Automatic Generation of Bridging Code for Accessing C++ from Java

Andreas Schade
IBM Research Division, Zurich Research Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
Tel.: +41 1 724 84 05, Fax.: +41 1 710 36 08
email: san@zurich.ibm.com

Abstract

Java is becoming increasingly important as a programming language for applications based on the network-centric computing paradigm. While more and more applications are written in Java, a large number of working C++ implementations for standard tasks still exist. Based on a domain model for language interoperability, this paper describes a mechanism for automatic bridging code generation between Java and C++. The structure of this bridging code is derived from the basic model. The paper gives a description of how C++ features are remodeled, and how C++ definitions are automatically mapped to Java.

— Research Paper —

Keywords: Network computing, Programming environments, Language design and implementation.

1 Introduction

The Java programming language is one of the keys to a new programming paradigm which uses the network as a computing resource. In such a network computing scenario, premanufactured components for a variety of purposes are maintained in software repositories in the network and can be downloaded and executed on demand drawing on the network for data and additional computing power. There are numerous advantages to this approach such as new software procurement schemes, simplified maintenance and reduced resource requirements.

Java is a well designed object-oriented language which provides powerful programming features. Over and above that, its networking and distribution facilities make Java increasingly important as a programming language for "real" problems rather than for making Web pages more appealing. Whereas many applications are being written in Java, there are still many software components, mainly in form of libraries, implemented in C++, which solve numerous standard tasks such as protocol handling and graphics. It would be wasteful to have to re-implement those libraries in Java. What is needed is an automatic

method by which the gap between Java and C++ is bridged, and existing C++ libraries can be transparently used from Java applications.

The Java programming language allows access to C code through its native code interface. C++ code cannot be used this way since it is not possible to instantiate C++ classes, call methods, access member variables of objects, or destroy C++ objects that are not needed anymore. Based on a domain model for language inter-operability, this paper illustrates how the necessary bridging code can be generated automatically, describes the structure of this bridging code and introduces a tool which produces this code based on C++ class definitions.

The remainder of this paper is organized as follows. In section two the domain model for language inter-operability is presented and the layered structure of the bridging code is derived. The sections three to five provide a top-down overview of the Java bridging code layer. Section three describes how C++ classes are represented in Java and how the inheritance relationships among them are modeled. The mapping of the C++ class members is explained in section four. Section five illustrates the data type mapping scheme between C++ and Java. The C++ part of the bridging code is described in section six and some implementation issues concerning the automatic generation of bridging code are outlined in section seven. Concluding remarks are given in section eight.

2 A model for language interoperability

Language interoperability between two different programming environments involves the crossing of domain boundaries. In fact, there are two different types of domains that are being crossed when interfacing C++ to Java: the language domain and the address domain.

A **language domain** comprises of all the entities written in the same programming language. Crossing the language domain boundary between C++ and Java means that there must be a method of remodeling the language features of one domain in the other.

The Java native code interface provides the only way "out" of the Java domain, but only normal C code, rather than C++ code, can be accessed directly. Therefore, object-oriented features such as inheritance relationships and polymorphism are not preserved beyond the Java domain boundary. This C code which merely consists of stub functions, constitutes the intermediate representation between the Java and the C++ language domain. In order to achieve inter-operation between Java and C++, another domain boundary (between C and C++) must be crossed. This situation is illustrated in figure 1.

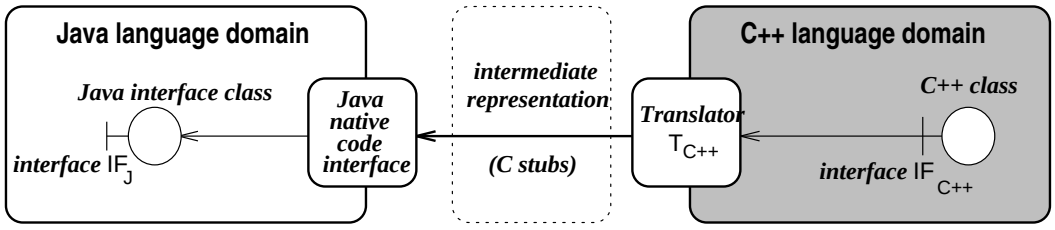

Figure 1: Crossing language domain boundaries between Java and C++.

Accessing a C++ class from Java means making its members accessible in the intermediate C format defined by Java's native code interface (stubs) and providing a Java

interface containing native methods which invoke these C stub routines. The first crossing is achieved by the translator T_{C++} which connects the C++ class interface IF_{C++} with the intermediate representation of stub functions, while the second transition is accomplished by the Java native code interface itself which connects the intermediate representation with a corresponding Java class. This class provides an interface IF_J, which can be used from any Java program that wishes to use the original C++ class. This model leads to a layered architecture of bridging code as depicted in figure 2.

