{2\ell}\} \\ \text{where}\quad \ell \leq k \quad \Delta \vdash O_1\ \ \Delta \vdash O_2 \quad \Delta \vdash M_{2i} \preceq M_{1i}\ \ \forall i \in \{1, .., \ell\} \end{array}}{\Delta \vdash O_1 <:\text{-}\ \ O_2}$$

(Method inclusion "$\preceq$" – II)

$$\frac{\begin{array}{l} M_1 = R_1\ \ m(\overline{P_1}) \quad M_2 =< \overline{X} \lhd \overline{N} > R_2\ \ m(\overline{P_2}) \quad \lhd \in \{:\ ,\ :\text{-}\} \\ \exists \overline{T} \quad \Delta \vdash \overline{T} \qquad\quad \Delta \vdash \overline{P_1} = [\overline{T}/\overline{X}]\overline{P_2} \qquad\quad \Delta \vdash R_1 = [\overline{T}/\overline{X}]R_2 \end{array}}{\Delta \vdash M_1 \ \ \preceq\ \ M_2}$$

(Method inclusion "$\preceq$" – III)

$$\frac{\begin{array}{l} M_1 =< \overline{X_1} \lhd_1 \overline{N_1} > R_1\ \ m(\overline{P_1}) \quad M_2 =< \overline{X_2} \lhd_2 \overline{N_2} > R_2\ \ m(\overline{P_2}) \\ \Delta \vdash \overline{P_1} = [\overline{X_1}/\overline{X_2}]\overline{P_2} \qquad\qquad\qquad\qquad \Delta \vdash R_1 = [\overline{X_1}/\overline{X_2}]R_2 \\ \lhd_1, \lhd_2 \in \{\ :\ ,\ :\text{-}\ \} \qquad\qquad \Delta \vdash \overline{N_1}\ \ \overline{\psi}(\lhd_1, \lhd_2)\ \ [\overline{X_1}/\overline{X_2}]\overline{N_2} \end{array}}{\Delta \vdash M_1 \ \ \preceq\ \ M_2}$$

$$\psi(\lhd_1, \lhd_2) = \begin{cases} \lhd_1 = \lhd_2 = \ : & then\ \ : \\ \lhd_1 = \lhd_2 = \ :\text{-} & then\ \ :\text{-} \\ \lhd_1 = \ :\ \ and\ \ \lhd_2 = \ :\text{-} & then\ \ :\text{-} \\ \lhd_1 = \ :\text{-}\ \ and\ \ \lhd_2 = \ : & then\ \ \eta\ \ where \\ \quad O_1 \eta O_2 = true\ \ if\ \ \{I|I <:\text{-}O_1\} \subseteq \{I|I <: O_2\}, \\ \quad \text{and}\ \ false\ \ \text{otherwise} \end{cases}$$

Figure 4: Type rules for two varieties of qualification

one of the *extend* or *export* based qualifications. For brevity, some obvious rules are omitted from Figure 4: A type variable X is well formed in the type context Δ if it belongs to the domain of Δ. The type $Object$ (the root of the IDL inheritance hierarchy) is well formed in any type context. Both subtyping relations are reflexive and transitive. Also, a type variable belonging to a type context is known to be in the corresponding subtyping relation with its bound.

The well-formedness rule in Figure 4 simply says that if the declaration of interface I begins with $interface I < \overline{X} \lhd \overline{N} >$, then a type $I < \overline{T} >$ is well formed only if all the components of $\overline{T}$ are well formed and if, in addition, substituting $\overline{T}$ for $\overline{X}$ respects the bounds $\overline{N}$. Also, note that the simultaneous substitution enables recursion and mutual recursion between variables and bounds [11]. The *named subtyping* rule ("<:") in Figure 4 is also straight-forward: the inheritance hierarchy is dictated by the interface table IT.

Intuitively, the type-rule for structural subtyping ("<:-") says that O_1 is a structural subtype of O_2 if "it exports all the methods" of O_2. (IDL attributes are seen as a pair of methods: a getter and a setter). Note that O_1 and O_2 are instantiated types, in a given type context Δ. To formalize this property we introduced the *inclusion relation* ("$\preceq$") between methods. If M_1 and M_2 are not type parameterized then $M_1 \preceq M_2$ if the method names and signatures are identical. It follows in this case that also $M_2 \preceq M_1$.

Type-parameterized functions can be viewed as a set of functions: one for each different instantiation of their generic types. If M_2 is type parameterized ($\overline{X} \lhd \overline{N}$), but M_1 is not, then $M_1 \preceq M_2$ if the method names are identical and there exist a set of well-formed types $\overline{T}$ such that the substitution/instantiation $[\overline{T}/\overline{X}]$ applied on M_2 yields a signature identical with that of M_1. The last case is when both M_1 and M_2 are type parameterized. Let us assume only one type parameter for M_1 and M_2: X_1 and X_2 respectively. (The generalization is straight-forward.) In order to have $M_1 \preceq M_2$ we need to have that the set of valid instantiation for X_1 is included in the set of valid instantiations for X_2. Assume an *extend-based qualification* $X_1 : O_1$ for X_1 and an *export-based qualification* $X_2 :\text{-} O_2$ for X_2. The set of interfaces that extend O_1 should be included in the set of interfaces that *implement* O_2 and the necessary and sufficient condition is $O_1 :\text{-} O_2$. A similar line of reasoning leads to the definition of the ψ operator in Figure 4. The last case leads to an overly technical result, which requires the type-checker to work hard. We prefer the more elegant alternative that excludes this case: if $X_1 :\text{-} O_1$ and $X_2 : O_2$ then M_1 is not $\preceq$-included in M_2.

3.5 GIDL to IDL Transformation

The implementation of our generic model employs a type erasure mechanism, based on the subtyping polymorphism supported by IDL. This preserves the interoperability between programs written over different implementations of the same software component architecture and allows our model to be easily adapted to enhance several software component architectures.

To achieve this, we constructed a translator from our GIDL to OMG IDL, accepting both regular IDL and GIDL specifications. When generating the IDL file, we first delete the generic type declarations from the GIDL file (delete the `template_dcl` productions in the GIDL grammar). Then the *unqualified/export-based qualified* type variables are substituted by the `any`/`Object` IDL type, while the *extend-based-qualified* ones are substituted by the (type variable erased) interface type they are supposed to extend. The result should be a valid OMG IDL file, which can be compiled with a regular IDL compiler.

It is obvious that during this transformation we are losing the generic type information encapsulated in the GIDL specification. We recover this information by generating skeleton/stub wrapper classes in the target languages that make use of the specific characteristics of the parametric polymorphism in these languages. If we run the GIDL translator over the specification shown in Figure 1, it will generate the IDL specification in Figure 5.

```
module GenericStructures {
  // ...

  interface PriorElem {
    short getPriority();
    short compareTo(in Object r);
  };

  interface PriorQueue1 {
    void enqueue(in PriorElem a);
    PriorElem dequeue();
    boolean empty();
    short size();
  };

  interface PriorQueue2 {
    void enqueue(in Object a);
    Object dequeue();
    boolean empty();
    short size();
  };

  // ...
};
```

Figure 5: The generated IDL specification

4. HIGH LEVEL IDEAS FOR MAPPING QUALIFIED GENERIC TYPES

Our generic type mechanism unifies the semantics of parametric polymorphism from different programming languages. In the implementation of our software tools we do as much work as possible at the unified level and in the GIDL to IDL translation, to minimize the language specific details.

4.1 Basic Ideas

Type-erasure for an *extend-based qualified* generic type is achieved by substituting it with the bounding-interface specified in the corresponding `template_dcl_unit` production. The Java and Aldor mappings are quite natural since this type of qualification is already supported. For the C++ language, due to its static binding time, the mapping can be achieved simply by casting an instance of the generic type to its corresponding qualifier. Note that this code is never executed at run-time, as shown in line marked "`//*`" in Figure 6.

We show below that the *export-based qualified* generic type can be reduced to an *extend-based qualification* relation at the GIDL level. The idea here is to find, for each *export-based qualified* generic type, all the possible interfaces that may implement the functionality of the associated qualifier.

