An Architecture for Generic Extensions

Cosmin E. Oancea [1]

The Department of Computer Science and Engineering, Texas A&M University, College Station, TX, 77843-3112, USA

Stephen M. Watt [1]

The Department of Computer Science, The University of Western Ontario, London, ON, N6A 5B7, Canada

Abstract

We examine what is necessary to allow generic libraries to be used naturally in a multi-language, potentially distributed environment. Language-neutral library interfaces usually do not support the full range of programming idioms that are available when a library is used natively. We investigate how to structure the language bindings of the neutral interface to achieve a better expressibility and code reuse. We furthermore 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. Our approach is to treat a library as a software component and to view the problem as one of component extension. 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.

Key words: Generics, C++ Templates, Software Component Architecture, Middleware, Curiously Reoccurring Template Pattern (CRTP), Type Members.

Email addresses: coancea@cs.tamu.edu (Cosmin E. Oancea), watt@csd.uwo.ca (Stephen M. Watt).

[1] Supported by the Natural Sciences and Engineering Research Council of Canada.

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 along 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 unsafe to compose libraries based on low level language-interoperability solutions, such as JNI or "`extern C`" with remote procedure calls. Therefore, components are usually constructed and accessed through some framework such as CORBA [17], DCOM [9] or the .NET framework [8]. 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 commonplace 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 previously described the Generic Interface Definition Language framework, GIDL [11], 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 [22]. 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 [20].

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 the 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 [11] 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 [14]. 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 services. They identify three language abstractions, namely "*selftype annotations*, *abstract type members*, and *traits*" that enable the design of first-class value components (components that use neither static data nor hard references to the required modules).

We look at the GIDL extension as a component that can be employed on top of other underlying architectures and which can be, in 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 reasoning about the type safety of the extension. 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 into first-class value components. In the GIDL case for example, one component should encapsulate the underlying architecture (UA) specifics and be statically generated. The other one should generically implement the extension mechanism. This would allow GIDL to be plugged in with various UAs 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 further diminish it.

We have found that the *curiously recurring template pattern* [2] (CRTP), *member types*[2] [3,5], and C++ simulated Scala-*traits* enable a better code structure in the sense described above. This is in agreement with observations of Odersky and Zenger [14]. We also note that at a very high-level this combination of

[2] Unfortunately, concepts, associated types and constraint propagation for generics are not yet part of the C++ language.

techniques resembles an object-oriented encoding of the generalized algebraic data types [23].

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)[10,19] functionality to a multi-language, distributed use. In this, we had two objectives.

The first objective was to determine to what degree the interface translation could preserve the "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. For example some 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 from conforming 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. We show empirical arguments to support our intuition that the GIDL extension design introduces very little overhead, if any.

The rest of the paper is organized as follows. Section 2 surveys the main programming techniques used in our implementation, and gives a high-level review of the GIDL framework. Section 3 presents at a high-level the rationale of our design and the technique we used to implement our extension framework, 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.

2 Background

We begin with a brief survey of the programming techniques we have used in our cross-platform mappings, and illustrate their use through specific examples. We then give an overview of the GIDL framework and the semantics of its model for parametric polymorphism. A more detailed account of may be found elsewhere [11].

```
template<class T> struct Base {
  T fun() {
    T* me = static_cast<T*>(this);
    return me->fun_help();
  }
};

struct X : Base<X> {
 X fun_help() { /* ... */ }
}
```

```
template<class T>
struct Base {
  static int num_obj = 0;

  void Base()
    { num_obj++; }
};

struct X : Base<X> { ... };
struct Y : Base<Y> { ... };
```

Fig. 1. (a) Simulating Virtual Functions, (b) Concise Extension via CRTP.

2.1 The Curiously Recurring Template Pattern

The *curiously recurring template pattern* (CRTP) of Coplien [2] is a C++ idiom in which a class `X` inherits from a parameterized base class `A` that has `X` as the instantiation of one of its type parameters (e.g. `class X : A<X>`).

Figure 1 shows two potential uses of CRTP. Example 1(a) simulates the effect of calling virtual functions, but without the potentially high cost associated to dynamic polymorphism. Example 1(b) exhibits a post-facto extension technique through which each class derived from `Base` records the number of objects belonging to that class. Note that a change in the implementation of `Base` would uniformly change the behavior of both classes `X` and `Y`, and also that `X` and `Y` do not share a common superclass.

The C++ bindings of GIDL use the CRTP idiom more in the sense of (b): `Base` would correspond to static code that depends on the specifics of the underlying architecture (UA) and implements the meta-interface of the extension, while `X` and `Y` would correspond to the code generated by the GIDL compiler, which does not depend on the UA.

2.2 Associated Types and Constraint Propagation

Czarnecki and Eisenecker use "publishing types" [3] to implement a system that allows automatic selection and composition of software components much in the same way as interchangeable parts and automated assembly lines are used in the car industry.

