GIDL User Guide

Cosmin Oancea and Stephen M. Watt

Ontario Research Centre for Computer Algebra
Department of Computer Science
University of Western Ontario
London Ontario, Canada N6A 5B7

Abstract. This document presents the Generic Interface Definition Language framework (GIDL), an extension of CORBA-IDL with support for parametric polymorphism and (operator) overloading. The motivation for this work is two-fold. First, we aimed at allowing multi-language software modules to be combined together to construct distributed applications. In this direction we have investigated what should be the attributes of a common model for parametric polymorphism, so that it can satisfy a wide range of language requirements for specific semantics and binding times. The current version of GIDL provides bindings for a set of representative languages: Java, C++ and Aldor. Second, we aimed at allowing language facilities to be used transparently in a heterogeneous environment. In this direction we have translated part of the C++ Standard Template Library (STL) to a distributed environment, and investigated how to extend the GIDL language bindings in order to preserve the library semantics and its coding idioms.
This document is structured as follows: Section 1 enumerates several articles related to the GIDL framework. Section 2 briefly presents the common object request broker architecture (CORBA), on top of which GIDL is implemented. Section 3 describes GIDL's semantics and the GIDL to IDL translation. Section 4 introduces the general architecture of the GIDL base application, and the high level ideas used in the mapping GIDL generic model to the target languages. Sections 5 and 6 describes the GIDL bindings for the C++ and Java languages. Section 7 describes step by step how to install the GIDL framework, while Section 8 demonstrates the use of the framework.

1 Documentation

There are several articles that relate to the GIDL framework. The most relevant one is "Parametric Polymorphism for Software Component Architectures", by Oancea and Watt [6], that introduces the semantics of the generic model proposed by GIDL, presents the high-level architecture of the GIDL base application and outlines the main ideas employed in generating the C++, Java and Aldor language bindings.

The paper "Generic Library Extension in a Heterogeneous Environment", by Oancea and Watt [7], explores the question of how to structure the GIDL C++ language bindings to achieve two high-level goals: The first goal is to design an extension framework as a component that can easily be plugged-in on top of different underlying architectures, and together with other extensions. The second goal is to use GIDL as a vechicle to export generic libraries to a distributed environment. We address two questions: to what degree can GIDL render the native library interface and semantics, and what are the techniques that will preserve the library coding idioms? In these contexts, the paper identifies the language mechanisms and programming techniques that foster a better code structure in terms of interface clarity, type safety, ease of use, and performance.

Another related paper [2] describes earlier results and synthesize experiments of supporting parametric polymorphism across language boundaries.

All these documents can be found in the *Doc/RelevantArticles* folder.

2 Common Object Requests Broker Architecture

The *Object Management Group* (*OMG*) is a non-profit organization that promotes the use of component technology in heterogeneous, distributed computing systems. OMG pursues this goal through developing standards which allow the distributed object oriented applications to be portable and to interoperate.

The Common Object Request Broker (*CORBA*) [12] is an OMG open standard, which defines an implementation independent architecture for building and seamlessly interconnecting multiple systems involving distributed objects, in a way transparent for the user. In practice, CORBA applications may have some vendor dependency.

CORBA applications are composed of objects, individual units of running software that combine functionality and data. Their design is based on *the OMG Object Model*. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in *OMG IDL* (*Interface Definition Language*) [11].

This allows the framework to be platform and language independent, in the sense that the client interfaces (to the objects), and the server implementations (of these object interfaces) can be specified in any programming language. A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference of the service provider, and actual parameters (if any). The object reference is an object name that defines an object reliably.

The remaining of this section briefly presents the IDL language (Section 2.1), and succinctly describes the components of the CORBA architecture (Section 2.2).

2.1 Interface Definition Language (IDL)

In order to achieve interoperability and portability, the CORBA standard requires the use of the IDL to describe the interfaces of remote objects. The interface is the syntax form of the promise the server object makes to the clients invoking it. This fixes the operations that will be performed and the parameters (input and output) for each. The IDL interface definition is independent of the programming language used for implementation, OMG having standardized mappings to popular programming languages like: C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python[11].

IDL is a declarative language whose syntax was constructed from a subset of C++ and Pascal instructions. It defines basic types (`short`, `byte`, `float`, `double`, `string`, etc.), structured types (`struct`, `sequence`, `array`, `module`), and provides signatures for `interface` types, fully specifying each operation's parameters. Multiple inheritance among interfaces is supported, but recently adopted features like function/operator overloading, and parametric polymorphism are not. However, since it targets distributed objects, IDL forces the user

```
module Examples {
  interface Transaction {
      // ...
  }

  interface BankServer {
    boolean   verifyPIN(in long acctNo, in long pin);
    void      getAcctSpecifics(in long acctNo, in string customerName,
                            out double balance, out boolean isChecking);
    boolean   processTransaction(in Transaction t, in long acctNo);
  }
}
```

Fig. 1. A simple IDL specification for a bank server application

to specify additional information with respect to the object interface, such as which method arguments are *input-only*, *output-only*, or *two-way* data transfers. This is achieved using additional keywords on method arguments, before their type specifications: `in`, `out`, and `inout`.

The remainder of this subsection presents an example of an IDL specification together with a C++ server implementation, and a Java client that accesses the server functionality. The reasons are twofold: First, we want to convey to the reader the "look and feel" of creating a multi-language application in an existing software component architecture (SCA). We are using CORBA in this case but the process is quite similar for DCOM, while for .NET it is even simpler (as it does not support "remote" objects). Second, and most importantly, we want the reader to "feel" the difference between a SCA and a foreign function interface. The latter usually leads to difficult and complex applications, as the programming style specific to a given language is disrupted by numerous calls to "special" kernel functions. It also leads to a rather un-safe program, as very little from what is "foreign" can be statically type-checked. We hope that our example shows that the SCAs are relatively *easy to use*, and *safe*, as they usually enforce statical type-checking of the foreign calls. Moreover, they are a better solution to the multi-language interoperability problem: in order to accommodate n languages, a usual SCA (one that uses an IL) would require $O(n)$ translators, while a solution based on foreign function interfaces would require $O(n^2)$ interfaces.

Figure 1 shows an IDL interface for a simplified bank account server. According to the specification, a `BankServer` object has three methods: one to verify a PIN number against an account, one to get specifics about an account, and one to process a transaction against an account.

In the `BankServer` interface, the two arguments to the `verifyPIN` method are declared as `in` parameters, since they are only used as input to the method and don't need to be read back when the method returns. The `getAcctSpecifics` method has two `in` parameters and two `out` parameters. The two `out` arguments are read back from the server when the method returns as output values. An

```
class BankServer_Impl : public virtual POA_Example::BankServer,
           public virtual ::PortableServer::RefCountServantBase {
private:
  int* pin_arr;   int* balance_arr;   int* account_nr_arr;

  int findAccountIndex(int acctNo) { /* ... */ }

public:
  BankServer_Impl(int* pin, int* bal, int* acctNo) { /* ... */ }

  virtual CORBA::Boolean verifyPIN(int acctNo, int pin)
      throw(CORBA::SystemException) {
      int index = findAccountIndex(acctNo);
      return (pin == pin_arr[index])? 1 : 0;
   }
   // ... further implementation
};
```

Fig. 2. Part of the C++ implementation of the `BankServer`

`inout` argument is both fed to the method as an input parameter, and read back when the method returns as an output value [3]. When the IDL interface is compiled into a client stub and a server skeleton, the input/output specifiers on method arguments are used to generate the code to marshal and unmarshal the method arguments correctly.

Figure 2 presents part of the C++ server that implements the `BankServer` interface. On line 1, our implementation (`BankServer_Impl`) inherits from the IDL skeleton class `POA_Example::BankServer` that has been automatically generated when the IDL specification in Figure 1 was compiled. Thus, it is "linked" to the CORBA framework. Note that the programmer's job is fairly simple, the code being very close to the one written for a single-space, C++ implementation of the bank server.

Figure 3 shows a Java client that uses the functionality of the C++ bank server implemented in Figure 2. The client assumes that the `BankServerObj.ior` file contains a string representation of the bank server object (line 6), together with type information and whatever is required to access the remote reference (machine address and port number). This string is parsed and a remote object interfacing to the server is created on line 12. On line 16 the object is "coerced" to its proper type (`serv` is of type `BankServer`). Finally, the server `serv` can be used as if it is local and it is implemented in Java. Lines 22 and 23 present the execution of two remote operation. Our bank server is required to verify a pin number against an account, and if proved valid, to perform a transaction.

```
static int run(org.omg.CORBA.ORB orb)
       throws org.omg.CORBA.UserException {
  org.omg.CORBA.Object obj = null;
  try {
    String refFile = "BankServerObj.ior";
    java.io.BufferedReader in =
      new java.io.BufferedReader(new java.io.FileReader(refFile));
    String ref = in.readLine();
    obj = orb.string_to_object(ref);
  }catch(java.io.IOException ex) { return -1; }

  Examples.BankServer serv  = Examples.BankServerHelper.narrow(obj);
  Example.Transaction trans = ...; //create a transaction object

  int acctNo = 1211356256;
  int pin    = 2145;
  if(serv.verifyPIN(acctNo, pin))
    serv.processTransaction(trans, acctNo);
}
```

Fig. 3. A simple Java client using the bank server

2.2 Overview of CORBA Architectural Components

This section follows the "A Brief Tutorial in CORBA" work of Kate Keahey [5]. Figure 4 shows the main components of the ORB architecture and their interconnections.

