Generic Library Extension in a Heterogeneous Environment

Cosmin Oancea Stephen M. Watt

Department of Computer Science
The University of Western Ontario
London Ontario, Canada N6A 5B7
{coancea,watt}@csd.uwo.ca

Abstract

We examine what is necessary to allow generic libraries to be used naturally in a heterogeneous environment. Our approach is to treat a library as a software component and to view the problem as one of component extension. Language-neutral library interfaces usually do not support the full range of programming idioms that are available when a library is used natively. We address how language-neutral interfaces can be extended with import bindings to recover the desired programming idioms. We also address the question of how these extensions can be organized to minimize the performance overhead that arises from using objects in manners not anticipated by the original library designers. We use C++ as an example of a mature language, with libraries using a variety of patterns, and use the Standard Template Library as an example of a complex library for which efficiency is important. By viewing the library extension problem as one of component organization, we enhance software composibility, hierarchy maintenance and architecture independence.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.2.2 [*Software Engineering*]: Modules and Interfaces, Software Libraries

General Terms Languages, Design

Keywords Generalized algebraic data types, Generics, Parametric Polymorphism, Software Component Architecture, Templates

1. Introduction

Library extension is an important problem in software design. In its simplest form, the designer of a class library must consider how to organize its class hierarchy so that there are base classes that library clients may usefully specialize. More interesting questions arise when the designers of a library wish to provide support for extension of multiple, independent dimensions of the library's behavior. In this situation, there are questions of how the extended library's hierarchy relates to the original library's hierarchy, how objects from independent extensions may be used and how the extensions interact.

This paper examines the question of library extension in a heterogeneous environment. We consider the situation where software libraries are made available as components in a multi-language, potentially distributed environment. In this setting, the programmer finds it difficult and rather un-safe to compose libraries based on low level language-interoperability solutions. Therefore, components are usually constructed and accessed through some framework such as CORBA [14], DCOM [6] or the .NET framework [5]. In each case, the framework provides a language-neutral interface to a constructed component. These interfaces are typically simplified versions of the implementation language interface to the same modules because of restrictions imposed by the component framework. Restrictions are inevitable: Each framework supports some set of common features provided by the target languages at the time the framework was defined. However, programming languages and our understanding of software architecture evolves over time, so mature component frameworks will lack support for newer language features and programming styles that have become common-place in the interim. If a library's interface is significantly diminished by exporting it through some component architecture, then it may not be used in all of the usual ways that those experienced with the library would expect. Programmers will have to learn a new interface and, in effect, learn to program with a new library.

We have described previously the Generic Interface Definition Language framework, GIDL [8], a CORBA IDL extension with support for parametric polymorphism and (operator) overloading, which allows interoperability of generic libraries in a multi-language environment. GIDL is designed to be a *generic* component architecture *extension*. Here "generic" has two meanings: First GIDL encapsulates a common model for parametric polymorphism that accommodates a wide spectrum of requirements for specific semantics and binding times of the supported languages: C++, Java, and Aldor [16]. Second, the GIDL framework can be easily adapted to work on top of various IDL-based component-systems in use today such as CORBA, DCOM, JNI [15].

This paper 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 enable the GIDL software components to reproduce as much of their original native language interfaces as possible, and to do so without introducing significant overhead. This allows programmers familiar with the library to use it as designed. In these contexts, we identify the language mechanisms and programming techniques that foster a better code structure in terms of interface clarity, type safety, ease of use, and performance.

While our earlier work [8] presented the high-level ideas employed in implementing the GIDL extension mechanism, this paper takes a different perspective, in some way similar to that of Odersky and Zenger. In [11], they argue that one reason for inadequate advancement in the area of component systems is the fact that mainstream languages lack the ability to abstract over the required ser-

LCSD'06 October 22, 2006, Portland, Oregon, USA.

vices. They identify three language abstractions, namely *abstract type members*, *selftype annotations*, and *modular mixin composition* that enable the design of first-class value components (components that use neither static data nor hard references).

We look at the GIDL extension as a component that can be employed on top of other underlying architectures and which can be, at its turn, further extended. Consequently, we identify the following as desirable properties of the extension:

- The extension interface should be type-precise and it should allow type-safety reasoning with respect to the extension itself. The type-safety result for the whole framework would thus be derived from the ones of the extensions and of the underlying architecture.
- The extension should be split in first-class value components. In the GIDL case for example, one component should encapsulate the underlying architecture specifics and be statically generated. The other one should generically implement the extension mechanism. This would allow GIDL to be plugged in with various backend-architectures without modifying the compiler.
- The extension should preserve the look and feel of the underlying architecture, or at least not complicate its use.
- The extension overhead should be within reasonable limits, and there should be good indication that compiler techniques may be developed to eliminate it.

In the context of GIDL's C++ bindings, we identify the language concepts and programming strategies that enable a better code structure in the sense described above. We particularly recognize *the generalized algebraic data types* paradigm [17] to be essential in enforcing a clear and concise meta-interface of the extension. In agreement with [11], we also find that the use of (C++ simulated) *abstract type members*, and *traits* allows the extension to be split into first-class value components. This derives the obvious software maintenance benefits.

The second part of this paper reports on an experiment where we have used GIDL to export part of the C++ Standard Template Library (STL) functionality to a multi-language, potentially distributed use. We had two main objectives:

The first objective was to determine to what degree the interface translation could preserve the coding style "look and feel" of the original library. Ideally, the STL and its GIDL-exported programs should differ only in the types used. This allows the STL programmers to easily "learn" to use the GIDL interface to write for example distributed applications. More importantly, this opens the door to a richer composition between GIDL and STL objects, as enabled by the STL orthogonal design of its domains. For example GIDL iterators are themselves valid STL iterators and thus they can be manipulated by the STL containers and algorithms. In this context we investigate the issues that prevent the translation to conform with the library semantics, the techniques to amend them, and the trade-offs between translation ease-of-use and performance.