Figure 2 illustrates Czarnecki and Eisenecker's approach: for a given layered architecture, each component from a certain layer (e.g. `ManulaTransmission`) has a template parameter to be instantiated with a component from the layer below it (e.g. `GasolineEngine`). The bottom layer is the configuration repos-

```
template <class Engine> struct ManualTransmission {
  typedef typename Engine::Config Config;
  enum { speeds = Config::speeds };
  Engine e;
};
template <class Config_> struct GasolineEngine {
  typedef typename Config_ Config;
  GasolineEngine() { }
};
struct Config1 {
  enum { speeds = 5 };
  typedef GasolineEngine<Config1>    Engine;
  typedef ManualTransmission<Engine> Transmission;
  // ...
};
```

Fig. 2. Publishing Type Member `Config` is used to pass information along all layers.

itory, which is passed via the `Config` type member from one layer to the next and thus, it is used to communicate configuration information to all layers. Our C++ mapping uses type members to address type-safety concerns.

Järvi *et al.* show how to provide support for associated types and constraint propagation [5] in object-oriented languages that support bounded parametric polymorphism, such as C# and C++ with concepts [4]. Associated types are constrained or qualified type members of classes, and type-member constraints can be refined in derived classes. Constraint propagation allows certain constraints on type parameters to be inferred from other constraints on those parameters and their use in expressions in base classes. The main benefit brought by associated types and constraint propagation is that function/interface signatures can be significantly shorter in terms of both number of constraints and number of generic types than when using only qualified generic types.

Our GIDL implementation supports neither associated types nor constraint propagation. While this can be easily accomplished at GIDL level with the translation proposed by Järvi *et al*, the main impediment is that this would compromise the ease of use of the GIDL framework for the languages that do *not* support these features, such as Java. (The Java client/server bindings and the GIDL interface would differ significantly in number of generic types and constraints; the Java user might need in effect to learn the translation rules in order to use the Java bindings.)

2.3 Parameterized/Generalized Algebraic Data Types

Functional languages such as Haskell and ML support generic programming through user-defined 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` has all its recursive uses in its definition 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 types other than the formal type parameters. Kennedy and Russo [7] show, among other things, that mainstream 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. Our C++ mapping resembles at a high-level Kennedy and Russo's translation.

2.4 The GIDL Framework

The Generic Interface Definition Language framework [11] (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 first summarize the GIDL model for parametric polymorphism, and then briefly describe the GIDL architecture.

The GIDL language

GIDL extends the CORBA–IDL [15] language with support for *F-bounded parametric polymorphism*, where type parameters can be qualified based on name or structural subtyping. 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 (structurally) qualified to export the whole functional-

```
/********************** GIDL interface **********************/

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

Fig. 3. GIDL Specification and C++ Client Code for a Binary Tree.

ity of its qualifier `Comparable<K>`; that is, the comparison operations `>` and `==`. However, it is not necessary that an instantiation `X` of `K` is a subtype of `Comparable<X>`. GIDL also supports a stronger qualification than `:-` denoted by `:` that enforces a (name) subtyping relation between the instantiation of the type parameter and the qualifier.

Figure 3 also presents the C++-GIDL client code that builds a simple binary tree whose root contains the data/key `7` and its two leafs contain the data/keys `6` and `8`. The `tree.find(i8)` call searches the tree for the node or leaf with the key equal to `8` and returns the data associated with it, in our case `8`. Note that the code is very natural for the most part; 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 [6], where objects are used to carry type information.

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. It 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, and the overhead of allocating the associated GIDL wrappers. The former can be addressed by aggressive inlining. The latter may be expensive for Java, for example, where GIDL wrappers are allocated on the heap. However, since these wrappers mainly store generic type information, one can anticipate that the allocation overhead can be effectively reduced in many cases by a combination of

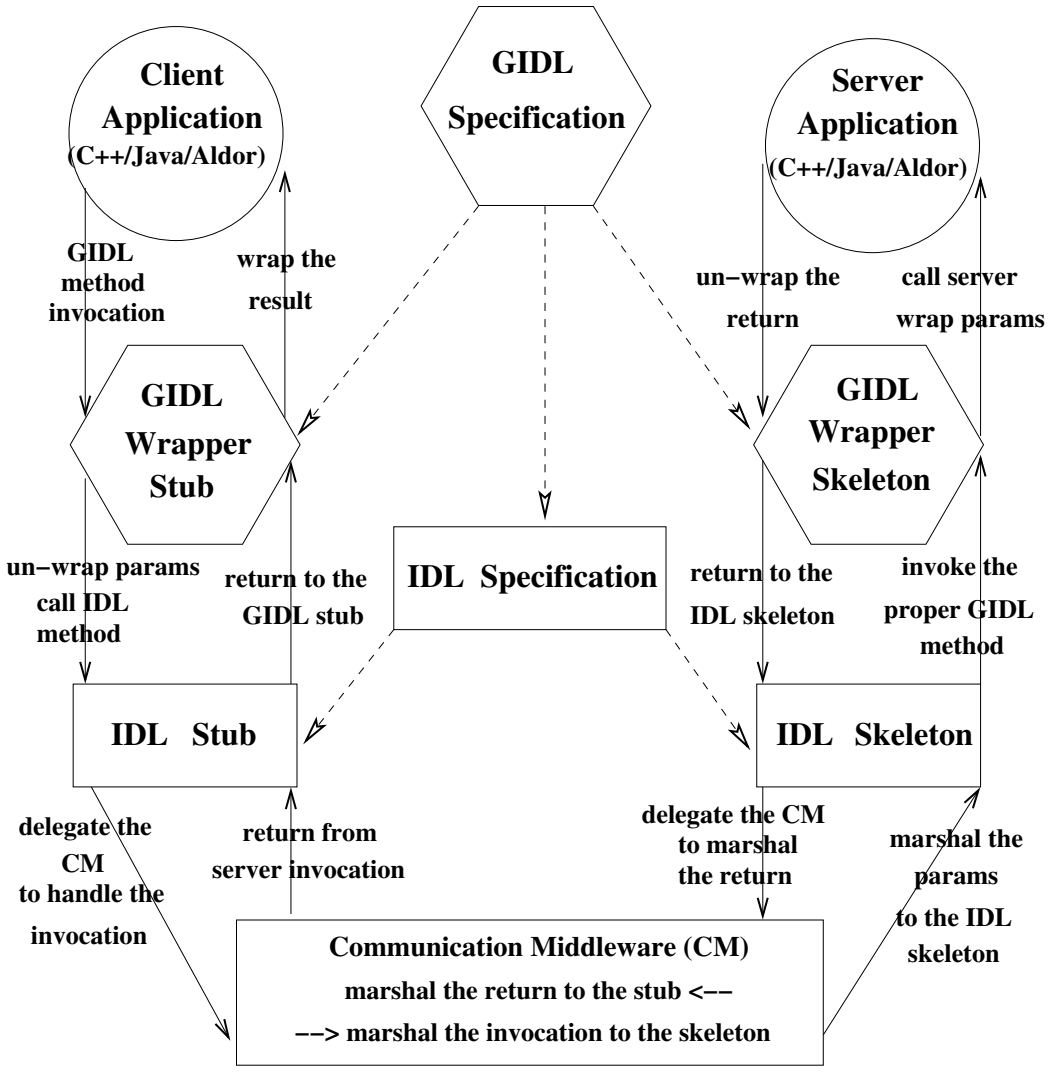

Fig. 4. GIDL Architecture.
Circle – user code. *Hexagon –* GIDL *component.*
Rectangle – underlying architecture component.
Dashed arrow – is compiled to.
Solid arrow – method invocation flow.

pointer aliasing, scalar replacement of aggregates, copy propagation and dead code elimination. This potential overhead is the price to pay for the generality of the approach: the generic extension will work on top of any UA vendor implementation while maintaining backward compatibility.

3 Problem Statement and High-Level Solution

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