The central component of CORBA is the *Object Request Broker* (*ORB*). The ORB is the middleware that establishes the client-server relationships between objects. It encompasses all of the communication infrastructure needed to identify and locate objects, handle connection management and deliver data. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results.

It is important to emphasize that both the client and the server of CORBA objects use an ORB to talk to each other (they both have an object manager associated with them), and this leads to the fact that any agent in a CORBA system may act as both a client and a server of remote objects. In general, the ORB is not required to be a single component; it is simply defined by its interfaces (see *ORB interface* in Figure 4). The *ORB Core* is the most important part of the ORB as it handles the communication of requests.

On the client side of an object request, the ORB is responsible for accepting client requests for a remote object, finding the implementation of the object in the distributed system, accepting a client-side reference to the remote object,

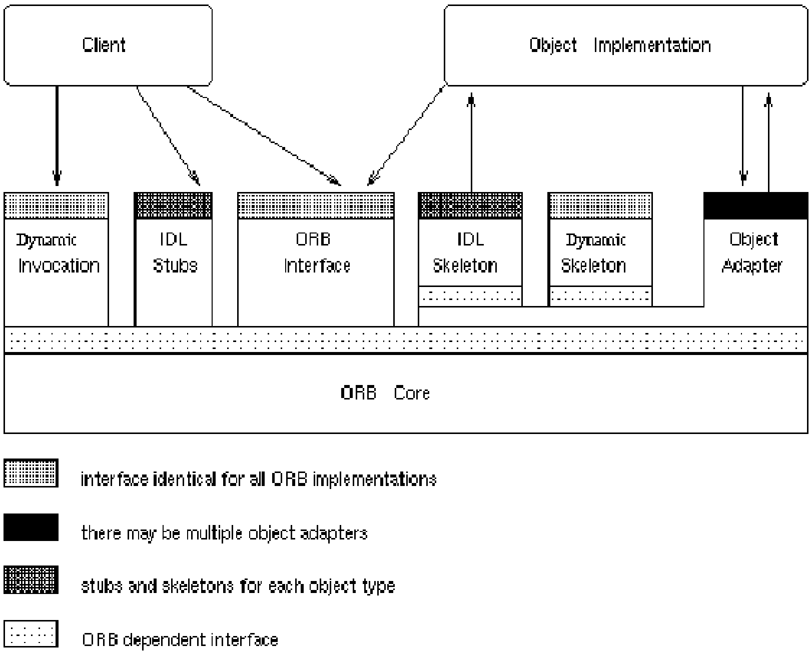

Fig. 4. Main components of the CORBA architecture

routing client method calls through the object reference to the remote object implementation, and accepting any results for the client. On the server side, the ORB lets object servers register new objects. When a client requests an object, the server ORB receives the request from the client ORB, and uses the object's skeleton interface to invoke the object's activation method. The server ORB generates an object reference for the new object, and sends this reference back to the client. The client ORB converts the reference into a language-specific form (a Java stub object, in our case), and the client uses this reference to invoke methods on the remote object. When the client invokes a method on a remote object, the server ORB receives the request and calls the method on the object implementation through its skeleton interface. Any return values are marshaled by the server ORB and sent back to the client ORB, where they are unmarshaled and delivered to the client program. So ORBs really provide the backbone of the CORBA distributed object system.

Given an IDL specification, the IDL compiler will generate IDL stub/skeleton code (not presented in Figure 4). The stub will act as an interface for the *remote object* while the skeleton will provide the implementation. To perform a remote operation, the client transfers a request to the ORB Core via the *IDL stub* or through the *Dynamic Invocation Interface* (*DII*). The *IDL stub* represents the mapping between the implementation language of the client and the ORB core. It follows that the client can be written in any language as long as the implementation of the ORB supports this mapping. The ORB Core then transfers the request to the object implementation which receives the request as an up-call through either an *IDL skeleton*, or a *dynamic skeleton* [5].

The *Object Adapter* (*OA*) is the architectural component responsible for the communication between the object implementation and the ORB core. It handles services such as generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping references corresponding to object implementations and registration of implementations. `POA` is one of the CORBA standard object adaptors (see Figure 2, line 1).

There are two ways to specify the object interfaces: through an IDL specification, or by directly adding them to the *Interface Repository (IR)* – a database which provides persistent storage of object interface definitions. *The Dynamic Invocation Interface* (*DII*) enriches the CORBA object with reflective features: it allows the client to specify requests to objects whose definition and interface are unknown at the client's compile time. To use *DII*, the client composes a request (in a standard way to all ORBs) that contains the object reference, the name of the operation to be invoked, and a list of parameters. The object services are retrieved from the IR and the proper operation is invoked.

3 Generic Interface Definition Language

We have briefly reviewed in the previous section the CORBA-IDL language. This chapter presents the syntax and semantics of Generic Interface Definition Language (GIDL), our extension to CORBA–IDL that supports parametric polymorphism, and method/operator overloading.

We emphasize that GIDL is not 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 our work. 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 a similar extension for various SCAs, 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 assumptions we constructed a generic model for GIDL in some ways similar to that of Java and GJ [15]. We are using a homogeneous implementation approach, based on a type erasure technique which ensures the backward compatibility with the non-generic applications written for the underlying SCA. 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 The GIDL 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: `f(A:with{op:(SI)->SI},a:A): SI ==...;`.

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 5.

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 be validated by the compiler 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 33 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 34 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.

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>{  };
```

```
module GenericStructures {
  interface PriorElem {
    short getPriority();
    short compareTo(in Object r);
  };
  interface Foo_extend : PriorElem { /* ..... */ };

  // Assume Foo_export is not in a "isA" logical relation with
  // PriorElem so we did not want to inherit from it
  interface Foo_export{
    short getPriority();
    short compareTo(in Object r);
    //...
  };
  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(); // OK
    Test<Foo_export, Foo_export> op2(); // ERROR
    Test<Foo_extend, Foo_extend> op2(); // OK
  };
  // ...
};
```

Fig. 5. Generic interfaces with different generic type qualifications

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 a *unqualified* or *universally qualified* generic types, similar to templates in C++ (e.g. PriorityQueue3<A>). This allows the instantiation to be any GIDL type.

GIDL **does not support type parameterized methods**, even if this feature is 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 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, one could pass extra reflective-parameters that encapsulate the type-information of the generic type instantiations. The server-side would then generate 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* could be finally called to complete the *original* invocation. The *generated methods* corresponding to different instantiations of the exposed type parameters could be cached for later reuse. However, we have not implemented this mechanism.

3.3 IDL Grammar Changes to support generic types

To provide syntax for parametric forms, we have modified the OMG IDL grammar as shown in Figure 6. The <template_dcl_unit> product is form by an identifier followed, optionally by an extend/export qualification. The latter comprises one of the :/:- symbols followed by a <scoped_name> product.

The <template_call_unit> product is satisfied by any type (<integer_type>, <char_type>, <value_based_type>, <scoped_name>, and so on).

The <template_dcl>/<template_call> are sequences of comma separated <template_dcl_unit>/<template_call_unit>.

To extend IDL with parametric polymorphism we have modified the derivation rule for the scoped_name product. The new rule is satisfied by ::Outer<T1,Object>::Inner<T2, String>, for example.

Furthermore, we have modified the derivation rules for <interface_header> and <forward_dcl> in order to allow the declaration of type-parameterized interfaces.

To allow F-bounded polymorphism, the type checking phase first records in the symbol table all the types introduced by a <template_dcl> product, and just then checks the validity of the qualifiers. Thus, expressions such as:

```
interface Test< A:Comparable<B>, B:Comparable<A> > ...
```

are accepted by the GIDL language.

```
//...

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

//...
```

Fig. 6. Adding support for parameterized interfaces to the IDL grammar

3.4 More semantics for the GIDL generic types

We discuss a few details, with examples referring to the GIDL specification in Figure 7. 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 [10, 15], 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 7, 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. Since the *export-based qualification* can be reduced to an *extend-based qualification* at GIDL level, the type checking mechanism in this case will be similar to the one presented above.