The next step is to construct an interface that:

- implements the whole functionality of the qualifier (for a proper instantiation of its generic types, if any),
- becomes a natural parent for the interfaces identified in the previous step (in the sense that the inheritance does not actually introduce new functionality),
- defines a minimal number of generic types

We call the constructed interface the *most specific generic antiunifier* (*MSGA*) of the export-based qualification. The MSGA can be seen as the most specific antiunifier [28] or equivalently the least general generalization [27] of the types that satisfy the *export-based qualifier.*

```
// GIDL specification:
interface Foo { /*...*/ };
interface Test<T1:Foo> { /*...*/ };

- - - - - - - - - - - - - - - - - - - - - - - - - - - -

// C++ mapping:
template<class T1> class Test :
virtual public ::GIDL::GIDL_Object {
  private:

    virtual void implTestFunction() {
      if(1) return;                  //*
      T1 a_T1; Foo a_Foo = (Foo)a_T1;
    }

  public:

    Test(::Test_var ob) {
      implTestFunction(); //...
    }//...
}
```

Figure 6: *Extend-based qualification* mapping to C++

```
interface Element {
  tp0 op(in tp1 a1, in tp2 a2, in tp0 a3,
         in tp3 a4, in tp1 a5);
};

interface TemplEl1<T1, T2> {
  T1  op(in T2 a1, in tp2 a2, in T1 a3,
         in tp3 a4, in T2 a5);
};

interface TemplEl2<T1, T2, T3> {
  T1  op(in tp1 a1, in T2 a2, in T1 a3,
         in tp3 a4, in T3 a5);
};

interface Test<A:-Element> {
  //use A
}
```

Figure 7: MSGA Example

Section 3.4 has already introduced and explained the GIDL type rules related to well-formedness and subtyping in the presence of qualified type variables. Next we discuss the main stages involved in the *MSGA* construction and we present an example. This *MSGA* could be used as the erasure type for its corresponding generic type. We have chosen not to do so, however, due to CORBA's IDL limitations and we use `Object` instead.

4.2 Mapping *Export-Based Qualification*

The algorithm for computing the *MSGA* associated with an *export-based qualification*, presented here, works under the assumption that the *extend-based qualification* has already been mapped to the target language. Each GIDL-interface that may satisfy the *export-based qualification* in certain circumstances (for a given instantiation of the generic type for example), shall be made to implement the *most specific generic antiunifier* (*MSGA*) interface associated with that *export-based qualification.*

```
interface MSGA<G0, G1, G2, G5> {
  G0 op( in G1 a1, in G2 a2,
         in G0 a3, in tp3 a4, in G5 a5 );
}

interface Element: MSGA<tp0,tp1,tp2,tp1> {...};

interface TemplEl1<T1, T2>: MSGA<T1,T2,tp2,T2> {...};

interface TemplEl2<T1, T2, T3>: MSGA<T1,tp1,T2,T3>{...};

interface Test<A : MSGA<tp0, tp1, tp2, tp1> >
{ //use A...};
```

Figure 8: The result of the MSGA Algorithm

As an example, consider the GIDL file shown in Figure 7. The `Test` interface uses an *export-based qualified* generic type. Among the valid candidates for the type instantiation one can list `Element, TemplEl1<tp0, tp1>, TemplEl2<tp0, tp2, tp1>`. Being given the methods in the `Element` interface and the set of interfaces defined in a GIDL specification, our task is to construct the most specific generic antiunifier (*MSGA*) of these candidates. First, we construct a new parameterized interface, with as many generic types as the number of parameters in all the methods of the "to be implemented" interface, plus the number of methods, as the return types should also be taken into account. In our example, the *MSGA* initially looks like:

```
interface MSGA<G0, G1, G2, G3, G4, G5> {
  G0 op( in G1 a1, in G2 a2,
         in G3 a3, in G4 a4, in G5 a5 );
}
```

Left like this, the interface created can make use of many different generic types, so we may want to simplify it. We create a matrix as below, in which the types that have to match will share the same column. If there is an interface that we can prove cannot implement the required functionality, it should not appear in the matrix.

G0	G1	G2	G3	G4	G5	MSGA
tp0	tp1	tp2	tp0	tp3	tp1	Element
T1	T2	tp2	T1	tp3	T2	TemplEl1
T1	tp1	T2	T1	tp3	T3	TemplEl2

The first thing to do is to identify the columns formed by the same non-generic type. This occurs in `G4`'s column in the above table. The next step is to remove the corresponding generic type from the template declaration part of the *MSGA* interface and substitute it with the non-generic type throughout the *MSGA*'s interface definition. In our example this would be substituting `tp3` for `G4`. A second simplification can be made if two columns are found to be equal. This occurs with columns 0 and 3 of our example. In this case we can also remove one of the generic types in the template declaration part of the *MSGA* interface and substitute it with the other generic type throughout the interface definition. Special care should be taken for the `void` return type, since it cannot be matched by any generic type instantiation.

Finally, all the interfaces found to be valid candidates to instantiate the *export-based qualified* generic type, are made to implement the simplified *MSGA* interface, as shown in Figure 8.

```
//A. GIDL specification//

// Eg. 1
interface Type1<A:-Type1<A> > {...};
interface Type2<B:-Type2<B> > : Type1<B> {...};

// Eg. 2
interface Elem<C>{...};
interface Test1<D:-Elem<D> >{...};
interface Test2<E:-Elem<E> >{...};


//B. MSGA constructs for the GIDL specification in A.//

// Eg. 1
1. interface MSGA1<A>{...}; //A:-Type1<A>
2. interface MSGA2<B> : MSGA1<B>{...}; //B:-Type2<B>
3. interface Type1<A : MSGA1<A>> : MSGA1<A>{...};
4. interface Type2<B : MSGA2<B>> :
                 Type1<B>, MSGA2<B>{...}; //***

// Eg. 2
5. interface MSGA3<T>{...}; //D:-Elem<D> and E:-Elem<E>
6. interface Elem<C> : MSGA3<C>{...};
7. interface Test1<D : MSGA3<D>>{...};
8. interface Test2<E : MSGA3<E>>{...};
```

Figure 9: More MSGA Issues

It is clear that only `Element`, `TemplEl1<tp0, tp1>` and `TemplEl2<tp0, tp2, tp1>` will not be signaled with a compiler error when substituted for the generic type `A` in the `Test` generic interface. Notice also that the *MSGA* is using only *unqualified* type parameters in order to cover all possible type instantiations and that the generic type qualifications of the candidate interfaces (`TemplEl1`, `TemplEl2`) do not influence the algorithm in any way.

Type parameterized functions are accommodated in a straightforward manner in the algorithm presented. Section 3.4 has provided the details: If at least one type instantiation of a function satisfies the signature of another function that appears in the export-based qualifier, then we consider that the type parameterized function satisfies the qualifier's function. Conversely, if the export-based qualifier exports a type parameterized function, then only another type parameterized function will satisfy it and only if its set of valid type instantiations includes the one of the qualifier's function.

There are two additional points to mention with respect to MSGAs. Figure 9 presents a legal GIDL specification, together with its corresponding MSGA bindings. The first example in Figure 9 shows that we must preserve the inheritance hierarchy among MSGAs. If this were not done, the compiler would find an error while checking the correctness of the `Type1<B>` type in line 4. The `B` bound is `MSGA2<B>`, but `B` should also be bounded by `MSGA1<B>` from the definition of `Type1` in line 3. If no inheritance relation were defined among `MSGA2` and `MSGA1` interfaces, a compile-time error would be signaled.

In order to keep the number of generated MSGAs to a minimum, a simple unification algorithm is employed among *export-based qualification* relations. The second example in Figure 9 shows that only one MSGA (`MSGA3`) is constructed for the `D` and `E` *export-based qualifications* (lines 6,7).

```
template<class T1> class Test :
        virtual public ::GIDL::GIDL_Object {

  private:
    virtual void implTestFunction() {
       if(1) return;
       T1 a_T1; tp1 var1; tp2 var2; tp3 var3; tp0 var0;
       var0 = a_T1.op(var1, var2, var0, var3, var1);
    }

  public:
    Test(::Test_var ob) {
       implTestFunction();
       // ...
    }

  // ...
}
```

Figure 10: Incorrect C++ mapping

4.3 Discussion

We should remember that during the GIDL to IDL translation, we are loosing the generic information present in our GIDL specification. We recover the lost information by generating wrapper classes corresponding to the constructs in the GIDL specification. We use the *MSGA* interfaces as a general approach for mapping the *export-based qualification* to both the C++ and Java programming languages (once we have implemented the mapping for the *extend-based qualifications*). They are also used at the GIDL level for type checking. One can notice that the *MSGA* construct introduces little runtime overhead, as it is used in the type-checking phase at compile time. The generated *MSGA* interfaces may look ugly, involving many generic type parameters. This is not a concern since their use is transparent to the user, as their only task is to ensure a correct translation of the GIDL semantics to the language bindings.