```
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)  { ... }
}
```

Fig. 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. (The precise parameter/result types are given in Figure 10.)

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

Section 2.4 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 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 the CRTP idiom, 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

```
template<class T_GIDL, class T_CORBA>
class Base_GIDL {
    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> ...
```

Fig. 6. CRTP-Based Pseudocode of GIDL's Meta-Interface of Casting Functionality.

demonstrates this approach. The key difference is that either a change of the UA or future extensions will require (i) modifications to the GIDL translator under the (naive) approach depicted in Figure 5, but (ii) mainly re-adjustments of the `Base_GIDL` class under the approach depicted in Figure 6. We see *three main advantages* for integrating the GIDL casting functionality in this way:

- This functionality is written now as a system component and not mangled inside the GIDL wrapper. It can be integrated either by inheritance (as in the C++ mapping), or by aggregation (which is used in 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 of 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. We found the CRTP idiom, together with (published) type members and Scala-like traits instrumental in achieving these desiderata.

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 *casting* functionality
(i.e. `Base_GIDL<Foo_GIDL, Foo_CORBA>`).

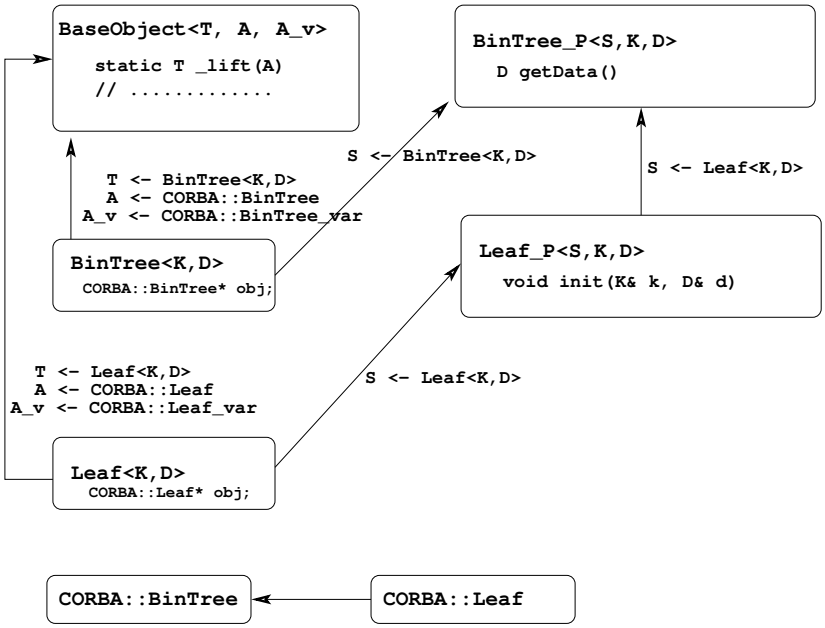

Fig. 7. Inheritance Graph for the Binary Tree Example in Figure 3.
Arrows go from subclass to superclass. CORBA types are prefixed by `CORBA::`.
The rest of the types belong to the GIDL wrappers.