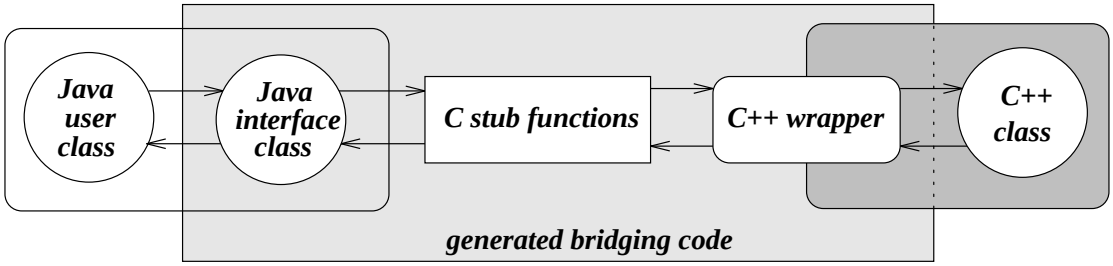

Figure 2: The layered bridging code architecture.

The bridging code which must be generated consists of three layers. Each of these layers is situated in a separate language domain. The Java interface class contains the native access methods which invoke C stub functions of the intermediate layer via the Java native code interface. The stub functions in turn call C++ wrapper routines which act in two roles, as a translator between C and C++ and as a client for the C++ class to be interfaced.

While interoperability between language domains is mainly dealt with at compile-time, the passing of the **address domain** boundaries reflects the run-time aspect of the entire inter-operation problem. Each invocation of a native Java interface method at run-time is forwarded to a class member on the C++ side. Again, there is some transformation necessary as information flows from one side to the other. This is done by address translation as illustrated in figure 3.

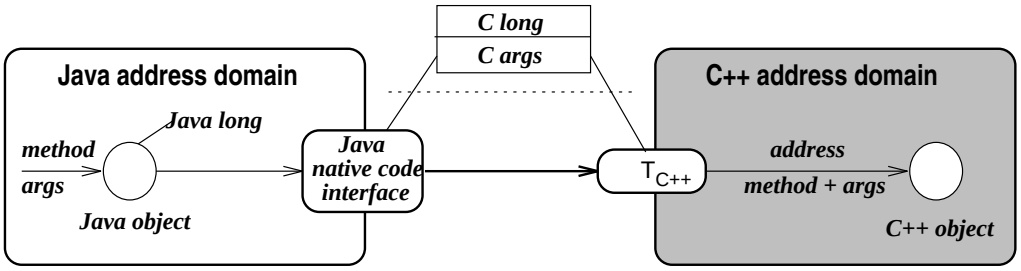

Figure 3: Crossing address domains between Java and C++.

The association between the Java interface and its corresponding C++ instance is maintained by a reference stored in the Java interface object. If a native method of this object is called, the translator T_J (the Java native code interface) maps the Java call onto a call of a C stub function. The parameters are passed as a type-independent stack. The first parameter is a pointer to the calling Java object. The stub routine transforms the parameter stack to a call of a wrapper function which is part of the translator T_{C++}. The

wrapper function intercepts the call and extracts the address of the corresponding C++ object from the first parameter. After type transformation of the remaining parameters the corresponding member definition is accessed at the C++ object[1]. It should be noted that the described translation scheme is bi-directional since the type mapping must also be applied to return values passed from C++ to Java.

3 Modeling C++ in Java

The basic idea is to represent each class of the C++ library to be interfaced by a dedicated Java class, called the *Java representative class*, which contains access points for all member definitions of the represented C++ class. Whenever components of the C++ class are to be used, its Java access method is invoked. Using the facade design pattern [GHJ+95], a number of problems must be solved. There must be exactly one Java representative object for each C++ class instance. This one-to-one relationship must be preserved during the life time of the C++ object. The C++ class gets instantiated when its Java representative class is instantiated. It ceases to exist when the Java garbage collector decides to remove the corresponding Java representative object.

Moreover, inheritance relationships between the C++ classes must be modeled among their Java representative classes. If a C++ class B inherits from another C++ class A, also the Java representative for B must inherit from the Java representative class for A. This means, that operations defined in the C++ class A can be invoked on the Java representative for B. Furthermore, if the Java representative class for B is instantiated, there must not be any superfluous instances of the C++ class A created, so that the one-to-one relationship still holds.

3.1 Java Representative Classes

As motivated by the address domain model shown in figure 3, all Java representative classes contain a private member variable `handle` which is used to store the reference to the C++ instance. This private member variable is of the Java type `long` since the reference is in fact the address of the associated C++ object. The handle member variable gets set in the constructor(s) of the Java representative class. Note, the handle variable is not a pointer as there is no pointer concept in Java. It is a variable of type long which is semantically meaningless if only the Java side is considered. It is turned into a pointer when passed across the boundary to C++ domain.