The second objective was to determine whether the interface translation could avoid introducing excessive overhead. We show how this can be achieved through the use of various helper classes that allow the usual STL idioms to be used, while avoiding unnecessary copying of aggregate objects.

The rest of the paper is organized as follows. Section 2 briefly recalls the GADT programming technique, and gives a high-level review of the GIDL framework. Section 3 presents the rationale for employing GADT-based techniques to extend existing frameworks, and outlines the issues to be addressed when translating the STL library to a heterogeneous environment. Section 4 describes the design of the GIDL bindings for the C++ language. Section 5 describes the "black-box" type translation of the STL library to a multi-language, distributed environment via GIDL and discusses certain usability/efficiency trade-offs. Finally Section 6 presents some concluding remarks.

```
data Exp t where
    Lit      :: Int      -> Exp Int
    Plus     :: Exp Int  -> Exp Int -> Exp Int
    Equals   :: Exp Int  -> Exp Int -> Exp Bool
    Fst      :: Exp(a,b) -> Exp a
eval :: Exp t -> t
eval e = case e of
    Lit    i     -> i
    Plus   e1 e2 -> eval e1 + eval e2
    Equals e1 e2 -> eval e1 == eval e2
    Fst    e     -> fst (eval e)
```

Figure 1. GADT-Haskell interpreter example.

```
public class Pair<A,B>                              { /* ... */ }
public abstract class Exp<T>                        { /* ... */ }

public class Lit : Exp<int>
{   public Lit(int val)                         { /* ... */ }   }
public class Plus : Exp<int>
{   public Plus(Exp<int> a, Exp<int> b)         { /* ... */ }   }
public class Equals : Exp<bool>
{   public Equals(Exp<int> e1, Exp<int> e2)     { /* ... */ }   }
public class Fst<A,B> : Exp<A>
{   public Fst(Exp<Pair<A,B>> e)                { /* ... */ }   }
```

Figure 2. GADT-C# interpreter example.

2. Background

The first subsection of this chapter introduces at a high-level the *generalized algebraic data types* [17, 4] (GADT) concept and illustrates its use through a couple of examples. The second subsection briefly recounts the architectural design of the GIDL framework and the semantics of the parametric polymorphism model it introduces. A detailed account of this work is given elsewhere [8].

2.1 Generalized Algebraic Data Types

Functional languages such as Haskell and ML support generic programming through user-defined (type) parameterized algebraic datatypes (PADTs). A datatype declaration defines both a named type and a way of constructing values of that type. For example a binary tree datatype, parameterized under the types of the keys and values it stores, can be defined as below.

```
data BinTree k d =  Leaf k d |
                    Node k d (BinTree k d) (BinTree k d)
```

Both value constructors have the generic result type `BinTree k d`, and any value of type `BinTree k d` is either a leaf or a node, but it cannot be statically known which. `BinTree` is an example of a *regular* datatype since all its recursive uses in its definition are uniformly parameterized under the parametric types `k` and `d`.

Generalized algebraic data types (GADTs) enhance the functional programming language PADTs by allowing constructors whose results are instantiations of the datatype with other types than the formal type parameters. Figure 1 presents part of the definition of the types needed to implement a simple language interpreter. Note that all the type-constructors (`Lit`, `Plus`, `Equals`, and `Fst`) refine the type parameter of `Exp`, and use the `Exp` datatype at different instantiations in the parameters of each constructor. Also `Fst` uses the type variable `B` that does not appear in its result type. These are recognized as attributes of the GADT concept; its usefulness is illustrated by the fact that one can now write a well-typed evaluator function (`eval`). The example is inspired from [4] and is written in an extension of Haskell with GADTs.

Kennedy and Russo[4] show, among other things, that existing object oriented programming languages such as Java and C# can express a large class of GADT programs through the use of generics, subclassing and virtual dispatch. A C# implementation of the interpreter using GADTs is sketched in Figure 2.

```
/********************** GIDL interface **********************/
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(); };
interface TreeFactory<K:-Comparable<K>, D> {
  Integer       mkInt(in long val);
  BinTree<K,D>  mkLeaf(in K k, in D d);
  BinTree<K,D>  mkNode
  (in K k, in D d, in BinTree<K;D> right, in BinTree<K;D> left);
};
/********************** C++ client code **********************/
TreeFactory<Integer, Integer> fact(...);  // get a factory object
Integer  i6=fact.mkInt(6),  i7=fact.mkInt(7), i8=fact.mkInt(8);
BinTree<Integer, Integer>     b6=fact.mkLeaf(i6,i6),
          b8=fact.mkLeaf(i8,i8), tree=fact.mkNode(i7,i7,b6,b8);
int      res = tree.find(i8).getValue();  // 8
```

Figure 3. GIDL specification and C++ client code for a binary tree

2.2 The GIDL Framework

The Generic Interface Definition Language framework [8] (GIDL for short) is designed to be a *generic* component architecture extension that provides support for parameterized components and that can be easily adapted to work on top of various software component architectures in use today: CORBA, DCOM, JNI. (The current implementation is on top of CORBA). We summarize the GIDL model for parametric polymorphism in Section 2.2, and briefly describe the GIDL architecture in Section 2.2. An in depth presentation of these topics can be found in [8].

The GIDL language

GIDL extends CORBA–IDL [12] language with support for *F-bounded parametric polymorphism*. Figure 3 shows abstract data type (ADT)-like GIDL interfaces for a binary tree that is type-parameterized under the types of data and keys stored in the nodes. The type-parameter `K` in the definition of the `BinTree` interface is qualified to export the whole functionality of its qualifier `Comparable<K>`; that is, the comparison operations `>` and `==`. GIDL also supports a stronger qualification denoted by `:` that enforces a subtyping relation between the instantiation of the type parameter and the qualifier. Figure 3 also presents C++ client code that builds a binary tree and finds in the tree the data of a node that is identified through its key. Note that the code is very natural for the most parts; the only place where CORBA specifics appear is in the creation of the factory object (`fact`).