```
1.  Vector< Long, RAI<Long>, RAI<Long> > vect  = ...; int i = 0;
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;
```

Fig. 8. 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.

This method breaks the subtyping hierarchy between the GIDL wrappers, and instead mimics subtyping by means of automatic conversion. Figure 7 presents the resulting inheritance graph structure for the binary tree GIDL specification in Figure 3. This solution will be discussed in detail in Section 4.

3.2 Preserving the STL Semantics and Code Idioms

Figure 8 gives an example of GIDL client code that retrieves a vector's iterator (`it_beg`), updates the vector, 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/GIDL) program to yield the expected result, `it` and `it_beg` have to reference different GIDL implementation instances that initially reference the same STL iterator as their internal representation. As the while-loop is executed, only the STL iterator corresponding to the `it` implementation object should be incremented (`++`). Otherwise, after the while-loop execution (lines $3-4$), `it_beg` will also point to its end. Unfortunately, the assignment operator of the GIDL wrapper does not clone a server instantiation but merely sets both it_beg and `it` to refer the same server object.

Finally, the instruction `*it++ = vect.size() - i++` is supposed to update the value of the iterator's current element. However the result of `*it` is a `Long_GIDL` object (i.e. basic-type value) that does not have a server implementation, and hence the iterator's elements on the server remain un-modified. It follows that these requirements are not achieved with the GIDL semantics of the C++ mapping. As detailed in Section 5.3, we can obtain the expected behavior with an extension mechanism applied to the GIDL wrappers 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 how the user interacts with the GIDL bindings, then we present how the casting functionality of the GIDL wrapper classes is implemented. We follow by showing how the GIDL inheritance hierarchies are implemented and comment on the language features that we found most useful in this context. We conclude this section with an informal discussion about the soundness of the translation mechanism.

4.1 Ease of Use

Ease of use has been one of the high-level goals of the GIDL framework design. The extra indirection required by the GIDL extension has made it possible for the GIDL wrappers to encapsulate a variety of constructors, cast and assignment operators that make the user interaction with the system more natural than with the original framework (CORBA).

Figures 9`A` and `B` illustrate the CORBA/GIDL code that inserts GIDL/CORBA `Octet` and `String` objects into `Any` objects, then performs the reverse opera-

```
// 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 = static_cast<T&>(b);
  if(!this->obj) this->obj = new CORBA::Any();
  T::_lift(this->obj, a);
}

// D. GIDL Arrays
interface Foo<T :- Number> {    //GIDL specification
    typedef T Array_T[100];
    T sum_array(inout Array_T arr); //sums up the elements of the array
};

// E. 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;
}
Long_GIDL sum   = foo.sum_array(arr);
cout<<"sum (4950): "<<sum<<endl;
```

Fig. 9. GIDL/CORBA Use of the `Any` Type.

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

Table 1
CORBA types for in, inout, out params and result. `ct` = `const`, `sl` = `slice`.

tion 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 9C 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 is (naturally) exported to the user [3], and that inherits from `GIDL_Type<T>` is `T`, therefore the static cast is safe. Finally the method calls the `T::_lift` operation (see Figure 10) that fills in the object encapsulated by the GIDL `Any` wrapper with the appropriate value stored in the `T`-type object.

Figure 9D 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 `Any_GIDL` does not provide an assignment operator or constructor that takes an `int` parameter. (However, if `Array_T` is defined as an array of longs the latter will work since the `Long_GIDL` type features the proper assignment operator).

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&`, and for the result is `T`, where `T` denotes an arbitrary GIDL type. The necessary type conversions are hidden in the GIDL wrapper.

[3] The GIDL stub roots `GIDL_Object` and `GIDL_Type` have protected constructors.