At a first sight, it might appear that in the case of the C++ mapping, an easier solution can be found for translating the *export-based qualification*. Namely, the `A:-Element` in Figure 7, can be **wrongly** mapped as in Figure 10.

This calls all the qualifier's functions on the generic type object. This translation is not consistent with the *export-based qualification* semantics, however, as the generic type instantiation may export a method that is a subtype of the qualifier's method (in the usual functional lattice) and type-checking will succeed when it should not.

It might be desirable to have functional subtyping encapsulated in the *export-based qualification* semantics, especially since one could then easily map the semantics of functional subtyping through generics. In this case the code above will do. We are still investigating this but it appears to be difficult: If the generic model assumes invariant parameterized types subtyping, the current MSGA algorithm is insufficient. Conversely, if covariant and contravariant parameterized type subtyping is assumed, the MSGA algorithm can be made to work, but enforcing the correctness of the GIDL semantics in the mapped languages complicates the language bindings and the user interaction with the framework. This discussion is beyond the scope of this paper. At present, as described, our choice has been for invariant subtyping.

5. THE ARCHITECTURE OF THE GIDL BASE APPLICATION

We now present a high level view of our GIDL architecture: that is how the architecture components are created and how they interact to accomplish an invocation successfully. We then show how a programmer may use our architecture. We argue the transparency of our design, in the sense that the programmer need not know the internal architecture, but only the mapping rules from GIDL to a specific programming language.

5.1 The GIDL Extension Architecture

Figure 11 illustrates the design of our proposed architecture. The circles stand for user's code. The rectangular boxes represent components in the standard OMG-CORBA architecture. This includes the IDL specification, the stub and skeleton, and the object request broker (ORB). The hexagons represent the components needed by our generic extension, including the GIDL specification and generated GIDL wrappers. The dashed arrows represent the *compiles to* relation among components. A GIDL specification compiled with our GIDL compiler will generate an IDL specification file, together with GIDL wrapper stub and skeleton bindings, which recover the lost generic type information.

The bottom part of the figure represents CORBA's internals. When compiling the IDL file with any vendor's IDL compiler, client stubs and skeletons will be generated and these serve as proxies for clients and servers respectively. Because the IDL defines interfaces so strictly, the stub on the client side will have no trouble matching perfectly with the skeleton on the server side, even if the two are compiled to different programming languages, or are running on different ORBs from different vendors, under different operating systems or hardware [26].

The solid arrows in Figure 11 depict method invocation. In CORBA, every object has its own unique object reference. The client must obtain an object's reference in a string representation. This is used by the ORB to identify the exact instance that must be invoked. As far as the client is concerned, it is invoking a method on the object instance. However, it actually calls the IDL stub that acts as a proxy and forwards the invocation to the ORB. It is the ORB's job to find the server, to pass the parameters, make the invocation and eventually to return a result to the client [26].

As stated previously, our generic extension for CORBA introduces an extra level of indirection in the original mechanism; in order to recover the generic type information lost by the GIDL to IDL transformation, stub and skeleton wrappers are generated to match the original GIDL specification. Basically, for every type in our GIDL specification, we construct C++/Java/Aldor wrapper stubs that reference the CORBA-stub objects generated by the IDL compiler. When the client invokes an operation, it actually calls a method on a GIDL stub wrapper object. The GIDL method implementation retrieves the CORBA-objects hidden by the wrapper-objects taken as parameters, invokes the method on the CORBA-object's stub hidden inside our wrapper class, gets the result, encloses it in a newly formed wrapper if necessary and returns it to the client application. The wrapper skeleton functionality is the inverse of the client. The wrapper skeleton method encapsulates the erased IDL objects with generics erased as GIDL ones, adding back the generic type's erased information. It invokes the user-implemented server method

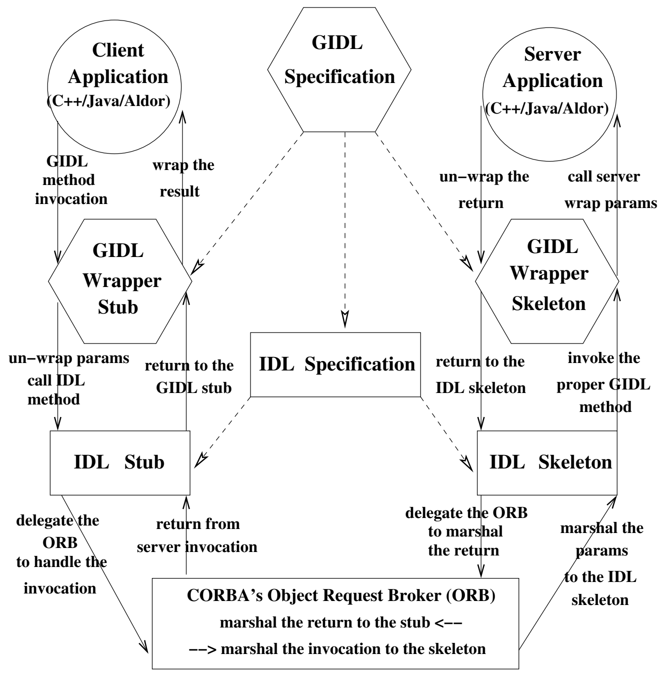
Client Application (C++/Java/Aldor)
GIDL Specification
Server Application (C++/Java/Aldor)
GIDL method invocation
wrap the result
un−wrap the return
call server wrap params
GIDL Wrapper Stub
GIDL Wrapper Skeleton
un−wrap params call IDL method
return to the GIDL stub
IDL Specification
return to the IDL skeleton
invoke the proper GIDL method
IDL Stub
IDL Skeleton
delegate the ORB to handle the invocation
return from server invocation
delegate the ORB to marshal the return
marshal the params to the IDL skeleton
CORBA's Object Request Broker (ORB)
marshal the return to the stub <−−
−−> marshal the invocation to the skeleton

Figure 11: GIDL **architecture for** CORBA
circle – user code
hexagon – GIDL *component*
rectangle – CORBA *component*
dashed arrow – is compiled to
solid arrow – method invocation flow

with these parameters, retrieves the CORBA IDL-object or value from the returned object and passes it to the IDL skeleton.

Clearly, for our implementation to be CORBA compliant, CORBA's *Interface Repository* (IR) model would have to be changed to handle parameterized interfaces. Two new IR–IDL interfaces for `TemplateDclUnit` and `TemplateCallUnit` extending the `IRObject` interface should be added to the IR meta model and the `InterfaceDef` IR-IDL interface should be modified to contain a sequence of `TemplateDclUnit` and a list of `TemplateCallUnit`. The definition of `ScopedName` would also have to be made to deal with templates. The `TypeCodes` and the string representation of references would also be extended to contain parameterized type information. But as we stated earlier, *it is not our goal to write a* CORBA *compliant extension of* IDL. Thus a detailed investigation of the changes that would have to be done at the CORBA interface repository level is orthogonal to the goal of this paper. We interested exclusively in adding genericity to IDL; the OMG's IR is a mirror of IDL's specifications, thus the same ideas apply.

A more delicate problem is modifying a software component architecture's runtime to deal with dynamic invocation on parameterized methods, as by the use of generics the number of GIDL types is potentially infinite. This is handled by run-time re-compilation techniques, similar to that described in [23]. We start with a set of candidates for the instantiation of the generic method and expand this set incrementally as invocations with different instantiation for generic types occur.

```
interface PriorElem {
   short getPriority();
   short compareTo(in Object r);
};

interface PriorQueue2<A:-PriorElem> {
   void    enqueue(in A a);   A       dequeue();
   boolean empty();           short   size();
   A creatNewA(in short s);
};
```

Figure 12: GIDL **for a simple priority queue**

With minimal modifications to the wrapper code generation, our generic extension architecture can sit on top of other software component architectures such as DCOM or JNI. Targeting DCOM is straight-forward, as its design is similar to CORBA.

Enhancing JNI is more subtle: Given a GIDL specification file, wrapper stubs are generated on the C++ and Java sides. These make use of parametric polymorphism and will ensure that the GIDL semantics are statically enforced in both mappings, similar to our design for CORBA. What differs is the implementation of the erased stub. On the C++ side, this corresponds to the mechanism provided by JNI to invoke the JVM; it can be mangled inside the wrapper classes and hidden from the user. To call Java code from C++, the C++ parameterized wrapper classes use the JNI mechanism to invoke, through JVM, the parameterized Java wrapper classes. To call C++ from Java, the parameterized Java wrapper classes, containing only native methods, are compiled and, as a result, the C++ generic erased stub is generated. The latter re-directs the invocation to the parameterized wrapper class.