```
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
};
```

Fig. 7. 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 traversing up 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 should note that in the `Comp` interface, the first appearance of `A` is related to a `template_dcl` production in the grammar, while the second one is related to a `template_call` (now `A` is a `scoped_name`, which was defined in the `template_dcl` part). The `op1` operation takes as parameter a `BaseStruct` variable. The `Comp` interface extends the `Base` interface, and `BaseStruct` is defined inside the `Base` interface. Thus, `BaseStruct` is also in the scope of the `Comp` interface, so that the `op1` declaration is valid. Notice that `BaseStruct` makes use of a generic type (i.e. `C`) defined within the `Base` interface. The signature of the `op1` function is dependent on the signature of the `BaseStruct` structure, whose exact syntax in this context is: `Base<A>::BaseStruct` (notice that `A` is a generic type and when the operation is invoked, it has to be substituted for a real type). The mapping to C++ and Java languages have to ensure `op1/op2` invocation consistency at client level. Thus, we see that the `BaseStruct - type` is dependent on the type of the generic type in the `Base` interface, and this information cannot be encapsulated at the symbol table level. Instead we need to attach to each `scoped_name` a template activation record (containing information about the generic type context in which it appears). In the case of the

BaseStruct appearing in the op1 operation of the Comp interface, the template activation record is filled with information by up traversing the inheritance tree of the Comp interface, until we find the actual definition of BaseStruct. In the case of the op2 operation appearing in the Comp interface, all the information is stored inside the scoped_name of the 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.

3.5 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 8 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 $\overline{X}\overline{\lhd}\overline{N}$ as a shorthand for $X_1 \lhd_1 N_1, ..., X_n \lhd_n N_n$. 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 $\overline{X}\overline{\lhd}\overline{N}$ where $\lhd_i$ is one of the *extend* or *export* based qualifications. For brevity, some obvious rules are omitted from Figure 8: 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 8 simply says that if the declaration of interface I begins with $interface I < \overline{X}\overline{\lhd}\overline{N} >$, 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 [4]. The *named subtyping* rule ("<:") in Figure 8 is also straight forward: the inheritance hierarchy is dictated by the interface table IT.

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

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

(Named subtyping "<:")

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

(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)

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

(Method inclusion "$\preceq$" – III)

$$\frac{\begin{array}{l} M_1 =< \overline{X_1 \triangleleft_1 N_1} > R_1\ \ m(\overline{P_1}) \quad M_2 =< \overline{X_2 \triangleleft_2 N_2} > R_2\ \ m(\overline{P_2}) \\ \Delta \vdash \overline{P_1} = [\overline{X_1}/\overline{X_2}]\overline{P_2} \qquad\qquad\qquad\qquad \Delta \vdash R_1 = [\overline{X_1}/\overline{X_2}]R_2 \\ \triangleleft_1, \triangleleft_2 \in \{\ :\ ,\ \ :\text{-}\ \} \qquad\qquad \Delta \vdash \overline{N_1\ \ \psi(\triangleleft_1, \triangleleft_2)\ \ [\overline{X_1}/\overline{X_2}]N_2} \end{array}}{\Delta \vdash M_1 \quad \preceq \quad M_2}$$

$$\psi(\triangleleft_1, \triangleleft_2) = \begin{cases} \triangleleft_1 \ = \ \triangleleft_2 \ = \ : & then \quad : \\ \triangleleft_1 \ = \ \triangleleft_2 \ = \ :\text{-} & then \quad :\text{-} \\ \triangleleft_1 \ = \ :\ \ and\ \ \triangleleft_2 \ = \ :\text{-} & then \quad :\text{-} \\ \triangleleft_1 \ = \ :\text{-}\ and\ \ \triangleleft_2 \ = \ : & then \quad \eta\ \ where \\ \quad O_1 \eta O_2 = true\ \ if\ \ \{I | I <:\text{-} O_1\} \subseteq \{I | I <: O_2\}, \\ \quad \text{and}\ \ false\ \ \text{otherwise} \end{cases}$$

Fig. 8. Type rules for two varieties of qualification

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 ($\overline{X} \triangleleft \overline{N}$), 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* $X_2 :\!- O_2$ 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 $O_1 :\!- O_2$. A similar line of reasoning leads to the definition of the ψ operator in Figure 8. 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 $X_1 :\!- O_1$ and $X_2 : O_2$ then M_1 is not $\preceq$-included in M_2.

3.6 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 5, it will generate the IDL specification in Figure 9.

```
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();
  };  // ...
};
```

Fig. 9. The generated IDL specification

We previously noted that, at least for the *un-qualified* type variables, any CORBA-IDL's type can be a candidate for the generic type substitution, and this includes basic types. There are some programming languages (GJ, Modula3), that do require the generic parameters to be classes/interfaces or both. For this reason, we decided to map the GIDL's basic types into some wrapper classes at the stub level (mapped programming language dependent) that would look like the following if they had a GIDL representation:

```
interface GIDL_Double{
  double getID();
  double getValue();
  double setValue(in double d);
};
```

3.7 Extending IDL with Operator/Method Overloading

Method Overloading

It was trivial to extend GIDL with method overloading. The compiler checks the names of all the operations exported by a certain interface. If it finds name-duplicates of methods with different signatures, it generates unique names for the corresponding methods. The IDL erased file uses these unique names. The targeted languages, C++, Java, Aldor, all support method overloading. Thus the GIDL wrapper will export the method names that appear in the GIDL specifica-

```
// GIDL specification
interface Overloading<T> {
    long fun(in T t);
    long fun(in T t, in long l);
};

// The corresponding IDL erased file generated by the compiler
// for the above GIDL specification
interface Overloading {
    long fun1(in Any t);
    long fun2(in Any t, in long l);
};

// C++ wrapper pseudocode for the Overloading interface
GIDL::Long fun(T t) {
    /*...*/
    CORBA::Long l = obj->fun1(T::_any_narrow(t));
    /*...*/
}
GIDL::Long fun(T t, GIDL::Long l) {
    /*...*/
    CORBA::Long l = obj->fun2(T::_any_narrow(t), GIDL::Long::narrow(l));
    /*...*/
}
```

Fig. 10. Adding support for method overloading

tion, but its implementation will call the corresponding CORBA operation that has an unique name. Figure 10 exemplifies this approach.

Operator Overloading

Currently the set of operators supported by GIDL are the one of C++, excepting the = operator:
"->*", "<<=", ">>=", "+=", "++@p", "++@a", "->", "--@p", "--@a", "-=", "*=", "¯, "==", "<=", "<<", ">=", ">>", "%=", "&&", "&=", "≙", "!=", "||", "|=", "[]", "()", "+@u", "+@b", "-@u", "-@b", "*@u", "*@u", "*@b", "/", "<", ">", "%", "&@u", "&@b", "ˆ, "!", " ", "|", ",".
The symbol @ is used to specify additional information about the operator, such as if it is binary or unary. b stands for binary operator, u stands for unary operator, p stands for prefix operator, a stands for postfix operator. Note that the user does not have to write this abreviatons, since the compiler can infer whether the operator is unary, binary, tertiary or multi, from the number of arguments the operator receives. However, if the user does not specify whether the operator is a prefix/postfix one, for example in the case of ++ or --, the compiler will chose by default the prefix form.

```
//...

<id_or_op> ::= IDENTIFIER:id   |   IDENTIFIER_OP:id
     ;

<ops_overl_metadata> ::=
            "Java" | "Cpp" | "Aldor" ":" <id_or_op>
     |
            "Java" | "Cpp" | "Aldor" ":" <id_or_op> ";" <ops_overl_metadata>
     ;

<method_name> ::=
            IDENTIFIER
     |
            IDENTIFIER_OP ["{" <ops_overl_metadata> "}"]
     ;

//...
```

Fig. 11. Adding support for operator overloading to the IDL grammar

However, not all the languages support operator overloading (e.g. Java), and the ones that suport this feature do not necessarily export the same set of operators. Our goal was for GIDL to implement a flexible extension mechanism for operator overloading. The compiler has a list of default names that it is going to use to replace the operator name in the GIDL wrapper for each target language. These names can be valid identifiers or operators for a certain language. The user can provide a specification file that overrides the default names of the compiler. Furthermore, we provide a mechanism that allows the programmer to again override these names in the GIDL specification, as below:

```
interface TestingOperators{
  void  operator"++@p"
            {Java: pp_pref; Cpp: operator"++"; Aldor: operator"++@p"}
        ();
};
```

The `++` operator of the `TestingOperators` interface will be mapped to the `++` operator for the C++ and Aldor languages, while the Java wrapper stub will replace it with a method called `pp_pref`. Although not implemented, it shall be a straight-forward extension for GIDL to accept any conceivable operator, and employ the code generation engine to decide whether that operator is legal for a specific language binding or not. In the latter case, the compiler will map it instead to a method whose name is a viable identifier for that language (in case the user has not provided a name for it).

To accomodate operator overloading in the IDL grammer, we introduced a new terminal, called OPERATOR_ID that recognizes symbols of the form: operator"<any>", where <any> can be any string that does not contain the space, tab, or the end of line characters.

Figure 11 presents the changes we made to the IDL grammer to support operator overloading. <id_or_op> denotes either an identifier or an operator identifier. The <ops_overl_metadata> is a sequence of pairs of one of the strings "Java", "Cpp", or "Aldor" and a GIDL identifier or operator. The method name is an identifier or an operator, where the latter can be followed by an <ops_overl_metadata> product.

Note that at the parser level, an operator name can be any string. It is the type-checking job to decide whether the operator is supported by GIDL or not. This allows the set of supported operators to be extended in the compiler without modifying the parser.

4 High-Level View of the GIDL Architecture

This chapter presents a high level view of the GIDL architecture: that is how the architecture components are created and how they interact to accomplish an invocation successfully. It then shows how a programmer may use our architecture, and argues 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.

4.1 The GIDL Extension Architecture

Figure 12 illustrates the design of the 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 [11].

The solid arrows in Figure 12 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 invokes 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 [11].

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.

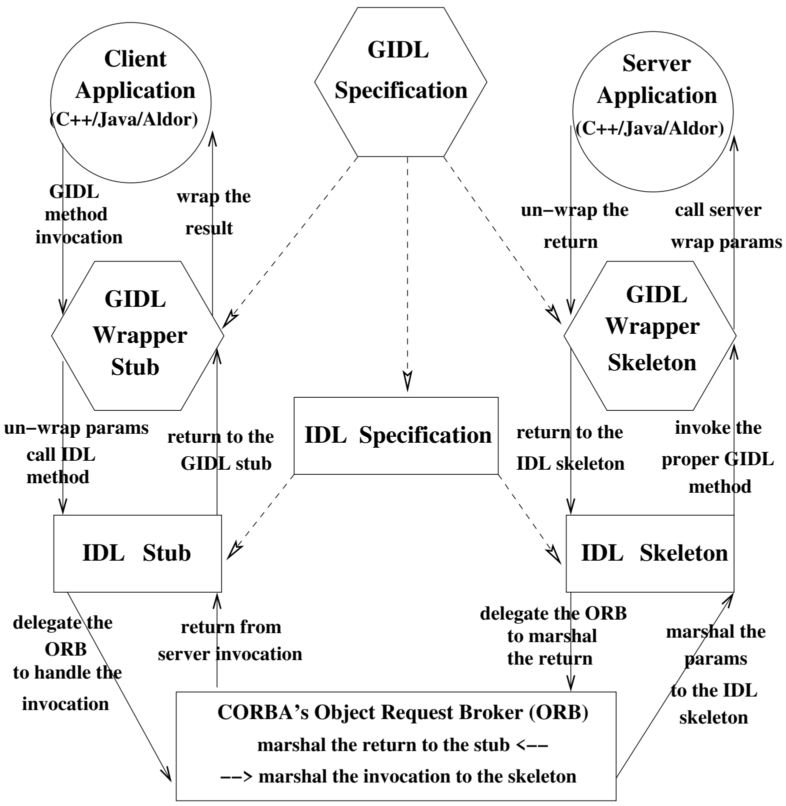

Fig. 12. GIDL architecture for CORBA
circle – user code
hexagon – GIDL *component*
rectangle – CORBA *component*
dashed arrow – is compiled to
solid arrow – method invocation flow

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 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. However, our main goal has been to add genericity to the IDL - level, hence the above features are not implemented.

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 (*IDL stub* box in Figure 12). 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 ("javah" utility) 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.

4.2 The User's Perspective

Consider the GIDL specification shown in Figure 13. When implementing the server side, the programmer should extend the generated skeleton wrapper classes `PriorQueue2` and `PriorElem`, implementing the operations that appear in the