By inheriting from a special Java base class in which the member variable `handle` is defined, the C++ reference becomes common to all Java representative classes.

3.2 Inheritance

Single inheritance among C++ classes must also be modeled between their corresponding Java representatives. Due to the required one-to-one relationship between the C++ instances and their Java representative objects, the Java constructors would call the

[1] As there is a translator per member definition, the name of this member is not passed as a parameter.

corresponding C++ constructor. This scheme does not have the desired effect if C++ inheritance chains are mapped to Java.

As shown in figure 4, the instantiation of a Java class B (1) the invocation of one of B's constructors would cause the instantiation of its parent Java class A (2), i.e., one of A's constructors would be executed. This constructor in turn would invoke its corresponding constructor of the C++ class A (3). If the construction of the parent class A is finished, the originally called constructor of the Java class B would call its corresponding C++ constructor (4), which again would, because of the order of constructing C++ classes, cause the constructor of the C++ class A to be executed (5). As a result of step (3) there is one superfluous instance of the C++ class A.

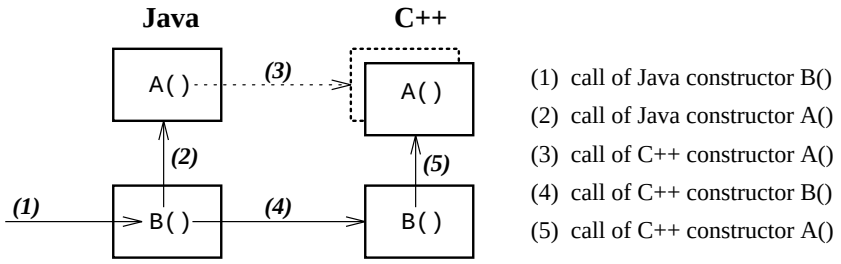

Figure 4: The problem of dangling C++ instances.

In the case of longer inheritance chains the situation is even worse. Generally speaking, in the case of an inheritance chain of the length n consisting of the classes A_1, A_2, ..., A_n (with A_1 being the top-most and A_n the most specific class), the constructor of a C++ class A_i $(1 \leq i \leq n)$ would be executed $n - (i - 1)$ times which would lead to $n - i$ superfluous instances. In total this model of related constructors results in $(n - 1) * n/2$ instantiations of the classes A_1 through A_{n-1}.

To prevent these superfluous instantiations, a special constructor in each Java representative class has to be introduced which is distinct from all other constructors that have a C++ link. The uniqueness of this special constructor is guaranteed by assigning a parameter of the Java class type `_NullConstructor_` Given that such a class does not exist in the C++ library, there cannot be any other Java constructor with the same signature. If a subclass in Java is instantiated, its constructor calls explicitly the special constructor of the parent class using the notation `super((_NullConstructor_)0)` avoiding the problem of instantiating these extra objects.

Using this scheme, the only constructor with a C++ link called is the constructor of the Java representative class which was meant to be instantiated in the first place. Since the top-most class of each inheritance chain on the Java side inherits from the previously mentioned base class of all Java representatives, the handle member variable is defined only once in each instance of an inheritance chain. At run-time an invocation of a method defined in any parent class is forwarded to the C++ child instance, and then dispatched to the appropriate base class instance according to the normal C++ resolution process as illustrated in figure 5.

Java does not support the concept of multiple inheritance. Therefore, multiple inheritance relationships between C++ classes are currently not remodeled. A possible approach is to introduce so-called *parent proxies* which are C++ classes representing the unified

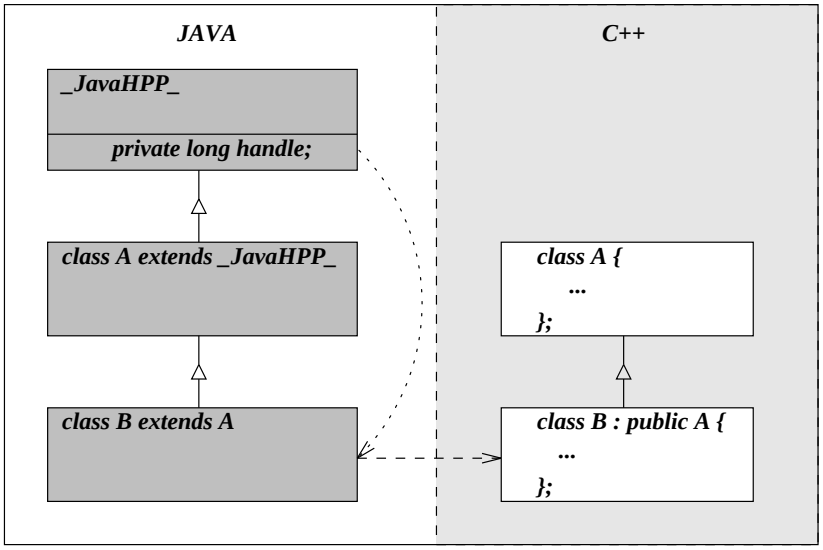

Figure 5: Preserving of C++ inheritance relations.

interface of all parent classes of a multiple inheritance relationship in C++. By this approach the multiple inheritance is conceptually mapped to a single inheritance relation between the original child class and the parent proxy which can be handled using the approach described above.