In summary, the generic extension for our CORBA case study can be applied on top of any CORBA-vendor implementation, while maintaining backward compatibility with standard CORBA applications. Moreover, with minimal changes, our architecture can be applied to various heterogeneous systems. Our approach has been to design a general and clean extension architecture and then to apply aggressive optimization techniques to reduce the overheads incurred by casting, and the extra indirection in invocation. One can anticipate that a combination of optimizations, including pointer aliasing, scalar replacement of aggregates, copy propagation and dead code elimination, will achieve this in most cases.

5.2 The User's Perspective

Consider the GIDL specification shown in Figure 12. When implementing the server side, the programmer should extend the generated skeleton wrapper classes `PriorQueue2` and `PriorElem`, implementing the operations that appear in the GIDL specification. This is the usual CORBA procedure for writing servers, so the user will find no difficulty here.

A excerpt from a C++ client program that makes use of the types defined in this GIDL specification is shown in Figure 13. Suppose the server is represented by a `GIDL::PriorQueue2<GIDL::PriorElem>` object. The client obtains a string representation of a reference to the generic type erased object, i.e. `::PriorQueue2` from the server (line 1).

```
1. CORBA::Object_var obj = orb->string_to_object(s);
2. GIDL::PriorQueue2<GIDL::PriorElem> gpq(pq_orig);
3. GIDL::PriorElem gPEobj =
     gpq.createNewA(GIDL::Short_GIDL(1));
4. gpq.enqueue(gPEobj); // OK
   // Obtain a reference to the CORBA::Object
5. gpq.enqueue(obj);    // Error
6. PEobj = gpq.dequeue();
7. GIDL::Short_GIDL sh = PEobj.getPriority();
8. cout << sh << endl;  //prints "1"
```

Figure 13: Code excerpt from a C++ client

It creates a generic wrapper stub (line 2) together with an IDL stub proxy. The latter is implemented inside the wrapper class constructor to hide the internal architectural design. From this point on, the user can transparently invoke the server functionality (lines 4, 6, 7). In Figure 13, `GIDL::Short_GIDL` is the C++ mapping type for GIDL's `short`. Line 5 generates a compile-type error, signaling the user that his code does not obey the GIDL specification semantics. If we look at the GIDL specification in Figure 12, the `enqueue` operation is supposed to take a parameter of type `A`. In our case the parameter is substituted by `GIDL::PriorElem`, since we are working with `GIDL::PriorQueue2<GIDL::PriorElem>`. Therefore the parameter of the `enqueue` function is expected to be of type `GIDL::PriorElem` and not `CORBA::Object`.

To conclude, our architecture places little burden on programmer's shoulders, as most of our implementation details are hidden. The steps in application design are the same as those required for a standard CORBA application, but now the implementation can use generic programming. The details of the language bindings for C++, Java and Aldor are given in the next section.

6. LANGUAGE BINDINGS

In Section 5 we discussed in general terms the high-level ideas involved in the design of our framework. Now, by seeing the mapping specifics, we complete the description of the overall architecture. There are two reasons for presenting aspects related to the language bindings: First, as the targeted languages cover a wide-range of parametric polymorphism semantics and binding time models, it is informative to understand how the mappings work. Second, it is of practical interest to mention some of the less obvious details that are important in achieving an effective mapping.

We do not give a formal proof for the correctness of the translation schemes for the language bindings. This would be a tedious task as none of the targeted languages, to our knowledge, have a complete formal model. An approach similar to that adopted for *Featherweight GJ* [11], working with only a small functional subset of the languages considered, might be used to prove that our *extension* does not introduce any run-time errors.

6.1 GIDL to C++ Mapping

This section describes how the stub and skeleton wrappers, presented in the previous section as high level components of our architecture, are implemented when the targeted language is C++. We first introduce the high-level mapping ideas, including the correspondence between GIDL and C++ types. We then elaborate on the wrapper object-model and on the C++ implementation of GIDL's export- and extend-based qualifications.

```
// GIDL:
interface GenericInterf<A> {
   struct GenericStruct {
      typedef A A_array[5][5];
      A_array field;
   };
};

- - - - - - - - - - - - - - - - - - - - - - - - - - - -

// C++:
template<class A> class GenericInterf: GIDL::GIDL_Object{
  struct GenericStruct : GIDL::GIDL_Object {
    typedef Array_GIDL<...,A,...> A_array;
    public: A_array field; // ...
  }
  // ...
}
```

Figure 14: Nested structures

6.1.1 High-Level Mapping Ideas

The mapping from GIDL to C++ is for the most part quite easy and natural, as the IDL syntax and semantics are quite close to those of C++. We closely follow the same conventions used in the standard IDL to C++ mapping, so the user will not feel any major conceptual difference when using our generic architecture.

GIDL modules are translated into C++ namespaces; GIDL interfaces into C++ (possibly template) classes, encapsulating all the functions that appear in the GIDL interface together with getter and setter functions for every attribute in the GIDL interface. A GIDL structure is mapped to a C++ class, with setter and getter functions for each field in the GIDL structure. GIDL basic types (`short`, `long`, *etc*) are mapped to corresponding C++ types, providing the expected functionality by means of operator overloading. GIDL's arrays and sequences are mapped by type instantiating a C++ generic array/sequence class in which the "[]" operator is overloaded. In our implementation, the relation between the wrapper objects and the associated CORBA-objects is many to one: There can be several wrappers storing the same CORBA-object. Memory management is simple, creating our wrapper objects on the stack only. Thus there is no need for explicit de-allocation.

Our GIDL-C++ stub and skeleton wrappers are encapsulated within the "GIDL" and "GIDL_implem" namespaces. The inheritance hierarchy at the GIDL specification level is preserved in the C++ mapping. GIDL scopes directly create C++ scopes, as the C++ semantics allows the definition of nested classes. A side-effect of this is that the generic types defined by a generic GIDL interface stay in the same position after the C++ translation and do not create generic type duplicates for the nested GIDL structures (as happens in the Java mapping case).

In the example shown in Figure 14, a GIDL specification containing a structure type nested inside an interface type is similarly translated to C++ as a nested definition of classes. The generic type parameter `A` is shared inside the nested scope.

Our wrapper objects, no matter what GIDL type they represent, can be seen as an aggregation of a reference to the erased CORBA value they represent, the generic type information associated with them and the casting functionality

```
template<class T1,class T2,class T3>
class Test : virtual public ::GIDL::GIDL_Object
{
  protected:
    ::Test_var* obj;
  private:
    virtual void implTestFunction() {
      if(1) return;
      T2 a_T2; MSGA_Foo msga = (MSGA_Foo)a_T2;     //1
      T1 a_T1; Foo a_Foo = (Foo)a_T1;              //2
    }
  public:
    Test(::Test_var ob)
    { obj = new ::Test_var(ob); implTestFunction(); }
    //...
    ::Test_var getOrigObj() { return *obj; }
    void setOrigObj(::Test_var o)
    { *obj = ::Test::_duplicate(o); }

    static ::Test_var _narrow(
      Test<T1,T2,T3> corba_obj_TIDL)
    { return *corba_obj_TIDL.obj; }

    static Test<T1,T2,T3> _lift(
      CORBA::Object_var corba_obj_TIDL)
    {
       return (Test<T1,T2,T3>(
         ::Test::_narrow(corba_obj_TIDL)));
    }
    static Test<T1,T2,T3> _any_lift(CORBA::Any_var a)
    { /*...*/ }
    static CORBA::Any_var _any_narrow(Test<T1,T2,T3> w)
    { /*...*/}
    //...
    virtual GIDL::Foo op(T1 a1_G, T2 a2_G, T3 a3_G,
      GIDL::Foo a4_G)
    {
      ::Foo_var a1= a1_GIDL._narrow(a1_G);         //3
      CORBA::Object_var a2= a2_GIDL._narrow(a2_G); //4
      CORBA::Any_var a3= a3_GIDL._any_narrow(a3_G);//5
      ::Foo_var a4  a4_G._narrow(a4_G);            //6
      ::Foo_var a0_GIDL= (*obj)->op(a1,a2,a3,a4);  //7
      GIDL::Foo retGIDL;
      return retGIDL._lift(a0_GIDL);               //8
    }
};
```

Figure 15: Excerpt of C++ wrapper stub code

they define. They also inherit the functionality provided by the corresponding GIDL type. This is similar to the "reified type" pattern of Johnson [12], where objects are used to carry type information or to some uses of dependent product types.