```
1 #define me() static_cast<T*>(this);
2 template<class T,class A,class A_v> class BaseObject :
3       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 T Self;
12 protected:  BaseObject() {}
13  void initBO(A* ob)          { me()->obj = ob; }
14  void initBO(const A_v& a_v) { me()->obj=a_v._retn();}
15  void initBO(const T& ob)    { me()->obj = ob.obj; } //
16  void initBO(const GIDL::Any_GIDL& ob)
17      {T::fillObjFromAny(*ob.getOrigObj(), getOrigObj());}
18  template<class GG> void initBO(
19   const BaseObject<GG,typename GG::GIDL_A,typename GG::GIDL_A_v>& o
20  ) { A* a = o.getOrigObj();   me()->obj = a; }
21  /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/
22
23  operator A*() const { return static_cast<A*>(me()->obj); }
24  template < class GG > operator GG() const {
25    GG g; // test GG subclass of the current class!
26    if(0) { A* ob; ob = g.getOrigObj(); }
27    void*& ref = static_cast<void*&>(g.getOrigObj());
28    ref = GG::_narrow(this->getOrigObj()); return g;
29  }
30  void setOrigObj(A* o)   { me()->obj = o; }
31 public:
32  A*&         getOrigObj()    { return static_cast<A*&>(me()->obj);}
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 };
```

Fig. 10. The Base Class for the GIDL Wrappers Whose Types Are GIDL Interfaces.

Figure 10 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 `GIDL_Type`, the supertype of all GIDL types, which has no state and all its constructors are declared `protected`. The erased object is available via the `me()` macro in line 1: the self type of this object is whatever its type parameter `T` is instantiated with. 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 initializers, assignment operators and accessor functions that work over various CORBA and GIDL types, allowing the user to manipulate in a natural way GIDL wrapper objects.

It is perhaps important to draw attention at this point to the fact that CORBA IDL inheritance maps to C++ inheritance. 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 assignment `A* a = o.getOrigObj();` statically checks that (i) the parameter `o` belongs to a valid GIDL type, (ii) whose erasure `GG::GIDL_A` is a subclass of `A` and hence it holds that, with respect to the GIDL specification, (iii) the GIDL type instantiation of `GG` is a subclass of the GIDL type instantiation of `T`. This irregular use of type members `GG::GIDL_A` and `GG::GIDL_A_v` in the `BaseObject` constructor is one of the GADT characteristics. The mapping also defines a type-unsafe cast operator (lines 24-29) that allows the user to cast an object to a more specialized type. The implementation statically checks that the result's type is a subtype of the current type (w.r.t. the GIDL specification).

4.3 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:

- As far as the representation is concerned, each GIDL wrapper stores precisely one (corresponding) CORBA-type object: its erasure. This is a scalability concern. Keeping the object layout of the GIDL wrapper small is important.
- In terms of functionality, the GIDL wrapper features only the casting functionality associated with its type; these 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 11 shows

```
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. 11. Naive Translation for the C++ Mapping.

such an attempt. If each GIDL wrapper stores its own representation as an object of its corresponding CORBA type, the wrapper object layout may become significantly large. 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 10). 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.

Figures 7 and 12 show our solution: the first depicts the inheritance graph of the types used to implement the GIDL wrappers, the second details on the implementation. 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 (i.e. `init`).

`Leaf_P` resembles Scala "traits" [13]: it holds no state, does not provide constructors, but only provide the services promised by the GIDL type and they implement the GIDL specification inheritance hierarchy. `Leaf_P` needs an *accessor* (`getObject_Leaf`) that returns the erased UA object; the CRTP idiom [4] is used again to avoid a virtual call. This implementation is correct since any UA types respect the inheritance graph of the GIDL specification (i.e. `::Leaf` is a subclass of `::BinTree`), and the `SELF` type is propagated via inheritance.

Finally, the `Leaf` wrapper class aggregates the casting functionality and the services promised by the GIDL specification by inheriting from `BaseObject` and `Leaf_P` respectively. It rewrites the functionality that is not subject to

[4] We thank one of the anonymous reviewers for this observation.

```
template<class S,class K,class D> class BinTree_P { ... }
template<class SELF,class K,class D> class Leaf_P :
                      public virtual BinTree_P<SELF,K,D>{
  protected:
    ::Leaf* getObject_Leaf()
      { return static_cast<SELF*>(this)->getOrigObj(); }
  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 Leaf_P< Leaf<K,D>, K, D >,
     public BaseObject<Leaf<K,D>,::Leaf,::Leaf_var>
{
  friend class BaseObject<Leaf<K,D>,::Leaf,::Leaf_var>;
  protected:
    typedef Leaf<K,D> T;
    typedef BaseObject<T,GIDL_A,GIDL_A_v> BT;
    ::Leaf* obj;
  public:
    Leaf()                              { }
    Leaf(const GIDL_A_v a)              { BT::initBO(a); }
    Leaf(const GIDL_A* a)               { BT::initBO(a); }
    Leaf(const T & a)                   { BT::initBO(a); }
    Leaf(const Any_GIDL & a)            { BT::initBO(a); }
    template <class GG> Leaf( const
      BaseObject<GG,typename GG::GIDL_A,typename GG::GIDL_A_v>& a
    )                                   { BT::initBO<GG>(a); }
    /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ETC. ***/
};
```

Fig. 12. Part of the C++ Generated Wrapper for the GIDL::`Leaf` Interface. `::Leaf` and `::Leaf_var` are CORBA types.

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>`.

Although not supported by the current implementation we remark that the virtual inheritance between, say, `BinTree_P` and `Leaf_P` can be eliminated. IDL and thus GIDL allow neither method refinement in subtypes, nor it allows that two methods who are in a subtype relation belong to the same interface (via

inheritance). Thus, even under diamond-like inheritance, for example `K<S> : A<S>` and `L<S> : A<S>` and `X<S> : K<S>, L<S>`, where `A`, `K`, `L`, and `X` are the traits components, `S` is the CRTP self type, and the method `void fun()` is declared in `A`, we could provide access to `fun` by adding it to `X`'s C++ wrapper: `void fun(){static_cast<A<S>*>(static_cast<K<S>*>(this))->fun();}`.