4 Mapping of C++ class member definitions

Having described the relationship between Java and C++ at the class level, this section is concerned with the public member definitions inside the C++ classes and their mapping to the target language. Public member definitions which can occur in a C++ class can be divided into the following categories

- polymorphic constructors,
- destructors,
- static and non-static polymorphic methods,
- static and non-static member variables of scalar or array types, and
- operator definitions and conversion functions.

The approach of the interfacing mechanism is to provide access points for each of the above definitions in the C++ class in the form of Java methods in the corresponding Java representative class. This mapping of C++ class members can also be used for static C++ definitions such as global variables, functions, or non-member operators which are defined outside any C++ class. For these, a special Java representative class will be generated providing the (static) access methods for these definitions. In general, for a C++ library consisting of n classes, at most $n + 1$ Java representative classes would be created.

4.1 Methods and Constructors

Methods of C++ classes are mapped to methods in the corresponding Java representative class. The method in the Java class will have the same name as its C++ counterpart. The mapping of the signature of C++ methods to Java is determined by the type mapping model as described in section 5. C++ methods defined as static are mapped to static Java access methods in the corresponding representative class. They can be invoked without having to instantiate the Java class before.

In order to map polymorphism in C++ classes to Java, an additional level of indirection must be introduced. This means that the Java access methods which can be used by the Java application are not implemented as native methods. The reason for this is that the Java native code interface only allows access to C code, and is therefore unaware of polymorphism. The solution to this problem is to use the name mangling scheme known from C++ linkers. The Java access method which is part of the API presented to the Java application is only a wrapper routine around a call to a private native method. The name of this private native method is the result of the mangling process applied to the method's name and signature. Due to the signature mangling the names of the private native methods are unique amongst the set of polymorphic version of a certain Java access method.

The following methods of a C++ class `A`

```
int M1 (void);
int M1 (int);
```

would be mapped to two pairs of Java methods consisting of a public access method and the private native method

```
private native int M1_p();
public int M1() { return M1_p(); }

private native int M1_pI(int arg0);
public int M1(int arg1) { return M1_pI(arg1); }
```

From the mapping viewpoint constructors are special methods without return types, which can be dealt with in the same way as normal methods. Polymorphic constructor versions in C++ are mapped using the mangling scheme as for methods. The assignment of the handle variable described in section 6.1 is not performed on the Java level. Catering for polymorphic constructors in the described manner also solves the problem that Java constructors cannot be native. Not the constructor itself but its private mangled routine will be native.

4.2 Member Variables

Member variables of C++ classes are mapped to a pair of access functions for retrieval and modification in the Java representative class. If the variables are declared constant, only the read function will be generated.

C++ member variables of scalar types are mapped according to the following scheme.

```
<C++ type> <member>;
```

$$\Downarrow$$

```
<Java type> get<member>();
void        set<member>(<Java type>);
```

The dependency between <Java type> and <C++ type> is explained in section 5. Similar to static methods, the access methods for static C++ member variables will be static as well.

Array type member variables are handled differently, due to Java's storage model for arrays. Instead of generating access functions that affect the whole array, `get` and `set` methods on the level of array elements are generated. The user is responsible for supplying index values within the valid range.

```
<C++ type>[ ][ ]...[ ] <member>;
```

$$\Downarrow$$

```
<Java type> get<member>(int i1, int i2, ..., int in);
void        set<member>(<Java type>,
                        int i1, int i2, ..., int in);
```

Access methods for array-type member variables require as many index parameters as the dimension of the C++ array to be interfaced. This mapping scheme allows using the same type mapping scheme as for scalar C++ types.

4.3 Operator Definitions

Aside from the following operators

```
::        .*        .        ?:
```

which cannot be overloaded in C++, all publicly defined member definitions of overloaded C++ operators are mapped to Java access methods. The notation capabilities, such as infix or postfix notation, of the particular operator are lost but the operator definitions remain accessible via the functional notation of the access methods. It is also possible to map non-member operator definitions using the special Java representative class for static definitions as a container class. Operator access methods contained in the Java representative class are created using the following conventions.

```
<C++ type> operator <symbol> (<C++ signature>);
```

$$\Downarrow$$

```
<Java type> operator<mapped symbol> (<Java signature>);
```

The C++ operator signs are mapped to special names in Java which become part of the access method name. Postfix and prefix versions of the increment and decrement operators are distinguished. The mapping of the C++ operator signs to their corresponding Java names is summarized in table 1.