The GIDL Extension Architecture

Figure 4 illustrates at a high level the design of the GIDL framework. The implementation employs a generic type erasure mechanism, based on the subtyping polymorphism supported by IDL. A GIDL specification compiled with the GIDL compiler generates an IDL file where all the generic types have been *erased*, together with GIDL wrapper stub and skeleton bindings, which recover the lost generic type information. Currently GIDL provides language bindings for C++, Java, and Aldor. Compiling the IDL file creates the underlying architecture (UA) stub and skeleton bindings. Every GIDL-stub (client) wrapper object references a UA-stub object. Every GIDL-skeleton (server) wrapper inherits from the corresponding UA-skeleton type. This technique is somewhat related with the "reified type" pattern of Ralph Johnson [3], where objects are used to carry type information.

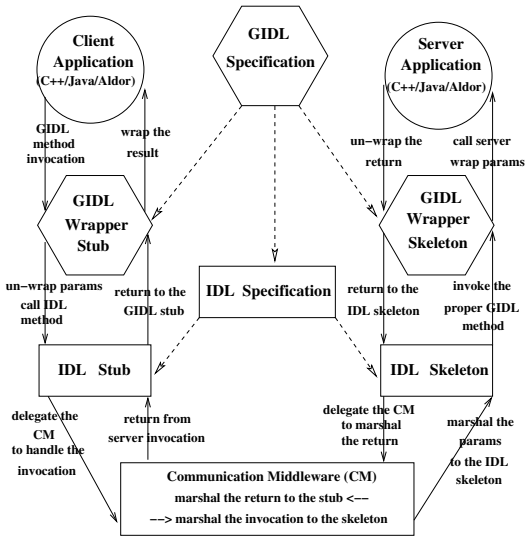

Figure 4. GIDL architecture
circle – user code; hexagon – GIDL component;
rectangle – underlying architecture component;
dashed arrow – is compiled to;
solid arrow – method invocation flow

The solid arrows in Figure 4 depict method invocation. When a method of a GIDL stub wrapper object is called, the implementation retrieves the parameters' UA-objects, invokes the UA method on these, and perform the reverse operation on the result. The wrapper skeleton functionality is the inverse of the client. The wrapper skeleton method creates GIDL stub wrapper objects encapsulating the UA objects, thus recovering the generic type erased information. It then invokes the user-implemented server method with these parameters, retrieves the UA IDL-object or value of the result and passes it to the IDL skeleton.

The extension introduces an extra level of indirection with respect to the method invocation mechanism of the underlying framework. This is the price to pay for the generality of the approach: this generic extension will work on top of any UA vendor implementation while maintaining backward compatibility. However, since the GIDL wrappers are mainly storing generic type information, one can anticipate that the introduced overhead can be eliminated by applying aggressive compiler optimizations.

3. Problems Statement and High-Level Solutions

This section states and motivates the main issues addressed by this paper, and presents at the high-level the methods employed to solve them: Section 3.1 summarizes the rationale and the techniques we have used to structure the GIDL language bindings. Section 3.2 outlines the main difficulties a heterogeneous translation of the STL library has to overcome, and points to a solution that preserves the library semantics and programming patterns.

3.1 Software Extensions via GADTs

Among the GADTs applications, the literature enumerates: strongly typed evaluators, generic pretty printing, generic traversal and queries and typed LR parsing. This paper finds another important application of the GADT concept: in the context of software architecture extensions. This section describes things at a high-level, while Section 4 presents in detail the C++ binding.

Section 2.2 has introduced GIDL as a *generic extension framework* that enhances CORBA with support for parametric polymorphism. The GIDL wrapper objects can be seen as an aggregation of

```
class Foo_CORBA {    /* ... */    }
class Foo_GIDL  {
    Foo_CORBA obj;   /* ... */
    Foo_CORBA  getOrigObj ()           { return obj; }
    void       setOrigObj (Foo_CORBA o)        { ... }
    static Foo_CORBA  _narrow    (Foo_GIDL o)  { ... }
    static Foo_GIDL   _lift      (Foo_CORBA o) { ... }
    static Foo_GIDL   _lift      (CORBA_Any a) { ... }
    static CORBA_Any  _any_narrow(Foo_GIDL a)  { ... }
}
```

Figure 5. Pseudocode for the casting functionality of the `Foo_GIDL` GIDL wrapper. `Foo_CORBA` is its corresponding CORBA class. `CORBA_Any`-type objects can store any CORBA-type values.

```
class Base_GIDL<T_GIDL, T_CORBA> {
    T_CORBA  getOrigObj ()           { return obj; }
    void     setOrigObj (T_CORBA o)          { ... }
    static T_CORBA   _narrow    (T_GIDL o)   { ... }
    static T_GIDL    _lift      (T_CORBA o)  { ... }
    static T_GIDL    _lift      (CORBA_Any a){ ... }
    static CORBA_Any _any_narrow(T_GIDL a)   { ... }  /* ... */
}
class Foo_GIDL : Base_GIDL<Foo_GIDL, Foo_CORBA> ...
```

Figure 6. GADT pseudocode for the casting functionality of the `Foo_GIDL` GIDL wrapper.

a reference to the corresponding CORBA object, the generic type information associated with them and the two-way casting functionality they define (CORBA-GIDL types). It follows that a GIDL wrapper is composed of two main components: the functionality described in the GIDL interface, and the *casting* functionality needed by the system for the two way communication with the underlying framework (CORBA).