```
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);
};
```

Fig. 13. GIDL code for a simple priority queue

```
CORBA::Object_var obj  = orb->string_to_object(s);
GIDL::PriorQueue2<GIDL::PriorElem> gpq(pq_orig);
GIDL::PriorElem gPEobj = gpq.createNewA(GIDL::Short_GIDL(1));

gpq.enqueue(gPEobj); // OK

// Obtain a reference to a CORBA::Object - obj ...

gpq.enqueue(obj);    // Error

GIDL::PriorElem  gPEobj  = gpq.dequeue();
GIDL::Short_GIDL sh      = PEobj.getPriority();

cout<<sh<<endl;  //prints "1"
```

Fig. 14. Code excerpt from a C++ client

GIDL specification. This is the usual CORBA procedure for writing servers, so the user will find no difficulty here.

An excerpt from a C++ client program that makes use of the types defined in this GIDL specification is shown in Figure 14. 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). 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 5, 11, 12). In Figure 14, `GIDL::Short_GIDL` is the C++ mapping type for GIDL's `short`. Line 9 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 13, 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.

4.3 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 generic model we do as much work as possible at the unified level and in the GIDL to IDL translation, to minimize the language specific details.

Basic Ideas

Type-erasure for an *extend-based qualified* generic type is achieved by substituting it with its bounding-interface. 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 15.

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 [14] or equivalently the least general generalization [13] of the types that satisfy the *export-based qualifier*.

Section 3.5 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

```
// 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(); //...
    }//...
}
```

Fig. 15. *Extend-based qualification* mapping to C++

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.

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*.

As an example, consider the GIDL file shown in Figure 16. 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 parame-

```
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
}
```

Fig. 16. MSGA Example

ters 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

```
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...};
```

Fig. 17. The result of the MSGA Algorithm

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 17.

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 3.5 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 18 presents a legal GIDL specification, together with its corresponding MSGA bindings. The first example in Figure 18 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

```
//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>>{...};
```

Fig. 18. More MSGA Issues

`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 18 shows that only one MSGA (`MSGA3`) is constructed for the `D` and `E` *export-based qualifications* (lines 6,7).

5 GIDL to C++ Mapping

This chapter presents the rationale behind the GIDL C++ bindings. We start by presenting the high-level mapping ideas, and the approach used to implement the casting functionality of the GIDL wrapper objects. We then show how the GIDL inheritance hierarchies are implemented and comment on the language features that we found most useful in this context. Finally, we demonstrate the ease of use of the GIDL extension and reason about the soundness of the translation mechanism.

5.1 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. 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 19, 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.

5.2 The Generic Base Class

Figure 20 presents a simplified version of the base class for the wrapper object whose GIDL type is `String`, `WString` or some interface. The type parameter `T` denotes the current GIDL class, `A` is its corresponding CORBA class, while `A_v` denotes the CORBA smart pointer helper type that assists with memory management and parameter passing. The `BaseObject` class inherits from the

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

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

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

Fig. 19. Nested structures

`ErasedBase` class that stores the type-erased representation under the form of a void pointer, and from the `GIDL_Type`, the supertype of all GIDL types. The `fillObjFromAny` and `fillAnyFromObj` functions abstract the CORBA functionality of creating an object from a CORBA `Any`-type value, and vice-versa. They are re-written for the `String`/`WString` types as the CORBA specific calls differ. The implementation provides overloaded constructors, assignment operators and accessor functions that work over various CORBA and GIDL types, allowing the user to manipulate in an easy and transparent way GIDL wrapper objects.

The generic constructor (lines 18-20) receives as a parameter a GIDL object whose type is in fact `GG`. The use of `BaseObject<GG, GG::GIDL_A,GG::GIDL_A_v>`, together with the cast to `A*` in line 20, statically checks that the instantiation of the type `GG` is a GIDL interface type that is a subtype of the instantiation of `T` (with respect to the original GIDL specification). This irregular use of the `BaseObject` type constructor is one of the generalized algebraic data types GADT characteristics. Note also the use of the *abstract type members* `GG::GIDL_A` and `GG::GIDL_A_v`. The mapping also defines a type-unsafe cast operator (lines 24-29) that allows the user to transform an object to one of a more specialized type. The implementation, however, statically ensures that the result's type is a subtype of the current type.

5.3 Handling Multiple Inheritance

We now present the rationale behind the C++ mapping of the GIDL inheritance hierarchies. There are two main requirements that guided our design:

```
class ErasedBase { protected: void* obj; };
template<class T,class A,class A_v> class BaseObject :
  public ErasedBase, public GIDL_Type<T> {
 protected:
  static void fillObjFromAny(CORBA::Any& a, A*& v) {
    CORBA::Object_ptr co = new CORBA::Object();
    a>>=co; A* w = A::_narrow(co); v = w;
  }
  static void fillAnyFromObj(CORBA::Any& a, A* v) { a<<=v; }
 public:
  typedef A GIDL_A;   typedef A_v GIDL_A_v;   typedef Self T;

  BaseObject(A* ob)          { this->obj = ob; }
  BaseObject(const A_v& a_v) {this->obj=a_v._retn();}
  BaseObject(const T& ob)    { this->obj = ob.obj; } //
  BaseObject(const GIDL::Any_GIDL& ob)
  {T::fillObjFromAny(*ob.getOrigObj(),getOrigObj());}
  template<class GG> BaseObject(
    const BaseObject<GG,GG::GIDL_A,GG::GIDL_A_v>& o
  ) { this->obj = (A*)o.getOrigObj(); }
  /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/

  operator A*() const { return (A*)obj; }
  template < class GG > operator GG() const{
    GG g; // test GG superclass of the current class!
    if(0) { A* ob; ob = g.getOrigObj(); }
    void*& ref = (void*&)g.getOrigObj();
    ref = GG::_narrow(this->getOrigObj()); return g;
  }
  A*& getOrigObj()  const { return (A*) obj; }
  void setOrigObj(A* o)   { obj = o; }

  static A*& _narrow(const T& ob){return ob.getOrigObj();}
  static CORBA::Any* _any_narrow(const T& ob) { /* ... */ }
  static T _lift(CORBA::Any& a, T& ob)
  { T::fillObjFromAny(a,ob.getOrigObj()); return ob; }
  static T _lift(CORBA::Object* o) { return T(A::_narrow(o));}
  static T _lift(const A* ob)      { return T(ob); }
  /*** SIMILAR: _lift(A_v) AND _lift(CORBA::Any& v) ***/
 };
```

Fig. 20. The base class for the GIDL wrapper objects whose types are GIDL interfaces. (We have omitted the `inline` keyword)

```
interface Comparable< K >
{ boolean operator">" (in K k); boolean operator"=="(in K k); };

interface BinTree< K:-Comparable<K>, D >
{ D getData();     K getKey();     D find(in K k);            };
interface Leaf< K:-Comparable<K>, D > : BinTree<K,D>
{ void init(in K k, in D d);                                  };
interface Node< K:-Comparable<K>, D > : BinTree<K,D>
{ BinTree<K,D> getLeftTree();    BinTree<K,D> getRightTree(); };

interface Integer : Comparable<Integer>    { long getValue(); };
```

Fig. 21. GIDL specification and C++ client code for a binary tree