C++ Symbol	Java Suffix	C++ Symbol	Java Suffix
+	PLUS	!=	NEQ
-	MINUS	&&	AND
*	MULT	||	OR
	PTR	<<	SHL
/	DIV	>>	SHR
%	MOD	-=	MINUSASSIGN
^	BXOR	/=	DIVASSIGN
&	BAND	%=	MODASSIGN
	ADR	+=	PLUSASSIGN
|	BOR	^=	BXORASSIGN
~	BNOT	&=	BANDASSIGN
!	NOT	|=	BORASSIGN
,	COMMA	*=	MULTASSIGN
=	ASSIGN	<<=	SHLASSIGN
<	LT	>>=	SHRASSIGN
>	GT	[ ]	INDEX
<=	LE	( )	FUNCCALL
>=	GE	==	EQ
++	POSTINCR	--	POSTDECR
	PREINCR		PREDECR

Table 1: Mapping of C++ operator symbols to Java suffixes.

For instance, a member definition of the C++ operator `--` (prefix decrement) defined as `A& operator--(A&);` contained in a C++ class `A` would therefore be mapped to a Java access method `A operatorPREDECR();` in its Java representative class.

With respect to the method of mapping operator definitions, mappable C++ operators are divided into six categories depending on the operator signature.

- *N*-ary operators: the function call operator `()`.
- *Binary* operators: `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `,`, `=`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&&`, `||`,`>>`, `+=`, `>>`, `-=`, `/=` , `%=`, `^=`, `&=`, `|=`, `*=`, `<<=`, and `>>=`.
- *Unary prefix* operators: `+`, `-`, `*`, `&`, `~`, `!`, the prefix increment operator `++`, and the prefix decrement operator `--`.
- *Unary postfix* operators: the postfix increment operator `++`, and the postfix decrement operator `--`.
- The *index* operator `[ ]`.
- The `new` operator.

The operator `->` (class member access) cannot be mapped to Java because of its special semantics. In C++, the operator `->` may be overloaded as a unary member function, which either returns a pointer to a class or an object of or a reference to a class. In the former case (pointer to class) the ordinary member selection semantics of the returned pointer is applied, whereas in the latter case (object or reference) the class member access function for objects of the returned type is called. The process is then applied recursively until

either a pointer type is returned or an error is detected since no suitable class member access function could be found.

The identifier following the `->` must be a member of the object to which a pointer is finally returned. If one were to attempt to map this concept to Java, a *call by name* semantics would be required for the operator access method which would have to pass the member identifier as an argument to the wrapper layer. Such call by name semantics, however, does not exist. Therefore the class member access operator cannot be mapped.

4.4 Destructors

C++ objects can have at most one destructor method, which is executed before the instance is removed. This can be triggered either explicitly by calling the destructor method `~<class>` or by using the C++ delete operator. The delete operator can only be overloaded having `void *` as the first argument and `size_t` as an optional second argument. Regardless of the actual signature, `delete` is always used in the normal manner `delete <pointer>`. Therefore it is not mapped according to the ordinary operator mapping scheme.

Destructors are the only C++ components for which a slightly different mapping scheme has to be deployed. Destructors for C++ objects must be attached to the Java garbage collection mechanism. For this purpose, Java provides so-called finalize routines. Therefore, C++ destructors will be called by a native finalize routine which exists for each Java representative class.

The finalize routine is called by the Java run-time system before the garbage collector removes any unreferenced instance of a Java class. Connecting the C++ delete operator to a native Java finalize routine guarantees that the C++ instance is removed if its Java representative object is deleted.

5 The Type Mapping Scheme

Type conversions applied in the mapping process influence the representation of C++ definitions in Java. The type conversion mechanism maps C++ data types to Java data types. Since the set of basic data types available in Java is smaller than in C++, there are cases in which C++ data are not representable in Java. The problem is that because of polymorphism, the type conversion scheme must be bijective; otherwise, there could be name clashes when mangling method or operator signatures.

The type conversion scheme distinguishes between three categories of C++ data types

- *Directly mappable* C++ types which have a direct representation in Java. These are
  - simple data types such as `char`, `int`, etc.,
  - strings (i.e., `char *`),
  - one-dimensional arrays for directly mappable simple types.
- C++ class types or references to C++ objects are mapped to their Java representative class.

- C++ types for which no direct mapping exists can be further divided into two categories

  - *Transparently mappable* data types which could be represented by a similar data type in Java. They are mapped to a Java class containing a *public* member variable of the similar Java type. Examples are `unsigned long`, `enum`, or `signed int`.
  - *Opaquely mappable* data types without any corresponding type in Java. They are mapped to Java classes containing a *private* member of type `long` which is a reference to the address of the C++ data (similar to the handle variable of each Java representative). Example are unions, pointers, etc.