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 10. 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 constructors, 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 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 interface guarantees that the creation of GIDL objects will not yield type-errors. It remains to examine method invoca-

```
// 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. 13. GIDL Interface and the Corresponding Stub/Skeleton Wrappers for `foo`.

tions. It is trivial to see from the implementation of the `_lift`, `_narrow`, and `_any_narrow` functions (Figure 10) 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: `b=G::_narrow[G]`$\circ$`G::_lift[A*] (a)` $\sim$ `a` *iff* `b` and `a` are pointers to objects of CORBA-type `A` and they refer to the same server object.)

Figure 13 presents the GIDL operation `Foo::foo()` and its C++ stub/skeleton mapping. The stub wrapper translates 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 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 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 – the client can still be on a remote machine.

We find C++'s *Standard Template Library* [19] (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. 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 [10,19] 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. For a while now these constraints have been defined as English annotations in the standard; the imminent language enhancement with "concepts" [4] will bring the needed formalism to express them at the interface level.

Figures 14 and 15 present excerpts of the GIDL iterators and vector interfaces respectively. We simulate *selftypes* [14] by the use of an additional generic type, `It`, bounded via a mutual recursive export based qualification (`:-`). This abstracts the iterator's functionality: `InpIt<T>` exports the `==(InpIt<T>)` method, while `RaiIt<T>` exports the `==(RaiIt<T>)` method. An *input iterator* has to support operations such as: incrementation (`it++`), dereferencing (`*it`),

```
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> >{};
```

Fig. 14. GIDL Specification for STL Iterators. @p/@a – prefix/postfix operators.

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 function-

```
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    assign(in T obj, in long index);
  T       getAtIndex   (in long index);
  void    swap         (in STLvector<T, Ite, II> v); //....
};
```

Fig. 15. GIDL Specification for STL Vector.

ality described in the STL specification: `RI` is expected to share *structural similarity* [1] with its qualifier `RandAccessIter`. Note that, unlike its underlying architecture, GIDL supports operator and method overloading. However, at this moment GIDL does not support type-parameterised functions, as this would require modifications of the UA stringified reference, and just-in-time recompilation of the server. For example, `STLvector` does not support the STL vector method: `template<class II> void assign(II fst, II last)`.

As observed in [11], 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 a STL-helper "base" class of their corresponding iterator category. For example `InputIter` stub extends the class `input_iterator<T,int>`, which defines the `iterator_category`, `distance_type`, and `value_type` members.

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. For example, Section 3.2 identifies, and Section 5.3 provides solutions for some of the problems derived from exporting STL over a distributed environment. These problems steam from the use of pointers and reference types, which are not supported by GIDL/IDL due to the distributed nature of the address space. Our approach is to use a *black-box* translation scheme that wraps the library objects into GIDL objects and to study what other constructs are required to enforce the library semantics.

```
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);
    }
};
```

Fig. 16. GIDL Vector and Input Iterator Server Implementations.

```
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;
```

Fig. 17. GIDL Client Code that Uses the STL Library.

Figure 16 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[5]. 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.

The GIDL code in Figure 17 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.

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 exe-

[5] While it would be more natural to use a pointer instead of an `unsigned long`, GIDL or IDL do not export a pointer type.

cuted, 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. Since `WrapType` is a CORBA type, its implementation lives in the server space. Either one of reading or writing the `T`-type object corresponding to the iterator's current element requires two remote calls from the client side: the first returns the `WrapType` object (i.e. `WrapType<T> wrap = *rai;`), and the second one either reads or sets the `T`-type value of `wrap` (i.e. `wrap.set(val)`). Furthermore, the user needs to instantiate the iterators and vectors over the `WrapType<T>` type, which is not natural as the GIDL interface specifies an iterator of `T`-type objects. The next section discusses how to remedy 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 18 illustrates our approach. `RaiIt_Lib` refines the behavior of its corresponding GIDL wrapper `RAI` to match the library semantics, and the same for `STLvect_Lib` and `STLvect`.

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 17 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.

```
template<class T,class Iter> class TrapperRAI  {
  protected:
    Iter it; Long_GIDL ind;
  public:
    TrapperRAI(const Iter& i, const GIDL_Long& index)
      { it = i; ind = index; }

    void T()                              { return it[ind];             }
    void typename T::GIDL_A()             { return it[ind].getOrigObj();}
    void operator=(const T& t)            { it.assign(t, ind);          }
    void operator=(typename T::GIDL_A a){ T t(a); it.assign(t, ind);  }
};

template<class T> class RaiIt_Lib :public GIDL::RAI<typename T::Self> {
  private:
    typedef GIDL::RAI<typename T::Self>           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()            { }
    RaiIt_Lib(const It& r) { GIDL_BT::initBO(r.getOrigObj()); }
    RaiIt_Lib(const RaiIt_Lib<T>& r)
                 { GIDL_BT::initBO(r.cloneIt().getOrigObj()); }

               operator It() { return static_cast<It>(*this); }
    TrapperRAI operator*  ()
         { return TrapperRAI<T,It>(static_cast<It>(*this),0); }
    TrapperRAI operator[] (Long_GIDL i)
         { return TrapperRAI<T,It>(static_cast<It>(*this),i); }