```
template<class K, class D> BinTree {
    protected:    ::BinTree* obj;
    public:   // system functionality
        void setOrigObj(::BinTree* o) { obj = o; }
              // GIDL specification functionality /* ... */
};
template<class K, class D> Node : public virtual BinTree<K, D> {
    protected:    ::Node* obj;
    public:   // system functionality
        void setOrigObj(::Node* o)   { obj = o; }
              // GIDL specification functionality
        BinTree<K,D> getLeftTree()   { /* ... */ }
};
```

Fig. 22. Naive translation for the C++ mapping

- As far as the representation is concerned, each GIDL wrapper stores precisely one (corresponding) CORBA-type object: its erasure. This is a performance concern. It is important to keep the object layout of the GIDL stub wrapper small.
- In terms of functionality, the GIDL wrapper features only the casting functionality associated with its type; in other words the *system* functionality is not subject to inheritance. This is a type-soundness, as well as a performance concern.

Throughout this section we refer to the GIDL specification in Figure 21. We first examine the shortcomings of a naïve translation that would preserve the inheritance hierarchy among the generated GIDL wrappers. Figure 22 shows such an attempt. If each GIDL wrapper stores its own representation as an object of its corresponding CORBA-type, the wrapper object layout will grow exponentially. An alternative would be to store the representation under the form of a void

```
template<class K,class D> class Leaf_P : public BinTree_P<K,D>{
  protected:
    virtual void*   getErasedObj() = 0;
    ::Leaf* getObject_Leaf(){ return (::Leaf*)getErasedObj(); }
  public:
    void init(const K& a1, const D& a2) {
      CORBA::Object_ptr& a1_tmp = K::_narrow(a1);
      CORBA::Any& a2_tmp = *D::_any_narrow(a2);
      getObject_Leaf()->init(a1_tmp, a2_tmp);
    }
};
template<class K,class D> class Leaf :
     public virtual Leaf_P< K, D >,
     public BaseObject<Leaf<K,D>,::Leaf,::Leaf_var>
{
  protected:
    typedef Leaf<K,D> T;
    typedef BaseObject<T,GIDL_A,GIDL_A_v> BT;
    void*   getErasedObj()     { return obj; }
  public:
    Leaf()                          : BT()  { }
    Leaf(const GIDL_A_v a)          : BT(a) { }
    Leaf(const GIDL_A* a)           : BT(a) { }
    Leaf(const T & a)               : BT(a) { }
    Leaf(const Any_GIDL & a)        : BT(a) { }
    template <class GG> Leaf(
      const BaseObject<GG, GG::GIDL_A, GG::GIDL_A_v>& a
    )                               : BT(a) { }
    /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/
};
```

Fig. 23. Part of the C++ generated wrapper for the GIDL::`Leaf` interface. `::Leaf` and `::Leaf_var` are CORBA-types

pointer in a base class and to use virtual inheritance (see the `BaseObject` class in Figure 20). However, then the system is not type-safe, since the user may call, for example, the `setOrigObj` function of the `BinTree` class to set the `obj` field of a `Node` GIDL wrapper. Now calling the `Node::getLeftTree` method on the wrapper will result in a run-time error. This happens because the `Node` wrapper inherits the *casting functionality* of the `BinTree` wrapper.

Figure 23 shows our solution. The abstract class `Leaf_P` models the inheritance hierarchy in the GIDL specification: it inherits from `BinTree_P` and it provides the implementation for the methods defined in the `Leaf` GIDL interface (n.n. `init`). Our mechanism resembles Scala [8] traits [9]. `Leaf_P` does not encapsulate state and does not provide constructors, but inherits from the `BinTree_P` "trait". It *provides the services* promised by the corresponding GIDL interface, and *requires an accessor* for the CORBA object encapsulated in the wrapper (the `getErasedObj` function).

Finally, the `Leaf` wrapper class aggregates the casting functionality and the services promised by the GIDL specification by inheriting from `Leaf_P` and `BaseObject` respectively. It rewrites the functionality that is not subject to inheritance: the constructors and the assignment operators by calling the corresponding operations in `BaseObject`. Note that there is no subtyping relation between the wrappers even if the GIDL specification requires it. However, the templated constructor ensures a type-safe, user-transparent cast between say `Leaf<A,B>` and `BinTree<A,B>`.

To summarize, the C++ binding uses GADT*s* and *abstract type members* to enforce a precise meta-interface of the extension. The latter we simulate in C++ by using templates in conjunction with `typedef` definitions. Further on, the functionality described in the GIDL interface is implemented via *traits*. We represent traits in C++ as abstract classes and the require services as abstract virtual methods. The latter are provided by the GIDL wrapper that "mixins" the two-way GIDL-CORBA casting with the functionality published in the specification. Our extension experiment constitutes another empirical argument to strengthen Odersky and Zenger's claim that *abstract type members*, and *modular mixin composition* are vital in achieving first-class value components. We would add the GADT technique to that.

5.4 Ease of Use

One additional feature of the GIDL framework, in our view, is that it is much simpler to be used than its underlying CORBA architecture. At a high-level, this is accomplished by making the GIDL wrappers to encapsulate a variety of constructors, cast and assignment operators.

Figures 24`A` and `B` illustrate the CORBA/GIDL code that inserts GIDL/CORBA `Octet` and `String` objects into `Any` objects, then performs the reverse operation and prints the results. Note that the use of CORBA specific functions, such as `CORBA::Any::from_string`, is hidden inside the GIDL wrappers; the GIDL code is uniform with respect to all the types, and mainly uses constructors and assignment operators. All GIDL wrappers provide a casting operator to their original CORBA-type object that is transparently used in the statement that prints the two objects. Figure 24C presents the implementation of the generic assignment operator of the `Any_GIDL` type. Since `GIDL_Type` is an abstract supertype for all GIDL types, its use in the parameter declaration statically ensures that the parameter is actually a GIDL object. By construction, the only class that inherits from `GIDL_Type<T>` is `T`, therefore the dynamic cast is safe. Finally the method calls the `T::_lift` operation (see Figure 20) that fills in the object encapsulated by the GIDL `Any` wrapper with the appropriate value stored in the `T`-type object.

Figure 24D presents one of the shortcomings of our mapping. The GIDL wrapper for arrays, as for all the other GIDL wrapper-types, has as representation its corresponding CORBA generic-type erased object. The representation for an `Array_T`-type object will be an array of the CORBA `Any` type objects, since the

```
// A. CORBA code
using namespace CORBA;
Octet oc = 1; Char* str = string_dup("hello"); Any a_oc, a_str;
a_str <<= CORBA::Any::from_string(str, 0);
a_oc  <<= CORBA::Any::from_octet (oc);
a_oc  >>= CORBA::Any::to_octet   (oc);
a_str >>= CORBA::Any::to_string  (str, 0);
cout<<"Octet (1): "<<oc<<" string (hello): "<<str<<endl;

// B. GIDL code:
using namespace GIDL;
Octet_GIDL oc(1); String_GIDL str("hello"); Any_GIDL a_oc, a_str;
a_oc  = sh;  a_str = str;  oc    = a_oc;  str   = a_str;
cout<<"Octet (1): "<<oc<<" string (hello): "<<str<<endl;

// C. The implementation of the Any_GIDL::operator=
template<class T> void Any_GIDL::operator=(GIDL_Type<T>& b){
  T& a = dynamic_cast<T&>(b);
  if(!this->obj) this->obj = new CORBA::Any();
  T::_lift(this->obj, a);
}

// D. GIDL Arrays
interface Foo<T> {    //GIDL specification
    typedef T Array_T[100];
    T sum_and_revert(inout Array_T arr);
};
// C++ code using the GIDL specification above
Foo<Long_GIDL> foo = ...; Foo<Long_GIDL>::Array_T arr;
for(int i=0; i<100; i++) {
    Long_GIDL elem(i);    arr[i] =  elem;
}
int sum=foo.sum_and_invert(arr); Long_GIDL arr_0=arr[0];
cout<<"sum (4950): "<<sum<<" arr[0] (99): <<arr_0<<endl;
```

Fig. 24. GIDL/CORBA use of the `Any` type

Data Type	In	Inout	Out	Return
fixed struct	ct struct&	struct&	struct&	struct
var struct	ct struct&	struct&	struct&	struct*
fixed array	ct array	array	array	array sl*
var array	ct array	array	array sl*	array sl*
any	ct any&	any&	any*&	any*
...	...	...	...	...

Table 1. CORBA types for in, inout, out parameters and the result. `ct` = `const`, `sl` = `slice`, `var` = variable.

erasure of the non-qualified type-parameter `T` is the `Any` CORBA type. Although the user may expect that a statement like `arr[i] = i` inside the `for`-loop should do the job, this is not the case. The reason is that `Any_GIDL` does not provide an assignment operator or constructor that takes an `int` parameter.

Another simplification that GIDL brings refers to the types of the `in`, `inout` and `out` parameter, and the type of the result. Table 1 shows several of these types as specified in the CORBA standard. The GIDL parameter passing scheme is much simpler: the parameter type for `in` is `const T&`, for `inout` and `out` is `T&`, for the result is `T`, where `T` denotes an arbitrary GIDL type. The necessary type-conversions are hidden in the GIDL wrapper.

5.5 Type-Soundness Discussion

We restrict our attention to the wrapper-types corresponding to the GIDL interfaces. The same arguments apply to the rest of the wrapper-types. Let us examine the type-unsafe operations of the `BaseObject` class, presented in Figure 20. Note first that any function that receives a parameter of type `Any_GIDL` or `CORBA::Any` is unsafe, as the user may insert an object of a different type than the one expected. For example the `Leaf(const Any_GIDL& a)` constructor expects that an object of CORBA type `Leaf` was inserted in `a`: the user may decide otherwise, however, and the system cannot statically enforce it. It is debatable whether the introduction of generics to CORBA has rendered the existence of the `Any` type unnecessary in GIDL at the user level. We decided to keep it in the language for backward compatibility reasons. The drawback is that the user may manipulate it in a type-unsafe way.

In addition to these, there are two more unsafe operations:

```
template < class GG > operator GG() const { ... }
static T _lift (const CORBA::Object* o) { ... }
```

The templated cast operator is naturally unsafe, as it allows the user to cast to a more specialized type. The `_lift` method is used in the wrapper to lift an export-based qualified generic type object (`:-`), since its erasure is `CORBA::Object*`. Its use inside the wrapper is type-safe; however, if the user invokes it directly, it might result in type-errors.

Our intent is that the user access to the GIDL wrappers should be restricted to the constructors, the assignment and cast operators, and the functionality

```
// GIDL specification
interface Foo<T, I:-Test, E: Test> {
  Test foo(inout T t,inout I i,inout E e);
}
// Wrapper stub for foo
template<class T, class I, classE>
GIDL::Test Foo<T,I,E>::foo( T& t, I& i, E& e ) {
  CORBA::Any&     et  = T::_any_narrow(t);
  CORBA::Object*& ei  = I::_narrow(i);
  CORBA::Test*&   ee  = E::_narrow(e);
  CORBA::Test*    ret = getObjectFoo()->foo(et, ei, ee);
  return GIDL::Test::_lift(ret);
}
// Wrapper skeleton for foo
template<class T, class I, class E> ::Test Foo_Impl<T,I,E>::foo
( CORBA::Any& et, CORBA::Object*& ei, ::Test*& ee ) {
  T& t=T::_lift(et);   I& i=I::_lift(ei);   E& e=E::_lift(ee);
  GIDL::Test  ret = fooGIDL(t, i, e);
  return GIDL::Test::_narrow(ret);
}
```

Fig. 25. GIDL interface and the corresponding stub/skeleton wrappers for function `foo`

described in the GIDL specification, while the rest of the casting functionality should be invisible. However this is not possible since the `_narrow` and `_lift` methods are called in the wrapper method implementation to cast the parameters, and hence need to be declared public.

A *type-soundness* result is difficult to formalize as we are unaware of such results for (subsets of) the underlying CORBA architecture, and the C++ language is type-unsafe. In the following we shall give some informal soundness arguments for a subset of the GIDL bindings. We assume that the user can access only wrapper constructors and operators and only those that do not involve the `Any` type. The precise GADT interface guarantees that the creation of GIDL objects will not yield type-errors. It remains to examine method invocations. It is trivial to see from the implementation of the `_lift`, `_narrow`, and `_any_narrow` functions (Figure 20) that the following relations hold:

`G::_lift[A*]`∘`G::_narrow[G] (a)` ∼ `a`
`G::_lift[Object*]`∘`G::_narrow[G] (a)` ∼ `a`
`G::_lift[Any]`∘`G::_any_narrow[G] (a)` ∼ `a`

where `[]` is used for the method's signature, ∘ stands for function composition, while `g1`∼`g2` denotes that `g1` and `g2` are equivalent in the sense that they encapsulate the reference to the same CORBA object implementation. (The reverse also holds.)

Figure 25 presents the GIDL operation `Foo::foo()` and its C++ stub/skeleton mapping. The stub wrapper will translate the parameter to an object of the corresponding CORBA erased type via the `_narrow`/`_any_narrow` methods. The skeleton wrapper does the reverse: lifts a CORBA type object to a corresponding GIDL type object. Since the instantiations for the `T`, `I`, and `E` type parameters are the same on the client and server side, the above relations and the exact GADT casting interface guarantee that the GIDL object passed as parameter to the stub wrapper by the client will have the same type and will hold a reference to the same object-implementation as the one that is delivered to the `fooGIDL` server implementation method. The same argument applies to the result object.

```
public interface GIDL_Value_Interf
{
   public static org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
   public Object getOrigObj();
   public void setOrigObj(Object o);
   public GIDL_Value_Interf lift(Object o);
   public Object narrow(GIDL_Value_Interf o);
   public org.omg.CORBA.Any any_narrow(GIDL_Value_Interf go);
   public GIDL_Value_Interf any_lift(org.omg.CORBA.Any a);
}
```

Fig. 26. The root parent for the GIDL types

6 GIDL to Java Mapping

This chapter describes how the stub and skeleton wrappers are implemented when the targeted language is Java. The Java mapping is a step back compared with the C++ one, in the sense that the casting functionality between CORBA and GIDL types is not implemented in a type-exact manner. Work is in progress to address this shortcomming.

6.1 Wrapper Stub Object Model

The Java translation follows the same 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. This interface, presented in Figure 26, specifies the two-way (CORBA-GIDL) casting functionality. As Java supports covariant method overriding, subsequent specializations may refine the result type, but not the parameter types. For example the `Test` interface will export the `GIDL.Test any_lift(org.omg.CORBA.Any a)` method.

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 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.

Figure 27 shows a piece of the generated wrapper stub for the following GIDL specification.

```
interface Foo { /*...*/ };
```

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

Interface `Test` receives three type parameters. `T1` is bounded to be a subtype of the class `Foo`, and this qualification is supported in Java. `T2` is export-based qualified by class `Foo`, therefore the Java mapping requires `T2` to extend the most specific generic unifier corrsponding to class `Foo` (see Section 4.3). `T3` is an unqualified type parameter, and thus the Java mapping requires that it extends from the root interface for all GIDL types `GIDL_Value_Interf`. As discussed above, the constructor of the `Foo` class receives, besides the CORBA reference, three additional parameters of types `T1, T2, and T3`.

The implementation of the `op` function (in Figure 27) 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 `//4`), a *unqualified* generic type is erased to the `CORBA.Any` type (line `//2`), an *extend-based qualified* generic type is unboxed to the IDL-stub type associated with its qualifier (line `//1`) and finally an *export-based qualified* generic type is erased to the `CORBA::Object` type (line `//3`). The IDL stub method is invoked on the object reference that our wrapper encapsulates (line marked 5) and finally the returned CORBA stub object/value is boxed inside a stub wrapper object and is returned to the client application (line `//6`). The additional parameter `obj_T1_GIDL` is used in this last instruction to create the result value. This is necessary because `new T1();` is not a valid instruction in Java.

6.2 Mapping GIDL Structures to Java

This section presents how the GIDL structures that are nested in the scope of a generic interface are translated to Java. An example of such a structure that uses the generic types of the enclosing interface is shown below.

```
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. These additional generic types will parameterized the Java class that corresponds to the GIDL structure.

The Java mapping for the `BaseStruct` parameterized structure, presented above is shown in Figure 28. 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

```
public class _TestStub <
    T1 extends GIDL.Foo, T2 extends GIDL.MGGUs.MGGU_Foo_0,
    T3 extends GIDL.GIDL_Value_Interf
> implements Test<T1,T2,T3> {
    protected defaultpkg.Test obj;
    private T3 obj_T3_GIDL;
    private T2 obj_T2_GIDL;
    private T1 obj_T1_GIDL;

    public _TestStub(defaultpkg.Test ob, T1 obj1, T2 obj2, T3 obj3)
    { obj = ob; obj_T1_GIDL = obj1; obj_T2_GIDL = obj2; obj_T3_GIDL = obj3; }

    public defaultpkg.Test getOrigObj()
    { return defaultpkg.TestHelper.narrow(obj); }

    public Test<T1,T2,T3> any_lift(org.omg.CORBA.Any a) {
        try {
            defaultpkg.Test ob = defaultpkg.TestHelper.extract(a);
            return (new _TestStub<T1,T2,T3>(
                (defaultpkg.Test)ob, obj_T1_GIDL, obj_T2_GIDL, obj_T3_GIDL)
            );
        }catch(Exception exc)
        { System.out.println(exc); return null; }
    }

    /** any_narrow, narrow, setOrigObj methods are not presented here **/
    /** ... **/

    public GIDL.Foo op(
        T1 a1_GIDL,
        T2 a2_GIDL,
        T3 a3_GIDL,
        GIDL.Foo a4_GIDL
    ) {
        defaultpkg.Foo a1 = a1_GIDL.narrow(a1_GIDL);        // 1
        org.omg.CORBA.Any a3 = a3_GIDL.any_narrow(a3_GIDL); // 2
        org.omg.CORBA.Object a2 = a2_GIDL.narrow(a2_GIDL);  // 3
        defaultpkg.Foo a4 = a4_GIDL.narrow(a4_GIDL);        // 4

        defaultpkg.Foo a0_GIDL = obj.op( a1, a2, a3, a4);   // 5

        return (T1)obj_T1_GIDL.lift(defaultpkg.FooHelper.narrow(a0_GIDL)); // 6
    }
}
```

Fig. 27. Excerpt of Java wrapper stub code

```
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 {
  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); }
}
```

Fig. 28. Java mapping for a GIDL nested structure

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.