  The names of the container classes are derived from the C++ type name. Pointers and array declarators are represented as suffixes in the class name. The container class concept also guarantees type safety as even opaquely mappable types can be distinguished from each other by the Java compiler.

Table 2 shows the type conversion rules used for the generation of the Java layer and the C++-wrapper/C-stub layer.

6 The C++ Wrapper Layer

The intermediate representation between the Java and the C++ domain which does not support the concept of object-orientation, must be transformed into valid C++ constructs. This transformation is carried out in the C++ wrapper layer which contains only static C++ functions. These functions are wrapper routines dispatching the native Java calls to the entities of the C++ classes to be interfaced. For each public member of a C++ class there is one wrapper routine.

6.1 Constructor wrappers

A constructor

```
A (void);
```

of a Java class A is associated with the following C++ wrapper routine

```
void A_InitA(HA *that) {
  unhand(that)->handle = (int64_t)new A();
  Dictionary::EnterObjectPair(
    (int64_t) that,
    unhand(that)->handle);
}
```

Though not belonging to a C++ class, the wrapper routines are always prefixed with the name of the C++ class they are related to.

The address of both the Java representative object (that) and the newly created C++ instance (handle) are stored in a dictionary. Each time an object reference is passed across

C++ type	Java type	C++ wrapper type
Simple types		
char	char	unicode char
unsigned char	byte	char
short	short	short
int	int	long
long	long	int64_t
float	float	float
double	double	double
void	void	void
Transparent types		
unsigned short	class USHORT	struct HUSHORT *
signed short	class SSHORT	struct HSSHORT *
unsigned int	class UINT	struct HUINT *
signed int	class SINT	struct HSINT *
unsigned long	class ULONG	struct HULONG *
signed long	class SLONG	struct HSLONG *
signed char	class SCHAR	struct HSCHAR *
enum_n	class ENUM_n	struct HENUM_0005fn[a] *
Composite types		
struct	<Java class>	H<class name>
class	<Java class>	H<class name>
union n	class UNION_n	struct HUNION_n
Pointers		
char *	String	HString
unsigned char *	byte[ ]	HArrayOfByte *
all other pointers	<opaque Java class>	H<opaque Java class>
References		
all references	<opaque Java class>	H<opaque Java class>
Arrays		
char[ ]	char[ ]	HArrayOfChar *
unsigned char[ ]	byte[ ]	HArrayOfByte *
short[ ]	short[ ]	HArrayOfShort *
int[ ]	int[ ]	HArrayOfInt *
long[ ]	long[ ]	HArrayOfLong *
float[ ]	float[ ]	HArrayOfFloat *
double[ ]	double[ ]	HArrayOfDouble *
all other arrays	long	int64_t
Multi-dimensional arrays		
<C++ simple type>[ ][ ]...	<Java simple type>[ ][ ]...	HArrayOfArray *
all other multi-dimensional arrays	<opaque Java class>	H<opaque Java class>
all other types	<opaque Java class>	H<opaque Java class>

Table 2: Type conversion scheme.

[a] In the intermediate representation the underscore symbol is used as a delimiter for package prefixes. Therefore, underscores occuring in a Java identifier are escaped by `0005f` as required by the Java native code interface.

the address domain boundary, it is intercepted and replaced by its corresponding reference in the other domain. The dictionary mechanism guarantees that there is always exactly one Java object per interfaced C++ object. The dictionary class is provided by a run-time library which is part of the shared library containing the bridging code.

6.2 Destructor wrapper

The destructor wrapper function is called whenever the Java garbage collector decides to remove the Java representative object. Each Java representative object contains a native finalize method associated with a destructor wrapper that uses the C++ `delete` operator to remove the C++ instance. Before the object is deleted, its entry in the dictionary is removed.

```
void A_finalize(HA *that) {
  Dictionary::RemoveEntry((int64_t)that);
  delete (A *)(unhand(that)->handle);
}
```

6.3 Method wrappers

The C++ wrapper routine for general methods differ from those for constructor methods in that they have to handle communication of the method's return values. For return values the type mapping mechanism as described in section 5 is applied. In the case of a return type being a class type (or a class reference), the dictionary mechanism is used to determine whether the return C++ object already has a Java representative object (i.e., the C++ object is already known on the Java side) or not. In the latter case it must be created using its special constructor and entered in the dictionary.