    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 STLVect_Lib : public
 GIDL::STLvector<typename T::Self,typename RI::Self,typename II::Self>{...}
```

Fig. 18. Library Iterator Wrapper and Its Associated Trapper.

Second, `RaiIt_Lib` defines a new semantics for the `*` and `[]` operators that now returns `TrapperRAI` objects. At a high-level, the *trapper* can be seen as a proxy for performing read/write operations. Its design resembles the lazy evaluation technique, as it captures the container and the index that needs to be read or updated. When the operation to be performed (read or write) is known the trapper invokes the corresponding container's method. The read operations is called when an automatic conversion is required; the write is called from the trapper's assignment operators. This technique solves the problem encountered at line 14 in Figure 17. Also, for most parts the use of the trapper is transparent for the user: the following work as expected under `gcc4.4.1`:

```
RaiIt_Lib<Long_GIDL> rai1 = ...; Long_GIDL g(3); long l = 4;
*rai = l; rai[1] = g; l = *rai; g = rai[2]; cout<<g<<" "<<l<<endl;
```

Unfortunately, for some STL iterations, the use of trappers, or in general of any proxy, is illegal. For example the STL forward iterator concept requires that operator* must return a true reference to the underlying value [6] – which in fact is not possible under CORBA due to the heterogeneous address space.

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. The use of the `Self` abstract type member in the extension clause of the iterator/vector library wrappers ensures that the library and GIDL wrappers hierarchies remain separated. For example `RaiIt_Lib<RaiIt_Lib<Long> >` inherits from `RAI< RAI<Long> >`. The consequence is that all the inherited operations have results belonging to GIDL types, and thus no un-necessary cloning operations are performed:

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

Further on, dereferencing/updating an element of a "composed" library iterator works as expected. For example, consider the following instructions:

```
RaiIt_Lib<RaiIt_Lib<Long> > it; **it=5;
```

The first `*` operation creates a trapper object belonging to the

```
TrapperRAI< RaiIt_Lib<Long>, RAI< RAI<Long> > >
```

type that inherits from the `RaiIt_Lib<Long>` type. Therefore, the second `*` operation is applied on a library wrapper object, and thus the update succeeds.

[6] We thank one of the anonymous reviewers for this observation.