In this way, we deal with two parallel type hierarchies: the original one (CORBA) and the one of the extension (GIDL). Figure 5 shows that each type of the extension encapsulates the functionality to transform back and forth between values of its type and values of its corresponding CORBA type, and also between values of its type and values of the CORBA type `Any`. Values of type `Any` can store any other CORBA type values, so GIDL uses type `Any` as the erasure of the non-qualified type-parameter.

This functionality can be expressed in an elegant way via GADTs, by writing a parameterized base class that contains the implementation for the casting functionality together with a precise interface, and by instantiating this base class with corresponding pairs of GIDL-CORBA types. Figure 6 demonstrates this approach. We see *three main advantages* for integrating the GIDL casting functionality via GADTs:

- This functionality is written now as a system component and not mangled inside the GIDL wrapper. It can be integrated either by inheritance (see the C++ mapping), or by aggregation (see the Java mapping).
- In addition it constitutes a clear meta-interface that characterizes all the pairs of types from the two parallel hierarchies, and makes it easier to reason about the type-safety of the GIDL extension.
- Finally, this approach is valuable from a code maintenance / post facto extension point of view. The casting functionality code is dependent on the underlying framework (CORBA, JNI, DCOM). Implementing it as a meta-program (see the C++ mappings), besides the obvious software maintenance advantages of being *static* and written only once (thus short), allows the GIDL compiler to generate *generic* code that is independent on the underlying architecture. Porting the framework on top of a new architecture will require rewriting this static code, reducing the modifications to be done at the compiler's code generator level.

```
1.  Vector< Long, RAI<Long>, RAI<Long> > vect  = ...;
2.  RAI<Long> it_beg=vect.begin(), it_end=vect.end(), it=it_beg;
3.  while(it!=it_end)
4.      *it++ = (vect.size() - i);
5.  sort(it_beg, it_end);     cout<<*it_beg<<endl;
```

Figure 7. C++ client code using a GIDL translation of STL. `RAI` and `Vector` are the GIDL types that model the STL random access iterator and vector types; `sort` is the native STL function.

The problem with this approach is that if the `Foo_GIDL` interface is a subtype of say `Foo0_GIDL` then it inherits the casting functionality of `Foo0_GIDL` – an undesired side-effect. The C++ binding addresses this problem by making the GIDL wrapper inherit from two components: one which respects the original inheritance hierarchy and which contains the functionality described in the GIDL specification, and one implementing the *system* functionality (`Base_GIDL<Foo_GIDL, Foo_CORBA>`).
This method breaks the subtyping hierarchy between the GIDL wrappers, and instead mimics subtyping by means of automatic conversion. This solution will be discussed in detail in Section 4. Since Java does not support automatic conversions, the Java mapping defines the casting component as an inner class of the GIDL wrapper, and uses a mechanism that resembles virtual types in order to retrieve and invoke the proper caster. The GIDL Java bindings are not however the subject of this paper.

3.2 Preserving the STL Semantics and Code Idioms

Figure 7 gives an example of GIDL client code that retrieves a vector's iterator (`it_beg`), updates it, sorts it and displays its first element. To allow such code, the translation needs to conform with both the native library semantics and its coding idioms.

First, to preserve the STL semantics, certain type properties must be enforced statically. For example, the parameters of the `sort` function need to belong to an iterator type that allows random access to its elements. As discussed in Section 5.1 these properties are expressed at the GIDL interface level by means of parametric polymorphism and operator overloading.

Second, for the (distributed) program to yield the expected result, `it` and `it_beg` have to reference different implementation-object instances sharing the same internal representation. Otherwise, after the execution of the while-loop (lines $3-4$), `it_beg` either points to its end, or it is left unchanged. Moreover, the instruction `*it++ = i` is supposed to update the value of the iterator's current element. Neither one of these requirements is achieved with the GIDL semantics. As detailed in Section 5.3, we can obtain the expected behavior with an extension mechanism applied to the GIDL stubs that overrides the default behavior in favor of one that satisfies the STL coding style.

4. Building a Natural C++ Interface from GIDL

This section presents the rationale behind the GIDL C++ bindings. We start by presenting the GADT 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.

4.1 The Generic Base Class

Figure 8 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

```
1 class ErasedBase { protected: void* obj; };
2 template<class T,class A,class A_v> class BaseObject :
3   public ErasedBase, public GIDL_Type<T> {
4  protected:
5   static void fillObjFromAny(CORBA::Any& a, A*& v) {
6     CORBA::Object_ptr co = new CORBA::Object();
7     a>>=co; A* w = A::_narrow(co); v = w;
8   }
9   static void fillAnyFromObj(CORBA::Any& a, A* v) { a<<=v; }
10 public:
11  typedef A GIDL_A;   typedef A_v GIDL_A_v;   typedef Self T;
12
13  BaseObject(A* ob)          { this->obj = ob; }
14  BaseObject(const A_v& a_v) {this->obj=a_v._retn();}
15  BaseObject(const T& ob)    { this->obj = ob.obj; } //
16  BaseObject(const GIDL::Any_GIDL& ob)
17  {T::fillObjFromAny(*ob.getOrigObj(),getOrigObj());}
18  template<class GG> BaseObject(
19    const BaseObject<GG,GG::GIDL_A,GG::GIDL_A_v>& o
20  ) { this->obj = (A*)o.getOrigObj(); }
21  /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/
22
23  operator A*() const { return (A*)obj; }
24  template < class GG > operator GG() const{
25    GG g; // test GG superclass of the current class!
26    if(0) { A* ob; ob = g.getOrigObj(); }
27    void*& ref = (void*&)g.getOrigObj();
28    ref = GG::_narrow(this->getOrigObj()); return g;
29  }
30  A*& getOrigObj()  const { return (A*) obj; }
31  void setOrigObj(A* o)   { obj = o; }
32
33  static A*& _narrow(const T& ob){return ob.getOrigObj();}
34  static CORBA::Any* _any_narrow(const T& ob) { /* ... */ }
35  static T _lift(CORBA::Any& a, T& ob)
36  { T::fillObjFromAny(a,ob.getOrigObj()); return ob; }
37  static T _lift(CORBA::Object* o) { return T(A::_narrow(o));}
38  static T _lift(const A* ob)      { return T(ob); }
39  /*** SIMILAR: _lift(A_v) AND _lift(CORBA::Any& v) ***/
40 };
```

Figure 8. The base class for the GIDL wrapper objects whose types are GIDL interfaces. (We have omitted the `inline` keyword)

`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 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.