The following code fragment is a wrapper function for a C++ method with the signature `B& M(A&);`

```
struct HB *A_M(HA *that, struct HA *arg1) {
  HB *rc;
  // invocation of M, the result is stored in hnd
  int64_t hnd = (int64_t)&(((A *)(unhand(that)->handle))
                ->M(*(class A *)(unhand(arg1)->handle)));
  if (!(rc = (HB *)Dictionary::GetJavaObject(hnd))) {
    // Since the C++ object hnd is not mapped yet,
    // its Java representative class must be created...
    rc = (HB *)execute_java_constructor(
                PRIVILEGED_EE,"B",0,
                "(Libm/cplusplus/_NullConstructor_;)",
                (void *)0);
    if (rc) {   // ...and entered in the dictionary.
      unhand(rc)->handle = hnd;
      Dictionary::EnterObjectPair((int64_t) rc,
                                  unhand(rc)->handle);
    }           // ... else fall trough
```

```
  } else {      // hnd is an already known C++ object
    unhand(rc)->handle = hnd;
  }
  return rc;    // for rc == NULL an exception is thrown.
}
```

The wrapper code for Java access methods of C++ member variables is similar to wrapper code generated for methods.

6.4 Operator wrappers

Wrapper functions for operators are similar to those for methods except that the C++ operator is applied in its original notation to the arguments of the wrapper function. As the first argument of member operator definitions is always the containing class instance itself, the wrapper functions also use the C++ instance as the left-most operand. This restriction does not hold for non-member operators which are the static form of member operator definitions.

The following wrapper function shows the non-member form of an operator `+` adding the integer members of a class type A and returning the integer result.

```
long Statics_operatorPLUS(HStatics *that,
                          struct HA *arg0,
                          struct HA *arg1) {
  return (long )(*(class A *)(unhand(arg0)->handle))
                +*(class A *)(unhand(arg1)->handle);
}
```

7 Implementation Issues

The mapping approach described so far has been implemented as a tool which reads the interface description of any given C++ library (i.e., the include files), parses it, and generates the necessary bridging code. As the class definition files are parsed, the tool attempts to resolve both preprocessor directives and type definitions. To do so, it is necessary to process the entire include file hierarchy up to the standard include files. This would result in the generation of some unnecessary bridging code.

To allow the user to select the C++ classes to be interfaced in a reasonable way, the implementation uses Java's class path approach in a slightly different form. Bridging code is generated for those classes whose class definition files was found in one of the directories contained in the class path (either given as a command line option or as an environment variable).

As illustrated in figure 6, for all selected C++ classes three files are generated resembling the three layers of the bridging code:

- The file <class>.java contains the Java representative class via which a Java application can access the C++ library.

- The file <class>Wrapper.cpp contains the C stub routines and the C++ wrapper function forming the lower two layers of the bridging architecture.
- The file <class>Wrapper.hpp defines the intermediate format for instances of <class> when passed across the language border.

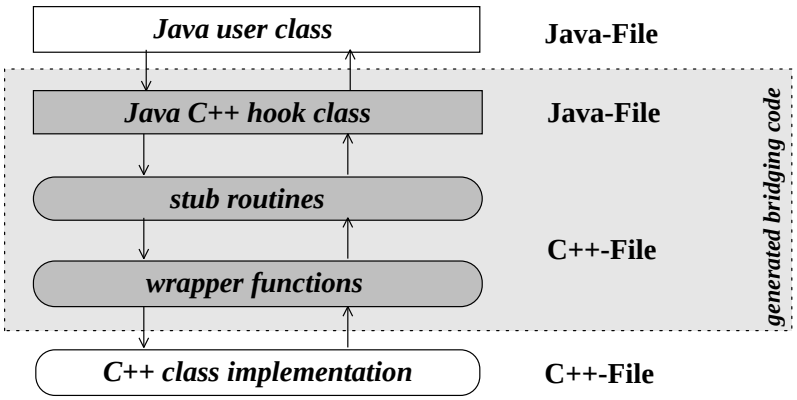

Figure 6: The file structure of the generated bridging code.

The implementation also provides a makefile builder to produce the shared library which provides the native code called by the Java Virtual Machine. Running the generated makefile invokes the code generator, compiles both the Java layer files and stub/wrapper layer files and produces a shared library as depicted in figure 7.