6.1.2 Wrapper Stub Object Model

Figure 15 shows A piece of the generated wrapper stub for the following GIDL specification.

```
interface Foo { /*...*/ };
interface Test<T1:Foo, T2:-Foo, T3>
{ Foo op(in T1 t1, in T2 t2, in T3 t3, in Foo f); };
```

As stated in Section 6.1.1, the type casting functionality is common to all the stub wrapper types. This is represented in Figure 15 by the `_lift` and `_narrow` methods. The `_lift` method returns a new instance of the wrapper class encapsulating the CORBA-object received as parameter, while `_narrow` returns the CORBA-object encapsulated by the wrapper object. The `_any_lift` and `_any_narrow` functions have a similar functionality, but they are used in conjunction with the CORBA::Any type, our erasure for the *unqualified* generic type.

The implementation of the `op` function (in Figure 15) illustrates the method invocation mechanism. All the wrapper objects received as parameters are unboxed to IDL stub objects. Following the type erasure rules, a wrapper interface type object is unboxed to the CORBA stub object it encapsulates (line `//6`), a *unqualified* generic type is erased to the `CORBA::Any_var` type (line `//5`), an *extend-based qualified* generic type is unboxed to the IDL-stub type associated with its qualifier (line `//3`) and finally an *export-based qualified* generic type is erased to the `CORBA::Object` type (line `//4`). The IDL stub method is invoked on the object reference that our wrapper encapsulates (line marked 7) and finally the returned CORBA stub object/value is boxed inside a stub wrapper object and is returned to the client application (line `//8`).

The last thing to note here is the C++ mapping of GIDL's export- and extend-based qualifications for generic type parameters. We remind the reader that C++ does not support restrictions on generic type parameters. We achieve this through the `implTestFunction()` function (in Figure 15), which is called from the wrapper class constructors.

As discussed in Section 4.1, our implementation relies on C++'s static binding. In the case of the *extend-based qualified* generic type, a simple cast to the qualifier's type suffices (shown on line `//2`). This enforces the condition that the substituted type has to inherit from the qualifier (`GIDL::Foo` in our case). The mapping of an *export-based qualification* requires the construction of the *MSGA* associated with that generic type declaration, as discussed in Section 4. The generic type instantiation is valid if the cast to the associated *MSGA* succeeds (line `//1` in Figure 15). Otherwise a compile-time error is generated during type checking.

Our mapping of parameter qualifications adds no run-time overhead, as our verification code (lines `//1` and `//2`) follows the statement `if(1) return;` so is never reached. After the type-checking phase is completed, any reasonable compiler will discover this and all the calls to `implTestFunction()` will be eliminated.

6.2 GIDL to Java Mapping

The Java mapping follows the same main lines as the C++ mapping. We create wrappers objects that encapsulate CORBA object references and recover the generic type information lost during the GIDL to IDL erasure transformation. We follow the same translation rules defined in the standard IDL to Java mapping. The GIDL inheritance hierarchy is translated to a corresponding inheritance hierarchy among Java interfaces, the root of the hierarchy being the `GIDL_Value_Interf` interface. We do this because Java classes do not support multiple inheritance.

One drawback of the Java mapping is that it requires the user's help. Java does not support object instantiation of a generic type parameter, e.g. `new A()`. Neither does it provide reflection feature on its generic types. The constructor of a parameterized class (which is the mapping of a GIDL type) will force the user to pass an extra parameter for each generic type introduced by that class. This is needed because otherwise we cannot enforce an exact boxing/unboxing mechanism between our wrapper objects and

```
package GIDL.Base;
import GIDL.*;

public final class BaseStruct<
    C extends GIDL.GIDL_Object,
    E extends GIDL.GIDL_Value_Interf
>
    implements GIDL.GIDL_Value_Interf
{
  // The encapsulated CORBA object
  private org.omg.CORBA.Object obj;
  private C c; private E e;

  public BaseStruct(C c, E e, org.omg.CORBA.Object ob)
  { obj = ob; this.c = c; this.e = e; }
  public BaseStruct(C c, E e)
  { this.c = c; this.e = e; }

  public BaseStruct<C, E> lift(org.omg.CORBA.Object b)
  { return (new BaseStruct<C, E>(c, e, b));  }
  public Base.BaseStruct narrow(BaseStruct<C, E> t)
  { return t.obj;  }

  public BaseStruct<C, E> any_lift(org.omg.CORBA.Any a){
    try{  Base.BaseStruct ob =
            Base.BaseStructHelper.extract(a);
          return (new BaseStruct<C, E>(c, e, ob));
    } catch(Exception exc){ /* ... */ }
  }
  public org.omg.CORBA.Any any_narrow(BaseStruct<C,E>o){
    try{  org.omg.CORBA.Any a = orb.create_any();
          Base.BaseStruct bb = o.obj;
          Base.BaseStructHelper.insert(a, bb);
          return a;
    } catch(Exception exc){ /* ... */ }
  }
  // ...

  public C get_field_c()
  { return (C)c.lift(obj.field_c); }
  public void set_field_c(C co)
  { obj.field_c = c.narrow(co); }

  public E get_field_e()
  { return (E)e.any_lift(obj.field_e); }
  public void set_field_e(C eo)
  { obj.field_e = e.any_narrow(eo); }
}
```

Figure 16: Java wrapper stub mapping

stub objects. The *virtual call* on such an object will invoke the correct boxing/unboxing function for the instantiated type, otherwise the `lift/narrow` methods will be called on the Java erased type and this is not correct.

We omit the implementation details and touch only upon the constructs that are mapped in a conceptually different manner than in the C++ case. The remainder is similar to the C++ mapping. We focus on the Java mapping of the implicitly parametric structures, that is GIDL structures that are nested in the scope of a generic interface, and that use some of the interface's generic type parameters. An example of such structure is the following:

```
    interface Base<C: Object, D, E> {
      typedef struct BaseStruct {
        C field_c;
        E field_e; };
    };
```

Since we have defined that the scope of a generic type parameter is throughout the interface in which it is declared, the example is perfectly legal GIDL code. In order to perform the mapping, we need to know which are the generic parameters used in the structure definition, and also any constraints that apply to them. The Java mapping for the `BaseStruct` parameterized structure, presented above would be that shown in Figure 16.

As we have seen with the C++ mapping, each wrapper stub class implements two methods: `lift` and `narrow`, which are used to encapsulate and retrieve a CORBA-object. However, since Java does not support any run time information with respect to type variables, we cannot declare the `lift` and `narrow` methods statically. We ask the user to provide a trivial object for each type variable in the declaration of an interface. This allows dynamic creation of new instances of the variable type using *virtual calls* to `lift`, `any_lift` on the *trivial* objects. The `any_lift` and `any_narrow` methods are similar to `lift` and `narrow` and are used for the unqualified generic types (as their erasure is the IDL `any` type). In addition, the GIDL wrappers provide an implementation for each method in the declaration of the corresponding GIDL interface and for any the `get` and `set` methods corresponding to fields in the structure definition.

6.3 GIDL to Aldor Mapping

We now describe the high-level ideas used in the Aldor mapping case. Since Aldor was not one of the programming languages for which CORBA provides standard mappings, we have developed our own mappings from scratch. As usual we avoid implementation details and keep the discussion at a high level.

6.3.1 The Aldor Programming Language

As we do not assume the reader to be familiar with the Aldor language, we briefly introduce it here. As mentioned earlier, Aldor [33, 34, 35] is a strongly typed functional programming language with a higher order type system and strict evaluation. Aldor has been used primarily in the area of symbolic mathematics software. The type system has two levels: each value belongs to some unique type, known as its *domain*. Domains are (in principle) run-time values, but they belong to *type categories* that are determined statically. Categories can specify properties of domains such as which operations they export and are used to specify interfaces and inheritance hierarchies. We want to emphasize that throughout this article, the term "category" refers to these type categories and not the objects of mathematical category theory. The biggest difference between the two-level domain/category model and the single-level subclass/class model is that a domain *is an element of* a category, whereas a subclass *is a subset of* a class. This difference eliminates a number of deep problems in the definition of functions with multiple related arguments. Dependent products and mapping types are fully supported in Aldor. Generic programming is achieved through explicit parametric polymorphism, using functions which take types as parameters and which operate on values of those types, e.g.:

```
    f(R: Ring, a: R, b: R): R == a * b - b * a
```

An example of Aldor program is shown in Figure 17. It defines a parametrized category `Module(R)`, representing the mathematical category of *R-Modules*. Categories specify requirements on parameters and state properties of the result. Domains belonging to `Module(R)` export the scalar

```
define Module(R: Ring): Category == Ring with {
   *: (R, %) -> %;
}

Monomial(R: Ring): Module(R) == add {
   Rep == SingleInteger;
   import from Rep;
   (r: R) * (x: %): % == per(r * rep x);
}
```

Figure 17: Aldor category/domain example

multiplication operation, `*`, which returns an element of the domain. In Aldor, within a domain-valued expression, the name `%` refers to the domain name being computed. This is fixed-pointed and can be used as a type name. A type `Rep` is defined for every domain to give a representation for `%`, while `rep` and `per` are type conversion functions (`rep:%->Rep; per:Rep->%`). The `Monomial` domain constructor is declared to return an element of the `Module(R)` category. It has the dependent mapping type `(R: Ring)-> Module(R)`, taking a parameter `R`, of category `Ring`, and producing an *R-Module*. Static analysis can use the fact that `R` provides all the operations required by `Ring`, thus allowing static resolution of names and separate compilation of parameterized modules.