4.2 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:

```
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()   { /* ... */ }
};
```

Figure 9. 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 3. We first examine the shortcomings of a naïve translation that would preserve the inheritance hierarchy among the generated GIDL wrappers. Figure 9 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 pointer in a base class and to use virtual inheritance (see the `BaseObject` class in Figure 8). 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 10 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 [9] traits [10]. `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 GADTs 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

```
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 ***/
};
```

Figure 10. Part of the C++ generated wrapper for the GIDL::Leaf interface. ::Leaf and ::Leaf_var are CORBA-types

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.

4.3 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 11A 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 11C 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 8) that fills in the object encapsulated by the GIDL Any wrapper with the appropriate value stored in the T-type object.

Figure 11D 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 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

```
// 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;
```

Figure 11. 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.

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.

4.4 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 8. 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 un-

```
// 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);
}
```

Figure 12. GIDL interface and the corresponding stub/skeleton wrappers for function `foo`

necessary 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 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 8) that the following relations hold:

`G::_lift[A*]`$\circ$`G::_narrow[G] (a)` $\sim$ `a`
`G::_lift[Object*]`$\circ$`G::_narrow[G] (a)` $\sim$ `a`
`G::_lift[Any]`$\circ$`G::_any_narrow[G] (a)` $\sim$ `a`

where `[]` is used for the method's signature, $\circ$ stands for function composition, while `g1`$\sim$`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 12 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.

5. Library Translation: Trappers

The immediate use of GIDL is to enable applications that combine parameterized, multi-language components. This section investigates another important application: what is required to use GIDL as a vehicle to access generic libraries beyond their original language boundaries, and what techniques can automate this process? For the purpose of this paper, we restrict the discussion to the simpler case when the implementation shares a single process space.

We find C++'s *Standard Template Library* (STL) to be an ideal candidate for experimentation due to the wealth of generic types, the variety of operators, and high-level properties such as the orthogonality between *the algorithm and container domains* it exposes. Furthermore, the fact that, for performance reasons, STL does not hide the representation of its objects poses new translation-related challenges. In what follows, we review the STL library at a high level, show the GIDL specification for a server encapsulating part of STL's functionality, identify and propose solutions to two issues that prevent the translation from implementing the library semantics, and discuss the performance-related trade-offs.

5.1 STL at a High Level

STL [2] is a general purpose generic library known for providing a high level of modularity, usability, and extensibility to its components, without impacting the code's efficiency. The STL components are designed to be *orthogonal*, in contrast to the traditional approach where, for example, *algorithms* are implemented as methods inside *container* classes. This keeps the source code and documentation small, and addresses the extensibility issue as it allows the user algorithms to work with the STL containers and *vice-versa*. The orthogonality of the algorithm and container domains is achieved, in part, through the use of iterators: the algorithms are specified in terms of iterators that are exported by the containers and are data structure independent. STL specifies for each container/algorithm the iterator category that it provides/requires, and also the valid operations exported by each iterator category. These are however defined as English annotations in the standard, as C++ lacks the formalism to express them at the interface level.

Figures 13 and 14 present excerpts of the GIDL iterators and vector interfaces respectively. We simulate *selftypes* [11] by the use of an additional generic type, `It`, bounded via a mutual recursive export based qualification (`:-`). This abstracts the iterators functionality: `InpIt<T>` exports `==(InpIt<T>)` method, while `RaiIt<T>` exports the `==(RaiIt<T>)` method. An *input iterator* has to support operations such as: incrementation (`it++`), dereferencing (`*it`), and testing for equality/non-equality between two *input iterators* (`it1==it2`, `it1!=it2`). A *forward iterator* allows reading, writing, and traversal in one direction. A *bidirectional iterator* allows all the operations defined for the *forward iterator*, and in addition it allows traversal in both directions. *Random access iterators* are supposed to support all the operations specified for *bidirectional iterator*, plus operations as: addition and subtraction of an integer (`it+n, it-n`), constant time access to a location `n` elements away (`it[n]`), bidirectional big jumps (`it+=n; it-=n;`), and comparisons (`it1>it2;` etc). The design of iterators and containers is non-intrusive as it does not assume an inheritance hierarchy; we use inheritance between iterators only to keep the code short. The `STLvector` container does not expect the iterators to be subject to an inheritance hierarchy, but only to implement the functionality described in the STL specification: `RI` is expected to share

```
  interface BaseIter<T, It:-BaseIter<T; It> > {
    unsigned long getErasedSTL();   It  cloneIt();
    void operator"++@p"();  void operator"++@a"();
  };
  interface InputIter<T,It:-InputIter<T;It> >:BaseIter<T,It>{
    T       operator"*"  ();
    boolean operator"==" (in It it);
    boolean operator"!=" (in It it);
  };
  interface ForwardIter<T, It:-ForwardIter<T; It> >
      : OutputIter<T, It>, InputIter<T; It>
      { void assign(in T t1);                              };
  interface BidirIter<T, It:-BidirIter<T; It> >
      : ForwardIter<T, It>
      { void operator"--@p"();    void operator"--@a"(); };
  interface RandAccessIter<T,It:-RandAccessIter<T,It> >
      : BidirIter<T, It> {
    boolean operator">"  (in It it);
    /*   same for "<", ">=", "<="   */
    Iterator operator"+" (in long n);
    Iterator operator"-" (in long n);
    void operator"+="    (in long n);
    void operator"-="    (in long n);
    T    operator"[]"(in long n);
    void assign(in T obj, in long index);
  };

  interface InpIt<T>   : InputIter<T, InpIt<T> >   {};
  interface ForwIt<T>  : ForwardIter<T, ForwIt<T> >{};
  interface BidirIt<T> : BidirIter<T, BidirIt<T> > {};
  interface RAI<T>     : RandAccessIter<T, RAI<T> >{};
```