7 Installation

We assume you have already downloaded the *gidl.tar.gz* file. Decompress the *gidl.tar.gz* file:

```
$ tar -zxvf gidl.tar.gz
```

The environment variable *GIDL_INSTALL* should point to the folder that is obtained as a result of the previous operation. *GIDL_INSTALL* contains three directories: *Doc*, *GIDLcompiler* and *Tests*. The *Doc* folder contains this document, and the GIDL relevant papers. The *GIDLcompiler* folder contains the sources and the instalation of the GIDL compiler, together with that part of the language bindings that is static (not generated by the GIDL compiler). The latter rezides in the *GIDLcompiler/Include* folder.

To proceed with the instalation type:

```
$ cd GIDLcompiler/src
$ ./compileGIDL.sh
```

The GIDL compiler will be built. Ignore the warning messages:

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

If you have not already done so, it is time to download and install the Mico and Orbacus CORBA implementations. The environment variables *ORBACUS_PATH* and *MICO_PATH* are to point to the instalation folders of these architectures. Similarly, the environment variables *JAVA_PATH* and *GCC_PATH* point to the instalation folders of Java and C++, respectivelly. All the C++ tests for GIDL were build under `gcc2.95.3`. Finally, define the following environment variables:

```
export GIDL_INCLUDE=$GIDL_INSTALL/GIDLcompiler/Include
export GIDL_MAIN=$GIDL_INSTALL/GIDLcompiler
export TLS_JAVA_PATH=$GIDL_INSTALL/GIDLcompiler/DTLS/classes
```

A sample of my `.bashrc` file is placed in *GIDL_INSTALL/dotbashrc*.

```
interface Comparable< T > {
    boolean    operator">" (in T k);    boolean    operator"=="(in T k);
};

interface Integer : Comparable<Integer>
{   long getValue();                    };

interface BinTree< K:-Comparable<K>, D > {
    D    getData();    K    getKey();    D    find(in K k);
};

interface Leaf< K:-Comparable<K>, D > : BinTree<K,D> { };
interface Node< K:-Comparable<K>, D > : BinTree<K,D> {
    BinTree<K,D>    getLeftTree();    BinTree<K,D>    getRightTree();
};

interface TreeFactory<K:-Comparable<K>, D> {
    Integer createInteger(in long val);
    BinTree<K,D>createLeaf(in K k, in D d);
    BinTree<K,D>createNode( in K k, in D d, in BinTree<K,D> right,
                 in BinTree<K,D> left );
};
```

Fig. 29. GIDL ADT-like specification of a binary tree

8 Examples

The `Tests` folder contains several examples of GIDL applications. Based on these examples we demonstrate how to use GIDL to write multi-language, distributed programs that make use of parametric polymorphism.

8.1 Abstract Data Type (ADT)-like binary tree

The *CppBinTreeADT* folder contains C+ code that implements a simple binary tree server, together with a C++ and Java client programs that use the server's functionality.

GIDL Interface for a Binary Tree

Figure 29 presents the content of the GIDL specification file *BinTree.tidl*. The `BinTree`, `Leaf` and `Node` interfaces implement the functionality of a binary tree. They are type-parameterized under the types of keys (`K`) and data (`D`) stored in the nodes/leaves. Each interface defines a number of operations that are guaranteed to be serviced by a server-implementation. As defined by CORBA-IDL, there are three parameter-passing conventions: The *in* parameter passing means call-by-value, *inout* parameter-passing is call-by-value-return, while *out* parameter passing does not expect the parameter to be a valid reference to an object-implementation.

We remind the reader that GIDL introduces a parametric polymorphism model that 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 was introduce to address code extensibility and re-usability issues, allowing the programmer to design a clean and precise specification, and to avoid unnatural inheritance relations between interfaces. Following the same approach as in Generic Java [10, 15], 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>`.

The type-variable `K` in the definition of the `BinTree` interface is an example of *export-based* qualification. A valid instantiation for `K`, say `Integer`, is required to implement the whole functionality of its qualifier, in this case `Comparable<Integer>`. In our case, this functionality consists of the comparison operations `>` and `==` that receive as parameter an object of type `Integer` and return a `boolean` value. Note also that the definition of the `K` parameter uses F-bounded quantification: the type parameter appears in the expression of its qualifier `Comparable<K>`.

The *extension-based* definition for the type-parameter `K` is denoted by `K : Comparable<K>`, with the semantics that the instantiation of `K`, say `Integer` is required to be in a subtyping relation with its qualifier:

```
Integer<:Comparable<Integer>.
```

GIDL also supports *unqualified* generic types, similar to templates in C++ (e.g. `K` in the definition of the `Comparable` interface). This allows the instantiation to be any GIDL type.

To wrap it up, Figure 29 specifies the interface of a binary tree. A binary tree can be a node, or a leaf; note that both `Node` and `Leaf` interfaces inherit from `BinTree`. Nodes and leafs implement the `getData`, `getKey`, and `find` services. The first two return the data/key stored in that tree. The latter finds the data of a child-node identified by a given key. In addition, a node implements the `getLeftTree` and `getRightTree` services that return its left and right child respectivelly. The `Integer` interface constitutes a valid instantiation for the `K` parameter in the definition of the `BinTree` interface. Finally the `TreeFactory` interface provides the means to construct nodes and leafs.

Compiling and Running the Binary Tree Example

The first time you run the GIDL compiler on a given GIDL specification you should start it in graphical mode:

```
java GIDLcompiler -GI BinTree.tidl &
```

Check the `idl`, `C++` and `Java` check boxes such that the compiler will generate both C++ and Java stubs. In the *Paths* tab fill in the paths where you want the language-specific stubs to be generated. Choose the current folder for the

```
 Make Leafs: b6 = (6,6), b8 = (8,8), b10 = (10,10), b13 = (13,13)
 Make Nodes: b7 = (7,7,b6,b8), b11 = (11,11,b10,b13), b9 = (9,9,b7,b11)
 b9: key: 9 data: 9
 Find in b9 key 8 with data: 8
END
```

Fig. 30. Client output for the binary tree example

idl file, the `./Cpp/./Java` folder for the C++/Java stubs. The information in the *Java naming/C++ naming* tabs teaches GIDL specific naming conventions of the underlying CORBA implementation. The *operators* tab allows the user to refine the default naming scheme for operator overloading. You can find two examples of such configuration files in the `$GIDL_INCLUDE` folder. As this example uses operator overloading, select the `OperatorsMappingCpp.txt` file, in the `$GIDL_INCLUDE` folder, for the C++ language. Press the `compile` button to generate the Java/C++ GIDL stubs. The next time you run the GIDL compiler for the same example, for example if you modify the specification, you need not start the graphical interface again. You can just type:
`java GIDLcompiler -i -j -c BinTree.tidl`
where `-i`,`-j`,`-c` stand for generate the IDL erased file, the Java/C++ GIDL stubs.

The IDL erased file `BinTree_erased_GIDL.idl` has been generated in the current directory (`CppBinTreeADT`). Copy it to the `Cpp` folder and `Java` folder and compile it with an `idl` compiler for C++ and Java, respectively. For example with MICO for C++ type in:
`Cpp$ idl --any --poa BinTree_erased_GIDL.idl`
while for Java type in
`Java$ jidl BinTree_erased_GIDL.idl`
You will find a `server.cpp` file that implements the binary tree in the `Cpp` folder and a `client.cpp` and a `client.java` in the `Cpp/Java` folders, which implement a simple client that creates a tree, looks for a node that is associated with a specific key, and returns the data corrsponding to that node/leaf. The `Cpp` folder also contains the `compile_client` and `compile_server` scripts that generate the client and server executables. Modify them to fit your working environment. To compile the Java client, just go in the `Java` folder and type
`$javac client.java`.
Now start the C++ server (`$./server`) with either the C++ client or the Java client. The output is shown in Figure 30.

Programming the Client and the Server

Figure 31 shows an excerpt of C++ client code that performs operations on a binary tree. The code first reads the CORBA server reference stored in the file `../Factory.ref`. (The server is the one that generates this file.) Then the CORBA function `string_to_object` is employed to build an object that acts as

```
int run(CORBA::ORB_ptr orb) {
    const char* refFile = "../Factory.ref";    ifstream in(refFile);
    char s[2048];    in >> s;    cout<<"Ref is:\n "<<s<<endl;
    CORBA::Object_var obj = orb->string_to_object(s);
    ::TreeFactory_var f = ::TreeFactory::_narrow(obj);

    GIDL::TreeFactory<GIDL::Integer, GIDL::Integer> factory(f);
    typedef GIDL::BinTree<GIDL::Integer, GIDL::Integer> BinIntTree;
    GIDL::Integer i6 = factory.createInteger(6),
                  i6d = factory.createInteger(6),
    /** same for i7, i7d, i8, i8d, i9, id9, i10, i10d, i11, i11d ... **/
    BinIntTree b6  = factory.createLeaf(i6, i6d);
    /** same for leafs b8, b10, b13 **/
    BinIntTree b7  = factory.createNode(i7, i7d, b6, b8);
    /** same for nodes b9 and b11 **/

    /** find the node corrsponding to the key 8 in the tree b9 **/
    GIDL::Integer res = b9.find(i8);
    if(!is_nil(res))
         cout<<"Find key 8 with data: "<<(int)res.getValue()<<endl;
    else cout<<"Not found"<<endl;
```

Fig. 31. Client output for the binary tree example

a proxy for the server. In our case the CORBA object's type is: `::TreeFactory`. So far we have used only CORBA functionality. To work with generic types, we next create a GIDL object that encapsulates the generic type information:
`GIDL::TreeFactory<GIDL::Integer, GIDL::Integer> factory(f);`
Note that the current version of GIDL does not support type-parameterized functions, as the mechanism to support them would involved run-time recompilation. Therefore we assume that the user knows the precise type of the server object, in our case `TreeFactory<Integer,Integer>`. From this point on, the user works with GIDL objects of exact (templated) types. The client code uses the functionality described in the GIDL interface to build a binary tree, by creating the appropriate nodes and leafs. Finally the client calls the `find` function on the binary tree object `b9` to get the data associated with the node identified by a key of value `9` and prints out the result. The Java client code is similar (take a look at the `Java/client.java` file).