6.3.2 GIDL Mapping

Both GIDL and Aldor provide a set of low-level types, for example fixed size integers and floating point numbers, strings, etc. The correspondence between these low-level types is straightforward. We use Aldor's *ex post-facto* domain extension on the basic types to extend them with the functionality needed by our framework. That is, the existing domains are extended to satisfy new type categories supporting GIDL in an aspect-oriented manner.

A GIDL interface is mapped to an Aldor domain/category pair. The category specifies the exports present in the GIDL interface, together with the casting functionality needed to link it to the CORBA environment and the domain provides the implementation. Because Aldor is not based on classes of objects the mappings of the exported operations all receive one extra parameter corresponding to the implicit "self" parameter of the GIDL methods. Multiple inheritance among GIDL interfaces is matched by multiple inheritance among the Aldor proxy categories. The `Join` operation on categories is used when multiple inheritance is required. Inner GIDL structures and interfaces are directly mapped into inner Aldor domains. Aldor directly supports both types of qualifications present in the GIDL model for generics.

Figure 18 provides an example of these ideas. Note that GIDL *export-based qualification* is directly mapped to Aldor by means of a type parameter qualified by an un-named category. This is specified by means of the "`with`" Aldor construct, which implies that a specific list of exports need to be provided by the type parameter.

The Aldor mapping easily accommodates type parameterized functions, with no need for recompilation, as Aldor supports types as first class values. Types can be passed as parameters to functions and are constructed at run-time.

```
// {\sc gidl} specification:

interface Monomial<R: Ring> : Ring, Module<R> {
    Monomial<R> *(R r, Monomial<R> mon);
}

interface Comp<A> { boolean compare(A a); }

interface Comparator<A:-Comp<B>, B:-Comp<A> > {...}

- - - - - - - - - - - - - - - - - - - - - - - - - - - -

-- Aldor stubs:

define MonomialCat(R:Ring) : Category ==
Join(RingCat, ModuleCat(R)) with {
    *: (R, %) -> %;
}

Monomial(R:Ring): MonomialCat(R) == add {
    -- ...
    (r:R) * (poly:%) : % ==
    -- CORBA remote invocation of the server ...
}

define ComparatorCat(
    A: with{compare:(B)->Boolean},
    B: with{compare:(A)->Boolean} ): Category ==
with {
    -- ...
}
```

Figure 18: Mapping GIDL qualifications to Aldor

7. GIDL AND LIBRARY TRANSLATIONS

This section explores how our generic model and architectural design may be used to expose facilities from a language to a multi-language environment and discuss our mappings in the context of automatic library translation.

Section 7.1 presents an experiment in which we have translated part of the C++'s Standard Template Library functionality into a GIDL server. We conclude that GIDL is able to express the STL semantics and, furthermore, that the library-translation process can be automated. As the GIDL and Aldor languages are very different, Section 7.2 investigates the high-level ideas involved in mapping the semantics of an Aldor library to a GIDL specification and finds that GIDL is effective in rendering these semantics. The Aldor libraries are sophisticated mathematical libraries for exact algorithms for linear and non-linear algebra and they use a high-density of complex type constructors. It is not uncommon for the mathematical types expressed in Aldor to be several levels deep, as in:

```
    Polynomial(Matrix(Complex(Fraction(Integer))))
```

Furthermore, libraries typically have relationships between type parameters, declared, e.g., as

```
    SimpleAlgebraicExtension(R: CommutativeRing,
                             P: UnivPolynomCategory(R),
                             p: P): Algebra(R) == ...
```

With their rich nature, the Aldor BasicMath library and the C++ Standard Template Library make two ideal candidates for experimentation.

```
interface InputIterator<
   T,
   It :- Iterators::InputIterator<T,It> >
: Iterators::BaseIterator<T,It>
{  boolean ==(in It it);  /*...*/  };

interface InpIt<T> : InputIterator<T, InpIt<T> >{};

interface STLvector<
   T,
   Ite :- Iterators::RandAccessIterator<T, Ite>,
   II  :- Iterators::InputIterator<T, II> >
{...};
```

Figure 19: Export-based qualification for iterators

7.1 Accessing the C++ STL in a Multi-Language Environment

We now describe an experiment in which we have exposed the C++ Standard Template Library to a multi-language setting. This tests both our generic model design, as we are able to express STL's fundamental requirements at the GIDL's level and our architectural model as we were able to translate it with minimal programming effort.

7.1.1 Key Features in the Design of STL

STL [7] comprises six major kinds of components: containers (e.g. vectors, lists), generic algorithms (e.g. find, merge, sort), iterators, function objects (classes containing one method that overloads the () operator), adaptors (components that modify the interface of another component), and allocators (memory management encapsulation).

One can identify two key differences between STL and many the other C++ libraries: First, STL containers are not built within an inheritance relation — they are not assumed to be derived from some common ancestor. Second, STL components are designed to be *orthogonal*, unlike traditional container class libraries where algorithms are associated with classes and implemented as member methods. This keeps the source code and documentation small, as for m containers and n algorithms that are applicable to all m containers, only n generic algorithms have to be written and not $m \times n$. On the other hand, the components orthogonal structure addresses the extensibility issue, as it allows user's algorithms to work with the library's containers, or user's containers to work with the library's algorithms. The orthogonality between the algorithm domain and the containers domain is achieved, in part, by the use of iterators; the algorithms in STL are not specified in terms of the data structure on which they operates, but in terms of iterators, which are data structure independent.

However, because of performance guarantees, it might be not possible to plug together any algorithm with any container: For example an efficient *generic* sort algorithm may require random access to the data and in the *list*'s case this is not possible. Thus, STL specifies for each container, which iterator categories it provides and for each algorithm, which iterator categories it requires. These are both defined as English annotations in the standard [7]. Thus, one may observe that C++ does not have sufficient formalism to express the set of requirements STL imposes on its iterators and containers. Next section shows how we can do better with GIDL.

```
interface Algorithms {
  <T, It:-Iterators::InputIterator<T,It> >
  It find(in It first, in It last, in T val);

  <T, II:-Iterators::InputIterator<T, II>,
      OI:-Iterators::OutputIterator<T, OI> >
  OI copy(in II first, in II last, in OI result);

  <T, FI:-Iterators::ForwardIterator<T, FI> >
  void replace(in FI first, in FI last, in T x, in T y);

  // ...
};
```

Figure 20: GIDL specification for STL algorithms

7.1.2 The GIDL Specification for STL

In our mapping, we have preserved the main design characteristics of STL. At the GIDL level, the design of iterators and containers is not intrusive: It does not assume any kind of inheritance from a common ancestor. This is achieved by the use of the *export-based qualification*. We also preserve the orthogonal structure of containers and algorithms. As our goal is to make available the STL library in a multi-component environment through a minimal coding effort, our server relies on the STL's functionality in its implementation.

We have abstracted the STL iterators' functionality, making it context independent. We have done this using additional generic types, bounded by a mutual recursive export-based qualification, as shown in Figure 19. It follows that `InpIt<T>` exports the method `boolean ==(InpIt<T> i)` while `RaiIt<T>` exports the method `boolean ==(RaiIt<T> i)`.

The code excerpt above shows how things work together. The `STLvector` container does not expect the iterators to be built within any inheritance relation, but only for them to implement the functionality described in the STL specification. We use inheritance among iterators just because this provides a better expressivity and keeps the code short. This is not a requirement, however, as seen from the `STLvector` and `InputIterator` definitions. One can extend our specification for a set of iterators without using any inheritance relation among them.