Figure 13. GIDL specification for STL iterators; @p/@a disambiguate between prefix/postfix operators

```
interface STLvector
<T, RI:-RandAccessIter<T,RI>; II:-InputIter<T,II> > {
  unsigned long getErasedSTL();
  RI begin ();   RI end();   T operator"[]"(in long n);
  void    insert(in RI pos, in long n, in T x);
  void    insert(in RI pos, in II first, in II last);
  RI      erase (in RI first, in RI last);
  void    assignAtIndex(in T obj, in long index);
  T       getAtIndex   (in long index);
  void    assign       (in II first, in II end);
  void    swap         (in STLvector<T, Ite, II> v); //....
};
```

Figure 14. GIDL specification for STL vector

structural similarity [1] with its qualifier `RandAccessIter`. Note that, unlike its underlying architecture, GIDL supports operator and method overloading.

As observed in [8], the GIDL interface is expressive, self-describing, and enforces the STL specification requirements at a high-level. Another interesting aspect is that GIDL stub wrappers for iterators are themselves valid STL iterators: They encapsulate the functionality specified by STL. They can also encapsulate the necessary type aliasing definitions, either by specifying them directly in the GIDL specification, or by making the GIDL stub wrapper extend the STL base class of their corresponding iterator category. For example `InputIter` stub extends the STL class `input_iterator<T,int>`. The latter is achieved by enriching the GIDL specification with meta data.

5.2 Implementation Approaches

GIDL is designed to be a *generic* extension framework that can plug in various back-ends as underlying architectures. An orthogonal, but nevertheless important, direction is to employ GIDL as middleware for exporting generic libraries' functionality to different environments than those for which they were originally designed. Our approach is to use a *black-box* translation scheme that wraps the

```
template <class T,class It,class It_impl,class II>
class STLvector_Impl :
  virtual public ::POA_GIDL::STLvector<T, It, II>,
  virtual public ::PortableServer::RefCountServantBase
{
  private:    vector<T>* vect;
  public:
    STLvector_Impl() { vect = new vector<T>(10); }
    virtual GIDL::UnsignedLong_GIDL  getErasedSTL()
        { return (CORBA::ULong)(void*)vect; }
    virtual void assign(T& val, GIDL::Long_GIDL& ind)
        { (*vect)[ind] = val; }
    virtual T getAtIndex(GIDL::Long_GIDL& ind)
        { return (*vect)[ind]; }
    virtual T operator[](GIDL::Long_GIDL& a1_GIDL)
        { return (*vect)[a1_GIDL]; }
    virtual It erase( It& it1_GIDL, It& it2_GIDL ) {
      T* it1 = (T*)it1_GIDL.getErasedSTL();
      T* it2 = (T*)it2_GIDL.getErasedSTL();
      vector<T>::iterator it_r = vect->erase(it1, it2);
      It_impl* it_impl = new It_impl(it_r, vect->size());
      return (*it_impl->_thisGIDL());
    }  // ...
};

template<class T,class It,class It_impl>
class InputIter_Impl :
  virtual public POA_GIDL::InputIter<T, It>,
  virtual public BaseIter_Impl<T, It, It_impl>,
  virtual public ::PortableServer::RefCountServantBase
{
  // private: T* iter; field inherited from BaseIter_Impl
  public:
    virtual It cloneItGIDL()
        { return (new It_impl(iter))->_thisGIDL(); }
    virtual GIDL::UnsignedLong_GIDL getErasedSTL()
        { return (CORBA::ULong)(void*)iter; }
    virtual T operator*()   { return *iter; }
    virtual GIDL::Boolean_GIDL operator==(It& it1_GIDL) {
      CORBA::ULong d1 = this->iter;
      CORBA::ULong d2 = it1_GIDL.getErasedSTL();
      return (d1==d2);
    };
};
```

Figure 15. GIDL vector and input iterator server implementations.

library objects into GIDL objects and to study what other constructs are required to enforce the library semantics.

Figure 15 exemplifies our approach. Each implementation of a GIDL type holds a reference to the corresponding STL object that can be accessed via the `getErasedSTL` function in the form of an `unsigned long` value. The implementation of the `erase` function retrieves the STL objects corresponding to the GIDL wrapper parameters, calls the STL `erase` function on the STL vector reference, and creates a new GIDL server corresponding to the iterator result. Note that the semantics of the `erase` function are irrelevant in what the translation mechanism is concerned.

The GIDL code in Figure 16 provides, in our opinion, the look and feel of regular STL code. The only thing that differs are the types for the vector and iterators (lines 1-4). A vector is obtained in line 6. The `rai_beg` and `rai_end` iterators point to the start and the end of the vector element sequence. Then the loop in lines 12-15 assigns new values to the vector's elements.