```
class Long_GIDL                                { long obj; /* ... */};
template <class T, class Iter> class Trapper /*: public T*/ { ... };
template <class S, class A   > class BaseObject             { ... };
template <class S, class T   > class GIDL_Iter_T            {
  SELF* me       ()         { return static_cast<SELF*>(this);    }
  void operator++()         { me()->getOrigObj()++;               }
  T    operator* ()         { return T(*(me()->getOrigObj()));    }
  T    operator[](long i)   { return T(me()->getOrigObj()[i]);    }
  void setCurrent(T& t, long index){me()->getOrigObj()[index] = t;}
  void operator=(const SELF& it){me()->setOrigObj( it.ptr ); }/*VIRTUAL*/
  // ...
};
template<class T> class GIDL_Iter:public GIDL_Iter_T<GIDL_Iter<T>,T>
    public BaseObject <GIDL_Iter<T>, typename T:GIDL_A*>    {
  typename T::GIDL_A* obj; // ...
}
template<class T> class LIB_Iter : public GIDL_Iter<T> {
  typedef Trapper<T,GIDL_Iter> TRAP; typedef GIDL_Iter<T> It;// ...
  TRAP operator* ()      { return TRAP(static_cast<It*>(this),0); }
  TRAP operator[](long i){ return TRAP(static_cast<It*>(this),i); }
}
void testOverhead() {
  // test 1-D Iterator
  GIDL_Iter<Long_GIDL> gidl_iter(SIZE);
   LIB_Iter<Long_GIDL> lib_iter=gidl_iter;
  for(int k=0; k<REPEAT; k++) {  lib_iter.setOrigObj( beg_iter );
    for(int i=0; i<SIZE; i++)  {
      tmp = *lib_iter; *lib_iter = i; ++lib_iter; sum += tmp;
  } }
  // test 2-D Iterator
  for(long k=0; k<REPEAT; k++) {
    GIDL_Iter< LIB_Iter<Long_GIDL> > gidl_gidl_int(SIZE2);
     LIB_Iter< LIB_Iter<Long_GIDL> > lib_lib_int = gidl_gidl_int;
    for(long i=0; i<SIZE2; i++) {
      LIB_Iter<Long_GIDL> lib_int(SIZE1); lib_lib_int[i] = lib_int;
      for(long j=0; j<SIZE1; j++)
        { Long_GIDL ipj(j+i); *lib_int = ipj; ++iter2_int; }
      /* ... */ delete iter2_iter1_int[i].getOrigObj();
    }           delete iter2_iter1_int.getOrigObj();
} }
```

Fig. 19. Testing Program for Measuring the GIDL-Like Extension Overhead.

This section estimates the overhead introduced by our extension mechanism. Unfortunately, trying to measure this overhead directly, by running CORBA programs on the same machine fails to give a relevant answer mainly because the overhead of the UA (CORBA) is huge. The running times of GIDL-based and CORBA-based application are the same, and are order of magnitude higher than that of the C++ application written for a single-address space.

In order to meaningfully estimate this overhead we use a single-address space program, shown in Figure 19, that uses types that resemble the ones used by our GIDL/library extension. For example `GIDL_Iter`, and `LIB_Iter` correspond to the `RAI` and `RAI_Lib` iterators, and `Trapper` corresponds to `TrapperRAI`. (Note that the erased state of `GIDL_Iter`, denoted by the `obj` member are normal – as oposed to CORBA – objects.)

We measure the overhead based on two tests, depicted in the `testOverhead` function. The first test uses a one-dimensional iterator and while traversing it it sums up and then sets its elements. The second test uses a two-dimensional iterator that is traversed to set new values to its elements. Allocation and dealocation are performed at entry and at the loop's exit respectively – this reduces somewhat the overheads since the cache layout would probably be poor. We compare against (normal) C++ code performing similar operations directly upon one- and two-dimensional arrays storing `long` values.

Since trappers are used extensively in our tests, we use three types of trappers to evaluate our extension-mechanism design choices against alternative solutions. *Perf Trapper* is the one used by GIDL – see Figure 18. *Mixin Trapper* is the one that uses mixin [18] programming – i.e. inherits from its type parameter `T`. It estimates the overhead associated to inheritance when the base class has non-void state. The third one, *Virt Trapper* uses mixin programming (`: public T`) and in addition requires the assignment operator from the base class `T` to be declared virtual. It estimates the cost of virtual calls if the extension architecture would have used them.

Table 2 shows the ratio between the running time of the extension and the running time of the equivalent, simple (usual) C++ code, for each of the three trappers that we used. The iterator's range size is varied from 2000 to 20000000 – as this increases the cache lines are broken and the overhead decreases.

We observe that our extension – *Perf Trapper* – does not introduce any overhead probably because it is compiled to nearly the same code (the results slightly fluctuate in both directions, so we have chosen to show the ratio 1). This demonstrates that trait-like inheritance (stateless inheritance) is efficient.

Estimation of the GIDL Slow-Down in a Single-Address Space						
	Trapper Type	2 10^7	2 10^6	2 10^5	2 10^4	2 10^3
1D-Iterator	Virt Trapper	6.70	7.72	9.93	10.37	10.06
	Mixin Trapper	1	1	1	1.05	1.21
	Perf. Trapper	1	1	1	1	1
2D-Iterator	Virt Trapper	2.38	2.54	3.81	11.59	8.65
	Mixin Trapper	1	1.02	1.54	5.42	3.68
	Perf. Trapper	1	1	1	1	1

Table 2
This table shows the *running-time ratio* between a GIDL-like translation of iterators in a single-address space and "optimal" C++ (STL) code. The implementation of the 1D-Iterator and 2D-Iterator is depicted in Figure 19. The size of the iterator range is varied from 20000000 to 2000.

Perf Trapper: the one in Figure 18 targeting performance.
Mixin Trapper: inherits from the type it represents (`Trapper : public T`).
Virt Trapper: it is a mixin trapper but the `=` operator is virtual in its base class.

Mixin Trapper's overhead on the 2D-Iterator test is significant, and the running time ranges from being 5.42 time slower to being as fast as the C program. Surprisingly this overhead is not exhibited on the 1D-Iterator test.

Virtual Trapper's overhead is significant on both applications, the extension running 10.06 to 2.38 times slower than the original code. The test programs were compiled with the *gcc* compiler version 4.4.1 under the maximum optimization level (`-O3`), on a 3.2 GHz Pentium 4 system.

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: the CRTP idiom, *member/associative types* and Scala-like *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'89)*, pages 273–280, 1989.

[2] James O. Coplien. Curiously Recurring Template Pattern. *C++ Report*, pages 2427, 1995.

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and Generative Programming. In *ACM SIGSOFT Software Engineering Notes*, Volume 24, Issue 6, pages 2–19, ISSN:0163-5948, 1999.

[4] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 291–310, 2006.

[5] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated Types and Constraint Propagation for Mainstream Object-Oriented Generics. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 1–19, 2005.

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

[7] A. Kennedy and C. V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 21–40, 2005.

[8] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'01)*, pages 1–12, 2001.

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

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

[11] C. E. Oancea and S. M. Watt. Parametric Polymorphism for Software Component Architectures. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 147–166, 2005.

[12] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger. Overview of the Scala Programming Language. *Technical Report IC 2004/64*, EPFL Lausanne, Switzerland, 2004.

[13] M. Odersky, V. Cremet, C. Rockl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of ACM European Conference on Object-Oriented Programming (ECOOP'03)*, 2003.

[14] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57, 2005.

[15] OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision 2.4 (October 2000), OMG Specification, 2000.

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

[17] J. Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley and Sons, 2000.

[18] Yannis Smaragdakis and Don Batory. Mixin-Based Programming in C++. In *Proceedings of the Generative and Component-Based Software Engineering Symposium (GSCE'00)*, pages 163-177, 2000.

[19] Alexander Stepanov and Meng Lee. The Standard Template Library. *HP Laboratories Technical Report 95-11(R.1)*, November 14, 1995. http://www.stepanovpapers.com.

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

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

[22] 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.

[23] H. Xi, C. Chen, and G. Chen. Guarded Recursive Data Type Constructors. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235, 2003.

[24] E. Ernst. Family Polymorphism. In *Proceedings of the ACM European Conference on Object Oriented Programming (ECOOP'01)*, pages 303-326, 2001