We believe that our STL specification is reasonably expressive and self-describing. A generic algorithm is mapped to a parameterized function, where the type parameters are qualified to enforce the semantics of the GIDL specification. An excerpt is given in Figure 20. This shows how the `find` algorithm uses two type variables in its specification. One, `T`, is unqualified, for the type of the values the iterator is holding. The other, `It`, uses export-based qualification and shares "structural similarity" [1] with its F-bounded qualifier, `InputIterator`.

7.1.3 Implementation Issues

Our implementation uses the STL as a black box, since the goal is not to rewrite STL but rather to export its functionality in a multi-language environment. The C++ GIDL stubs make heavy use of overloading, as STL exposes these features in the specification of iterators and containers. The operations `++, --, +=, -=, [], *` are exported by certain types of iterators, and `==, !=, >, <` are exported by both iterators and containers.

Our GIDL objects for STL are simple wrappers for the STL library constructs. For example, our implementation of `STLvector` keeps an STL `vector` as instance variable and, upon invocation, it calls the appropriate method of the STL `vector` and wraps the returned object to give the corresponding GIDL type. Generic algorithm and function objects are mapped to parameterized functions and interfaces, each containing only one method: the function call operator `()`. Their implementation simply calls the STL function and again wraps the result to a valid GIDL type.

We have already seen that the iterators, together with the functional objects are of central importance in ensuring the STL *orthogonal* design. One consequence of our mapping is that the GIDL stub wrappers corresponding to iterators and functional objects are themselves valid STL types and therefore they can be uniformly manipulated by "native" STL exports such as algorithms and containers, if so desired. An invocation on such a GIDL iterator will redirect the call to the (possibly remote) server side. Thus it is not only possible to have a "black box" automatic library translation strategy, but also to have a distributed implementation for iterators and functional objects.

An interesting and challenging problem is to optimize the usage of the generic library in a multi-language, distributed setting, as many of the implicit assumptions taken in the original design of the library are no longer true. These include the assumptions of a single space and language environment. For example, if in a distributed environment one is traversing an iterator, the performance will greatly suffer, since in order to obtain each value, a foreign, possibly remote call is performed. We have investigated a thread level speculation approach to reduce the communication and dispatching overheads in [22].

7.2 Accessing Aldor's BasicMath Library in a Multi-Language Environment

This section investigates the high-level ideas involved in translating the semantics of the Aldor library to GIDL. This includes the two level type system, functional and category subtyping, dependent types and other issues. As Aldor and GIDL are quite different, the design of a translation scheme that enforces the semantics of the Aldor type system in GIDL is a test of our generic model.

Our previous work in the context of automatic library translation [23] studied what was required to use Aldor libraries to extend Maple [20], a dynamically typed, functional, computer algebra language, in an effective and natural way. The resulting framework, which we called *Alma*, implemented a high-level correspondence between Maple and Aldor concepts and was able to automatically generate Maple stubs corresponding to the functionality of an Aldor library. The user could manipulate Aldor and Maple objects in a completely uniform manner. Work is in progress to enhance the *Alma* framework with support for GIDL specifications.

The main challenge in mapping Aldor semantics in GIDL is to achieve proper functional subtyping constraints for the GIDL interfaces representing Aldor functions. We are do this using semantics constructors for subtype, supertype and set membership relations. These are implemented as the trivial parameterized interfaces `SubType<T>`, `SuperType<T>` and `InstanceOf<T>`. Fluet and Pucella [9] employ a similar "phantom types" technique that uses free type variables to encode subtyping information together with a Hindley-Milner based type system [6] to enforce it.

```
define Ring        : Category ==         with   {...};
define CatA(R:Ring): Category ==         with   {...};
define CatB(R:Ring): Category == CatA(R) with   {...};

fun(b:CatB(Integer),a:CatA(Integer)): Boolean == {...};

define IntegerNumberSystem: Category == Ring with {
  greater : (%, %)         -> %;
  coerce  : (BInt$Machine) -> %;
}
Integer: IntegerNumberSystem == add { ... };

ListCategory(S: Type): Category == with {
  nil     : ()  -> %;
  isEmpty : (%) -> boolean;
  first   : (%) -> S;
  rest    : (%) -> %;
  sort    : ((S, S) -> Boolean, %)    -> %;
  merge   : ((S, S) -> Boolean, %, %) -> %;
}
List(S: Type) : ListCategory(S) == add { ... }
```

Figure 21: Excerpt from Aldor Integer and List

Suppose we want to expose the Aldor exports shown in Figure 21 for use in a multi-language environment. Figure 22 shows the corresponding GIDL specification.

To simulate Aldor's two level (domain/category) type system, we have introduced the trivial GIDL generic interface `InstanceOf<T>`. If the Aldor domain `DomA` $\in$ `CatA`, then in GIDL we make `DomA` to inherit from `InstanceOf<CatA>`. To correctly handle functional and category subtyping we have introduced the trivial GIDL generic interfaces `SubType<T>` and `SuperType<T>`. To express an Aldor subtype or supertype relation between the categories `CatA` and `CatB`, in GIDL we make `CatA` extend `SubType<CatB>` or `SuperType<CatB>` as appropriate. This is needed because if we make `CatA` directly extend `CatB`, we cannot express the supertype relation. This does not introduce any significant run-time overhead. Categories are used in Aldor only to specify properties of domains, such as which operations they export and to qualify type parameters. Domains provide the implementation. Whereas type categories will normally form an interesting subtype lattice, Aldor libraries have only trivial subtype relations among domains. We therefore provide no sub/super-typing relation between them at the GIDL level.

In Aldor, all domains and categories satisfy the `Type` type. The `Domain` and `Category` interfaces are the base classes for the GIDL interfaces corresponding to the Aldor domains and categories. They both thus inherit from `InstanceOf<Type>`. An Aldor declaration such as `R:Ring` is given at the GIDL level as a type qualification `R: InstanceOf<Ring>`.

Figure 22 demonstrates how function parameters and functional subtyping mapped into GIDL. The definition of `ListCategory` in Figure 21 expresses that both `merge` and `sort` functions receive as first parameter a function whose signature is `(S,S)->Boolean`, where `S` is qualified by `Type` in Aldor and `InstanceOf<Type>` in GIDL. Corresponding to the signature of the functional object, the GIDL compiler generates the `Sign_SS_Bool` interface containing only one function, generically named `GIDLapply`. The signature of the function `GIDLapply` illustrates functional subtyping, with contravariant subtyping for parameter types. Note that the types of the parameters `S1` and `S2` are supertypes of the original parameter type `S` of the original declarations for `sort` and `merge`.

```
/********************* Root Types *********************/
interface Type       {}; interface InstanceOf<T>{};
interface SubType<T> {}; interface SuperType<T> {};
interface FunctionalType              {};
interface Category : InstanceOf<Type> {};
interface Domain   : InstanceOf<Type>
{    <C:Category> boolean has(in C c);    };

/********************* Categories *********************/
interface Ring: Category, SubType<Ring>,
  SuperType<Ring>, SuperType<IntegerNumberSystem>    {};

interface CatA< R:InstanceOf<Ring> > : SubType<CatA<R>>,
  Category, SuperType<CatB<R>>, SuperType<CatA<R>>   {};

interface CatB< R:InstanceOf<Ring> > : SubType<CatB<R>>,
  Category, SubType<CatA<R>>, SuperType<CatB<R>>     {};

interface IntegerNumberSystem : Category, SubType<Ring>,
  SubType<IntegerNumberSystem>,
  SuperType<IntegerNumberSystem>                     {};

interface ListCategory< S:InstanceOf<Type> > : Category,
  SubType<ListCategory<S>>,SuperType<ListCategory<S>>{};

/************ Domains: Integer and List<S> ************/
interface GlobalExports {
  Boolean fun(in CatA a, in CatB b);
};

interface Integer : Domain, InstanceOf< Ring >,
    InstanceOf< IntegerNumberSystem >, SubType<Integer>,
    SuperType<Integer> {
  Boolean greater (in Integer i1, in Integer i2);
  Integer coerce  (in long l);
};

interface List< S:InstanceOf<Type> >: Domain,
    SubType<List<S>>, SuperType<List<S>>,
    InstanceOf< ListCategory<S> > {
  List<S> nil    ();
  Boolean isEmpty(in List<S> l);
  S       first  (in List<S> l);
  List<S> rest   (in List<S> l);

    < S1 : SuperType<S>, S2 : SuperType<S>,
      F:-Sign_SS_Bool<S, S1, S2> >
  List<S> sort1  (in F f, in List<S> l);

    < S1 : SuperType<S>, S2 : SuperType<S>,
      F:-Sign_SS_Bool<S, S1, S2> >
  List<S> merge  (in F f, in List<S> l1, in List<S> l2);
};

/***************** Functional Types *****************/
interface fun : FunctionalType {
  Boolean GIDLapply( in CatB<Integer> b,
                     in CatA<Integer> a  );
};

interface greater : FunctionalType {
  Boolean GIDLapply( in Integer i1, in Integer i2 );
};

interface Sign_SS_Bool< S : InstanceOf<Type>,
              S1 : SuperType<S>, S2 : SuperType<S> >
: FunctionalType {
  Boolean GIDLapply( in S1 s1, in S2 s2 );
};
```