There are, however, *two problems* with the current implementation. The first appears in line 14 where *dereferencing is followed by an assignment* as in `*rai=val`. In C++ this assigns the value `val` to the iterator's current element. The GIDL code does not accomplish this: the result of the `*` operator is a `Long_GIDL` object whose value is set to `val`. The iterator's current element is not updated as no request is made to the server. The origin of this problem is that GIDL does not support reference-type results, since the implementation and client code are not assumed to share the same process space.

```
1.  typedef GIDL::Long_GIDL   Long;
2.  typedef GIDL::RAI<Long>   rai_Long;
3.  typedef GIDL::InpIt<Long> inp_Long
4.  typedef GIDL::STLvector<Long,rai_Long,rai_Long>
5.          Vect_Long;
6.  Vect_Long vect   = ...;
7.  rai_Long iter    = vect.begin();
8.  rai_Long rai_end = vect.end();
9.  rai_Long rai_beg = iter;          // problem 2
10.
11. int count = 0;
12. while( rai_beg!=rai_end ) {
13.     if(*rai_beg!=33)
14.         *rai_beg++ = count++;     // problem 1
15. }
16. cout<<*iter<<endl;
```

Figure 16. GIDL client code that uses the STL library.

The second problem surfaces in line 16, where the user intends to print the first element of the vector. The copy constructor of the GIDL wrapper *does not create* a new implementation object, but instead *aliases* it: After line 9 is executed, both `rai_beg` and `iter` share the same implementation. Consequently, at line 16 all three iterators point to the end of the vector. The easy fix is to replace line 9 with `rai_Long rai_beg = iter.clone()` or with `rai_Long rai_beg = iter+0`. We are aiming, however, for a higher degree of composition between GIDL and STL components, where for example GIDL iterators can be used as parameters to STL algorithms. Since the STL library code is out of our reach, the direct fix is not an option.

One way to address the first problem is to introduce a new GIDL parameterized type, say `WrapType<T>`, whose object-implementation stores a `T` value while its GIDL interface provides accessors for it: `interface WrapType<T> { T get(); void set(in T t) }` . `WrapType` is a special GIDL type: its constructors and assignment operators call the `set` function, while its cast operator calls the `get` function to return the encapsulated `T`-type object. Instantiating the iterator and vector over `WrapType<T>` instead of `T` fixes the first issue. The main drawback of this approach is that it adds an extra indirection. In order to get the `T` type object two server calls are performed instead of one. Furthermore, it is not user-transparent, as the iterators and vectors need to be instantiated over the `WrapType` type. The next section discusses the techniques we employed to deal with these issues.

5.3 Trappers and Wrappers

We preserve the STL's programming idioms under GIDL by extending the GIDL wrapper with yet another component that enforces the library semantics. Figure 17 illustrates our approach. `RaiIt_Lib` refines the behavior of its corresponding GIDL wrapper `RAI` to match the library semantics.

First, it provides two sets of constructors and assignment operators. The one that receives as parameter a library wrapper object *clones* the iterator implementation object, while the other one aliases it. The change in Figure 16 is to make `rai_Long` and `Vect_Long` alias `RaiIt_Lib<Long>` and `STLvect_Lib<Long,rai_Long,rai_Long>` types, respectively. Now `iter`/`rai_end` alias the implementation of the iterators returned by the `begin`/`end` vector operations, while `rai_beg` clones it (see lines 7, 8, 9). At line 16 `iter` points to the first element of the vector, as expected.

Second, the `RaiIt_Lib` class defines a new semantics for the `*` operator that now returns a `Trapper` object. At a high-level, the *trapper* can be seen as a proxy for performing read/write operations. It captures the container and the index and uses container-methods to perform the operation. The "trapper" in Figure 17 extends its type parameter, and thus inherits all the type parameter operations. In addition it refines the assignment operator of `T` to call an iterator method to update its elements. This technique solves the problem encountered at line 14 in Figure 16 and it can be applied in a more general context to extend GIDL with *reference-type results*. Note that the use of the *trapper* is transparent for the user. The type `TrapperIterStar` does not appear anywhere in the client code. Furthermore, objects belonging to this type can be stored and manipulated as `T&` objects. For example, `T& t = *it; if(t<0) t=-t;` will successfully update the iterator's current element. This requires however that the GIDL wrappers declare the `=(T&)` operator *virtual*.

```
template<class T,class Iter> class TrapperIterStar : public T {
  protected:
    Iter it;
  public:
    TrapperIterStar(const Iter& i)
      { it = i; obj = (*it).getOrigObj(); }
    TrapperIterStar(const TrapperIterStar<T,Iter>& tr)
      { it = tr.it; obj = (*it).getOrigObj(); }

    void operator=(const T& t)
      { it.assign(t); obj = t.getOrigObj(); }
    void operator=(const TrapperIterStar<T,Iter>& tr)
      { it.assign(tr.getOrigObj()); obj = tr.getOrigObj(); }
};

template<class T> class RaiIt_Lib : public GIDL::RAI<T::Self> {
  private:
    typedef GIDL::RAI<T>                          It;
    typedef TrapperIterStar<T,It>                 Trapper;
    typedef GIDL::BaseObject<It,::RAI,::RAI_var>  GIDL_BT;
  public:
    typedef T                                     Elem_Type;
    typedef Self                                  It;

    RaiIt_Lib()           : GIDL_BT()               {}
    RaiIt_Lib(const It& r): GIDL_BT(r.getOrigObj()) {}
    RaiIt_Lib(const RaiIt_Lib<T>& r)
                : GIDL_BT(r.cloneIt().getOrigObj()) {}

    operator It()                    { return *this; }
    Trapper  operator*()  { return Trapper( *this ); }

    void operator=(const It& iter)
        { setOrigObj(iter.getOrigObj()); }
    void operator=(const InpIt_Lib<T>& iter)
        { setOrigObj(iter.cloneIt().getOrigObj()); }
};

template<class T,class RI,class II> class Vect_Lib
: public GIDL::STLvector<T::Self,RI::Self,II::Self>{...}
```

Figure 17. Library Iterator Wrapper and its associated Trapper that targets ease of use.