Figure 32 shows the implementation of the `BinTree` GIDL interface. Every GIDL implementation should inherit the corrsponding GIDL skeleton class, in this case `POA_GIDL::BinTree<K,D>`, and implement the functionality described in the GIDL interface: the `getKey`, `getData`. The `find` function is declared abstract, to be implemented in the `Leaf` and `Node` classes. Note that CORBA-IDL does not support method overloading/overriding. GIDL alleviate this shortcomming to some degree by supporting method and operator overloading. However, method

```
template<class K, class D>
class BinTree_Impl : public virtual POA_GIDL::BinTree<K,D>,
    public virtual ::PortableServer::RefCountServantBase {
  protected:
    K key;
    D data;
  public:
    BinTree_Impl() {  }
    virtual K getKeyGIDL() throw(CORBA::SystemException) { return key; }
    virtual D getDataGIDL()throw(CORBA::SystemException) { return data;s}
    virtual D findGIDL( K& a1_GIDL )throw(CORBA::SystemException)=0;
};
```

Fig. 32. Server implementation for the BinTree GIDL interface

overriding is not supported. This is because, virtual dispatch can be achieved at the implementation (server) level if desired, by declaring the given function abstract in the parent and implementing it in the child classes.

```
int run(CORBA::ORB_ptr orb) {
   CORBA::Object_var poaObj = orb -> resolve_initial_references("RootPOA");
   PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(poaObj);
   PortableServer::POAManager_var manager = rootPoa -> the_POAManager();

   GIDLImplem::TreeFactory_Impl<GIDL::Integer, GIDL::Integer>* fact_impl =
new GIDLImplem::TreeFactory_Impl<GIDL::Integer, GIDL::Integer>();
   ::TreeFactory* factory = fact_impl->_thisGIDL();

   CORBA::String_var s = orb -> object_to_string(factory);
   const char* refFile = "../Factory.ref";
   ofstream out(refFile);   out << s << endl; out.close();

   manager -> activate(); orb -> run(); return EXIT_SUCCESS;
}
```

Fig. 33. Server program

Finally, Figure 33 shows the program that runs the server. The first three lines in the `run` function create an instance of an *portable server inheritance model* (POA) manager. Then a factory server object corresponding to the GIDL type `TreeFactory<Integer; Integer>` is created and its corresponding CORBA reference is published in the `../Factory.ref` file, in order to be used by the client. Finaly, the POA manager is activated and the object request broaker (ORB) may now service client requests.

8.2 Standard Template Library Mapping

The folder CppSTL contains the code that exports part of the C++ Standard Template Library (STL) to a heterogeneous environment via GIDL. The IDLspec folder contains the GIDL specifications, while the Stubs folder contains the C++/Java client and the C++ server.

Compiling and Running the STL Example

First, go into the IDLspec folder, and run the GIDL compiler in graphical mode:

```
java GIDLcompiler -GI STLfactory.tidl &
```

Check the idl, C++ and Java check boxes such that the compiler will generate both C++ and Java stubs. In the *Paths* tab, choose the CppSTL/Stubs/CppServer folder as the location where the compiler will generate the C++ stubs. Similarly, choose the current folder for the *idl* file, and choose CppSTL/Stubs/JavaClient for Java. As this example uses operator overloading, in the Operators tab, select the OperatorsMappingCpp.txt file for the C++ language. The latter is located in the $GIDL_INCLUDE folder, for the C++ language. Press the compile button to generate the Java/C++ GIDL stubs. The next time you run the GIDL compiler for this application, for example if you modify the specification, you need not start the graphical interface again. You can just type:

```
java GIDLcompiler -i -j -c STLfactory.tidl
```

where -i,-j,-c stand for generate the IDL erased file, the Java/C++ GIDL stubs.

The IDL erased file STLfactory_erased_GIDL.idl has been generated in the current directory (IDLspec). Copy it to the ../Stubs/CppServer/, ../Stubs/CppClient, and ../Stubs/JavaClient folders. Go in each of these folders and compile the idl file:

```
CppServer$ idl --any --poa BinTree_erased_GIDL.idl
CppClient$ idl --any --poa BinTree_erased_GIDL.idl
JavaClient$ jidl BinTree_erased_GIDL.idl
```

for C++ and Java respectively.
Edit the CppSTL/Stubs/CppServer/STLfactory_erased_GIDLGIDL.h and replace the following code in the BaseIterator_GIDLP class:

```
typedef T value_type;
typedef GIDL::UnsignedLong_GIDL size_type;
typedef GIDL::Long_GIDL difference_type;
```

with this one below:

```
typedef T value_type;
typedef unsigned int  size_type;
typedef int  difference_type;
typedef input_iterator_tag    iterator_category;
typedef T*  pointer;
typedef T&  reference;
```

This is needed in order to allow the GIDL iterators to be valid STL iterators. Next go to the `CppSTL/Stubs/CppServer` folder and copy the `STLfactory_erased_GIDLGIDL.h` file into the `CppSTL/Stubs/CppClient` folder. Go into each of these folders (`CppServer`, `CppClient`, `JavaClient`) and modify the `compile_client`/`compile_server` scripts to suit your working environment. You can now run the C++ server with either the Java or C++ client.

```
CppServer$ ./server
CppClient$ ./client
JavaClient$ java client
```

STL translation discussion

Technical details about translating STL to a heterogeneous environment can be found in the *Generic Library Extension in a Heterogeneous Environment*, by Oancea and Watt [7]. The `../CppSTL/Stubs/LibTransl.h` file implement the library wrappers and the trappers that enable the STL library semantics and programming idioms at the GIDL level.

Note that the client code is quite close to the STL programming style. Some casts to the C++ basic types are still needed though. The code below reads the first ten elements of an iterator and set them to $10 + i$ where i goes from 0 to 10.

```
isOK = true;
for(int i=0; i<10; i++) {
    int elem_i = (int)gidl_vect[i];
    if(elem_i!=i) isOK = false;
    gidl_vect[i] = i + 10;
}
```

The code below demonstrates how the use of the `replace` algorithm. The iterator elements which are equal to 19 are replaced with 99.

```
rai_Long beg = gidl_vect.begin();
rai_Long end = gidl_vect.end();
rai_Long tmp = beg;
replaceAlg.replace(beg, end, 19, 99);
while(tmp!=end) {
    int elem = (int)*tmp++;
    if(elem == 19) cout<<"ERROR"<<endl;
}
```

8.3 Comprehensive Example

The last GIDL example can be found in the `TestGIDL` folder. Its purpose is to demonstrate how to use various GIDL types: array, sequences, structures, interfaces. The folder `TestGIDL/Cpp` contains a C++ client and server, while the folder `TestGIDL/Java` contains a Java client and server.

Folow the same procedure described in detail for the latter two examples to compile the C++/Java client and server. Run the java client with the Java server, and the C++ client with the C++ server. You should get a list of prints of the form:

```
After calling  op...: OK
Before calling op...: OK
```

You may try to run the Java server with the C++ client, or vice-versa. You will note that: First, the Java/C++ server/client are incompatible in the sense that they are not doing the same thing. The consequence is that the printouts will not look like the one above. This is to be expected! Second, the Java/C++ client may broke. This is because of incompatibilities between the underlying CORBA implementations (passing `out` parameters, for example).

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. Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, pages 119–130. Mirton Publishing House, 2004.
3. J. Farley. *Java Distributed Computing.* O'Reilly, 1998. "Wiley computer publishing.".
4. 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.
5. K. Keahey. A Brief Tutorial on CORBA, http://www.cs.indiana.edu/ kksiazek/tuto.html.
6. C. E. Oancea and S. M. Watt. Parametric Polimorphism for Software Component Architectures. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 147–166, 2005.
7. C. E. Oancea and S. M. Watt. Generic Library Extension in a Heterogeneous Environment. In *Library Centric System Design Workshop (LCSD'06)*, 2006.
8. M. Odersky and al. Technical Report IC 2004/64, an Overview of the Scala Programming Language. Technical report, EPFL Lausanne, Switzerland, 2004.
9. M. Odersky, V. Cremet, C. Rockl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of ECOOP'03*, July 2003.
10. M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, 1998.
11. OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision2.4 (October 2000), OMG Specification, 2000.
12. OMG. Common Object Request Broker: Architecture and Specification. Revision2.4 (October 2000), OMG Specification, 2000.

13. G. D. Plotkin. A Note on Inductive Generalization. In *Machine Intelligence*, pages 153–163, 1970.
14. J. C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence, 5(1)*, pages 135–151, 1970.
15. 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.