Figure 22: GIDL for Aldor exports of Figure 21

The `sort` and `merge` methods of the `List` interface are parameterized with the qualifications

```
< S1: SuperType<S>, S2: SuperType<S>,
  F:- Sign_SS_Bool<S, S1, S2> >
```

and `F` is used instead of the functional type. The export based qualification of `F` ensures that all the possible candidates will be taken into account (see MSGA in Section 4.2). Ultimately, the GIDL compiler will find all the Aldor exports that may have a functional subtype of `(S,S)->Boolean` and will generate interfaces, each containing one method named `GIDLapply`. For example the `greater` and `fun` GIDL interfaces correspond to the Aldor functions with the same names. The `fun` function is a valid first parameter for the `sort` function in `List<CatB<Integer>>` as `CatA<Integer>` extends `SuperType <CatB<Integer>>`. Similarly, `greater` is a valid first parameter for the `sort` and `merge` functions in `List<Integer>` interface as the two signatures are identical.

Finally, we note that in Aldor functional subtyping is most often trivial. Assume that the qualification for `S` is a defined category instead of `Type`. It follows that the extra parameters `S1` and `S2` are not needed because no non-trivial supertype exists. Similar reasoning explains why the mapping does not introduce an extra qualified type:

```
S3: SubType<Boolean>
```

for the return type of the functional object.

To conclude, Figure 22 is a legal GIDL specification that enforces the semantics of the original Aldor code in Figure 21, and the translation process described here has been generalized and automated.

8. CONCLUSIONS

The overall goal of this work has been to examine the feasibility of using parametric polymorphism in the context of multi-language software component systems. We have shown that there are no major impediments to doing this.

The first step was to define a model for generics that could support the interface semantics for generics in a range of different programming languages. This led to our definition of Generic IDL.

We have seen from our implementation of GIDL that qualification of type parameters can be enforced in various target languages, even when the target language does not support qualification of its generics. We have shown that both extension-based and export-based qualification can be supported effectively. Their implementation introduces almost no run-time overhead.

We have shown that this parameterization in GIDL can be supported by translation to IDL, with the generation of appropriate stub/skeleton wrappers. This allows such code to be used with existing CORBA implementations and to take advantage of the usual support for distributed applications. Applications which are not distributed, may make use of GIDL simply to support multi-language use of generic modules. This use involves minimal overhead. We have also shown that, with little modification, GIDL can be used to extend other interconnection architectures such as DCOM and JNI.

Finally, we have demonstrated that it is feasible to export parametric polymorphic libraries to a multi-language environment via GIDL: We have implemented a component that accesses a significant part of the C++ STL functionality.

From this, we have seen how imposing qualification restrictions can improve the precision and safety of the STL library interface. We have also presented the main ideas involved in mapping the Aldor language features to the GIDL level. This allows Aldor libraries to be used across language boundaries via GIDL.

Using GIDL, component-based applications can enjoy the accepted benefits of generic programming: provide clearer and more precise specifications, eliminating ambiguities in the object interface definition, and ultimately exhibit a greater degree of component re-use.

While many special-purpose programming languages have supported parametric polymorphism for some time, it has really only been C++ which has been in mainstream use. Now, with the availability of generics in Java, it is rather important that we understand how to support generics in a multi-language setting. This paper has aimed to make a contribution in this area.

Acknowledgments

The authors would like to thank Michael Lloyd who implemented the Aldor bindings for GIDL and one of the anonymous referees for thorough and helpful comments.

9. REFERENCES

[1] P. Canning, W. Cook, W. Hill and W. Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, 1989.

[2] L. Cardelli. Basic Polymorphism Typechecking. In *Science of Computer Programming*, pages 8 (2): 147–172, April 1987.

[3] R. Cartwright and G. L. Steele. Compatible Genericity with Run Time Types for the Java Programming Language. In *OOPSLA'00 Proceedings*. ACM, 1998.

[4] Y. Chicha, F. Defaix and S. Watt. TR537 - The Aldor/C++ Interface: User's Guide. Technical report, Computer Science Department - The University of Western Ontario, 1999.

[5] Y. Chicha, F. Defaix and S. Watt. TR538 - The Aldor/C++ Interface: Technical Reference. Technical report, Computer Science Department - The University of Western Ontario, 1999.

[6] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[7] A. S. David R. Musser, Gillmer J. Derge. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[8] J. Donahue and A. Demers. Data Types Are Values. In *ACM Transactions in Programming Languages and Systems*, pages 426–445, 1985.

[9] M. Fluet and R. Pucella. Phantom Types and Subtyping. In *IFIP TCS*, pages 448–460, 2002.

[10] J. Y. Girard. Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[11] A. Igarashi, B. Pierce and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[12] R.E. Johnson. Type Object. In *E-Proc EuroPLoP*, http://www.cs.wustl.edu/ schmidt/europlop-96/ papers/paper21.ps, 1996.

[13] M. P. Jones. A System of Constructor Classes: Overloading and Implicit higher-order polymorphism. In *Proc. Functional Programming Languages and Computer Architecture*, pages 52–61. ACM, 1993.

[14] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 conference*, 2000.

[15] H. Ledgard. *ADA An Introduction/ADA Reference Manual*. Springer-Verlag, New York, 1981.

[16] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[17] D. MacQueen. An Implementation of SML Modules. In *Conference on Lisp and Functional Programming*, pages 212–223. ACM, 1988.

[18] N. McCracken. The Typechecking of Programs with Implicit Type Structure. In *Semantics of Data Types, LNCS n.173*, pages 301–316. Springer-Verlag, 1984.

[19] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, pages 17:348–375, 1978.

[20] Maplesoft, *Maple User Manual*. Maplesoft—a division of Waterloo Maple, 2005.

[21] G. Nelson. *Systems Programming with MODULA-3*. Prentice Hall, 1991.

[22] C. E. Oancea, J. W. A. Selby, M. Giesbrecht and S. M. Watt. Distributed Models of Thread-Level Speculation. *Proc. PDPTA*, pages 920–927, 2005.

[23] C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface Between Two Approaches to Computer Algebra. *Proceedings of the ACM ISSAC*, pages 261–268, 2005.

[24] M. Odersky, P. Wadler, G. Bracha and D. Stoutamire. Pizza into Java: Translating Theory into Practice. In *ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[25] M. Odersky, P. Wadler, G. Bracha and D. Stoutamire. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA'1998 Proceedings*, pages 183–200, 1998.

[26] OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision2.4 (October 2000), OMG Specification, 2000.

[27] G. D. Plotkin. A Note on Inductive Generalization. In *Machine Intelligence*, pages 153–163, 1970.

[28] J. C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence, 5(1)*, pages 135–151, 1970.

[29] J. C. Reynolds. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation*, pages 408–425. Springer-Verlag LNCS 19, 1974.

[30] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[31] Sun. Java Native Interface homepage: http://java.sun.com/j2se/1.4.2/docs/guide/jni.

[32] M. Viroli and A. Natali. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. In *OOPSLA'00 Proceedings*, pages 146–165. ACM, 2000.

[33] S. Watt, P. Broadbery, S. Dooley, P. Iglio, J. Steinbach and R. Sutor. A First Report on the $A^\sharp$ Compiler. In *ISSAC 94 Proceedings*, pages 25–31. ACM, 1994.

[34] S. M. Watt. Aldor. In J. Grabmeier, E. Kaltofen and V. Weispfenning, editors, *Handbook of Computer Algebra*, pages 154–160, 2003.

[35] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach and R. S. Sutor. *AXIOM Library Compiler User Guide*. Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.

[36] D. Yu, A. Kennedy and D. Syme. Formalization of Generics for the .NET Common Language Runtime. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.