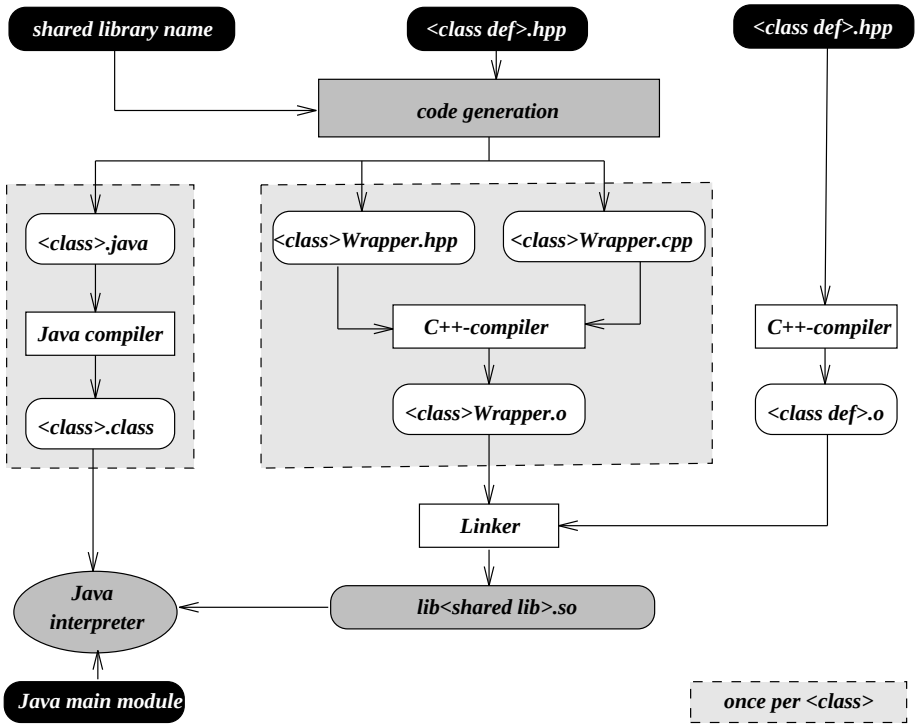

Figure 7: Generating and using a shared library providing Java-accessible C++ code.

The C++ classes to be interfaced can be given either as source code or in form of a library. When compiled, they can become part of the generated shared library or remain

an autonomous module. Figure 7 also shows the generation of system-dependent import and export files which define the entry and exit points of the produces shared library. The method has been implemented for AIX, OS/2[2], Windows NT and Windows95[3].

8 Conclusions

In this paper a general method for access to C++ from Java has been presented based on a domain model for language interoperability. The model has lead to a layered structure of the bridging code consisting of a Java layer, a C stub layer, and a C++ wrapper layer. The interfacing method supports C++ features such as polymorphism and inheritance which are modeled on the Java side by providing Java representative classes that issue calls to the C++ implementation. Public member definitions inside each C++ class such as methods, member variables, operator definitions, constructors, and the destructor are mapped onto access methods in the corresponding Java representative class. For each C++ data type a mapping mechanism is applied which provides a unique Java representation which prevents Java type clashes resulting from the mapping process. The bridging method caters for static C++ member definitions and C++ definitions outside of classes allowing thereby to access to ordinary C code in an automated way.

The interfacing method has been implemented in form of a tool called *CrossWalk for Java* which generates layered bridging code according to the domain model. For each C++ class, a Java file containing the Java representative class, and a C++ implementation and definition file containing wrapper code, is generated. The tool also provides a makefile builder which generates makefiles for each supported platform simplifying the generation of the shared library which contains the bridging code.

The tool has been tested with a variety of existing C++ libraries; however the implementation has still some limitations. Valid Java code would be generated currently for multi-dimensional arrays used as parameter types in C++. However, due to different memory representations which require (generation of) explicit conversion rules, these arrays are not mapped properly in the generated C++ wrapper layer. There is also some research needed to address the remodeling of advanced C++ features such as templates, exceptions and multiple inheritance.

The automatic bridging code generation for inter-operation between Java and C++ solves a common problem for the majority of Java applications. As more and more applications are being written in Java, it relieves the programmer from re-implementing existing C++ implementations of basic tasks in Java. Therefore, the migration process towards Java is simplified and the average development time for Java applications is reduced.

The method can also be deployed for providing Java interfaces for working C++ server implementations allowing the development of Java clients without having to modify the server which, thus, can still collaborate with existing C++ clients. This is especially useful in environments where language-independent communication platforms such as CORBA are not yet supported, since clients and server interface objects can communicate via RMI, the Java Remote Method Invocation interface [Jav96].

We note that Ilog and SunSoft have announced a joint project called Twin-

[2]AIX and OS/2 are registered trademarks of International Business Machines Corporation.

[3]Windows NT and Windows95 are trademarks of Microsoft Corporation.

Peaks [ILO96], which is supposed to deliver a tool also related to Java and C++ interoperation. From the currently available information it appears that the basic concepts of TwinPeaks and CrossWalk are comparable.

9 Acknowledgments

The author would like to thank Thomas Eirich for valuable ideas concerning the implementation and Günther Karjoth for long testing sessions which helped to mature the project.

References

[GHJ+95] Gamma E., Helm P., Johnson R. and Vilssides J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading MA, 1995.

[ILO96] ILOG, Inc.: TWINPEAKS – Bringing Together C++ and Java, A White Paper. ILOG, Inc., 1996.
`http://www.ilog.com/Articles/TwinPeaks_WP.html`

[Jav95] JavaSoft, Inc.: The Java Language Specification. JavaSoft Inc., 1995.
`http://java.sun.com/doc/language_specification.html`

[Jav96] JavaSoft, Inc.: Remote Method Invocation Specification. JavaSoft Inc., 1996.
`http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-spec.ps`