We conclude this section with several remarks. It is easy to anticipate how GIDL metadata can drive the compiler to generate the library wrapper code that captures the library semantics. All that is needed is the name of a method-member: `cloneIt` for the iterator's copy constructor and `assign` for the type-reference result. When available, the library wrappers should replace the GIDL corresponding types. For example, when using an STL algorithm with GIDL iterators, the former should be parameterized by the library wrapper types. Finally, note that nesting library wrappers is safe: We have that `RaiIt_Lib<RaiIt_Lib<Long> > it; **it=5;` works correctly. Also, the use of the `Self` abstract type member in the extension clause of the iterator/vector library wrappers ensures that the their inherited operations return GIDL wrapper objects. Therefore no unnecessary cloning operation are performed:

```
Vect_Lib<Long,RaiIt_Lib<Long>,RaiIt_Lib<Long> > v;
RaiIt_Lib<Long> it = vect.begin();
```

```
template<class T,class Iter> class TrapperIterStar {
  protected:    Iter it;
  public:
    TrapperIterStar(const Iter& i)   { it = i; }
    TrapperIterStar(const TrapperIterStar<T,Iter>& tr)
        { it = tr.it; }
    operator T()                     { return *it; }

    TrapperIterStar<T::Elem_Type, T> operator*() const
        { return *(*it); }
    void operator=(const TrapperIterStar<T,Iter>& trap)
        { it.assign(trap.it.operator*()); }
    void operator=(const T& t)       { it.assign(t); }
};
```

Figure 18. Trapper model that targets performance

Trapper Type	200000	20000	2000	200
EOU trapper	13.4	11.7	5	3.4
Perf. trapper I	1	1.4	1.5	1.68
Perf. trapper II	1	1.05	1.16	1.17

Table 2. The table shows the time ratio between trapper-based and optimal STL code that tests the read/write operation on the iterator's elements. The size of the iterator is varied from 200 to 200000.
EOU trapper = the one in Figure 17 (ease of use).
Perf Trapper I = the one in Figure 18 (performance).
Perf Trapper II = improved version of the latter, which by-passes the extra indirection introduced by the GIDL wrappers.

5.4 Ease of use - Performance Trade-off

The *trapper's* design is a trade-off between performance and ease of use. The implementation above targets ease of use, since a trapper object can be disguised and manipulated under the form of a `T&` object. An alternative, targeting performance, can model the trapper as a read/write lazy evaluator as shown in Figure 18. Note that the mix-in relation is cut off, and instead the support for nested iterators is achieved by exporting the `*` operator. It follows that the trapper cannot be captured as a `T&` object and used at a later time. The intent is that a trapper is subject to exactly one read or write operation (but not both), as in: `T t = *it++; *it = t; t.method1();`. The trapper's purpose is to postpone the action until the code reveals the type of the operation to be performed (read or write). Consequently, the constructors and the `=` operators are lighter, while a write operation accesses the server only once (instead of twice). Furthermore, this approach does not require the `=` operator to be declared *virtual* in the GIDL wrapper.

Table 2 shows the trapper-related performance results. Notice that the code using the trapper targeting ease of use is from 3.4 to 13.4 times slower than the optimal STL code, while the one targeting performance incurs an overhead of at most 68%. As the iterator size increases, the cache lines are broken and the overhead approaches 0. The test programs were compiled with the *gcc* compiler version 3.4.2 under the maximum optimization level (`-O3`), on a 2.4 GHz Pentium 4 machine.

We found the *trapper* concept quite useful and we employed it to implement the GIDL arrays. The previous design was awkward in the sense that, for example, the `Long_GIDL` class was storing two fields: an `int` and a pointer to an `int`. The latter pointed to the address of the former when the object was not an array element and to the location in the array otherwise. All the operations were effected on the pointer field. By contrast, the *trapper* technique allows a natural representation consisting of only one `int` field.

6. Conclusions

We have examined a number of issues in the extension of generic libraries in heterogeneous environments. We have found certain programming language concepts and techniques to be particularly useful in extending libraries in this context: GADT, *abstract type members* and *traits*. Generic libraries that are exported through a language-neutral interface may no longer support all of their usual programming patterns. We have shown how particular language bindings can be extended to allow efficient, natural use of complex generic libraries. We have chosen the STL library as an example because it is atypically complex, with several orthogonal aspects that a successful component architecture must deal with. The techniques we have used are not specific to the STL library, and therefore may be adapted to other generic libraries. This is a first step in automating the export of generic libraries to a multi-language setting.

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] A. S. David R. Musser, Gillmer J. Derge. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley (ISBN 0-201-37923-6), 2001.

[3] R. E. Johnson. Type Object. In *EuroPLoP*, 1996.

[4] A. Kennedy and C. V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–40, 2005.

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

[6] Microsoft. DCOM Technical Overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp, 1996.

[7] Sun Microsystems. JavaBeans http://java.sun.com/products/javabeans/reference/api/, 2006.

[8] 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.

[9] M. Odersky and al. Technical Report IC 2004/64, an Overview of the Scala Programming Language. Technical report, EPFL Lausanne, Switzerland, 2004.

[10] M. Odersky, V. Cremet, C. Rockl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of ECOOP'03*.

[11] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57, 2005.

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

[13] OMG. Common Object Request Broker: Architecture and Specification. Revision2.4 (October 2000), OMG Specification, 2000.

[14] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley and Sons, 2000. "Wiley computer publishing.".

[15] Sun. Java Native Interface Homepage, http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

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

[17] H. Xi, C. Chen, and G. Chen. Guarded Recursive Data Type Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 224–235, 2003.