

Non-intrusive object introspection in C++[‡]



Tyng-Ruey Chuang^{*,†}, Y. S. Kuo and Chien-Min Wang

Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan

SUMMARY

We describe the design and implementation of system architecture to support object introspection in C++. In this system, information is collected by parsing class declarations, and is used to build a supporting environment for object introspection. Our approach is non-intrusive because it requires no changes in the original class declarations and libraries; hence, binary compatibility between objects before and after the addition of introspective capability is ensured. This is critical if one wants to integrate third-party class libraries, which are often supplied as black boxes and allow no modification, into highly dynamic applications. We present two applications: the first is automatic I/O support for C++ objects, and the other is interactive exercise of dynamically loaded C++ class libraries. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: C++; object introspection; software reuse; object-oriented software development

1. MOTIVATION

Many object-oriented programming languages, such as CLOS [1], Java [2–4], Objective C [5], and Smalltalk [6], provide introspective language features that allow the state of an object to be observed and altered by means of a uniform mechanism that is equally applicable to objects of all classes. In these languages, binding between a method and the object to be applied can be delayed until runtime, and binding requires no static type-checking between the object and the method. Often called dynamic binding, this approach makes it easy to construct applications whose classes are dynamically loaded and executed. Several classes of applications need dynamic class loading. For example, an integrated development environment (IDE) for an object-oriented language can be considered to be such an application because it needs to compile, link, and execute class definitions on demand.

*Correspondence to: Tyng-Ruey Chuang, Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan.

†E-mail: trc@iis.sinica.edu.tw

‡A preliminary version of this paper was presented under the title 'Non-intrusive Object Introspection in C++: Architecture and Application' in the *20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

Contract/grant sponsor: National Science Council, Taiwan; contract/grant number: NSC 85-2221-E-001-009, NSC 85-2221-E-001-010, NSC 86-2213-E-001-003 and NSC 86-2213-E-001-005

The C++ programming language does not support object introspection [7]. It does provide Run-Time Type Information (RTTI), a run-time class identification mechanism, and virtual functions, mechanisms for run-time resolution of method implementation for polymorphic objects. However, these mechanisms are limited in their functionality since they do not allow full access to an object. For example, one cannot query an object for applicable methods, nor can one gain full access to the data members of any object.

These problems are usually solved by using a meta object framework, such as the System Object Model (SOM) of IBM [8], the Common Object Model (COM) of Microsoft [9], or another similar framework. A framework of this kind requires that introspective objects belong to classes that are either derived from a predefined 'Object' class or are themselves instances of certain meta classes. This creates difficulties in integrating existing class libraries that have been developed without using the framework. The situation becomes worse if the class libraries are provided by third-party vendors and are supplied with no source code. Another disadvantage of the above-mentioned framework is that objects with introspective capability are not compatible with their non-introspective counterparts: the memory layout of an object must be changed to accommodate introspection functionality. In addition, the introspective object may respond differently to existing methods.

We show in this paper how object introspection can be introduced into an existing C++ class library without intruding on its class definitions, and without altering the class hierarchy of the library. An object functions in the same way whether it is capable of introspection or not. In addition, one can apply methods or access states to objects in an introspective manner that bypasses the static type-checking rule of C++.

In Section 2, we provide background and discuss previous work related to introducing object introspection/reflection into C++. We then outline the system architecture of our non-intrusive scheme for C++ object introspection in Section 3. Section 4 discusses several important implementation issues. Section 5 describes two applications that have been developed using our introspective C++ environment. We then discuss the limitations of our approach in Section 6 and draw conclusions in Section 7.

2. BACKGROUND AND RELATED WORK

Reflection is defined as the integral ability of a program to observe or change its own code as well as aspects of its programming language (syntax, semantics, or implementations) at run-time [10]. A programming language is said to be reflective if it provides its programs with the capability of reflection. An important concept in reflective programming languages is *reification*, the process by which aspects of an executing program are brought up using a representation that is expressed in the given language and made available to the program. Furthermore, reification data are causally connected to the related reified aspects such that a modification to one of them affects the other. So far, few programming languages provide the full power of reflection, a very powerful concept whose true implications may not be clear to either language designers or users. However, several languages, such as Lisp and Prolog, do have a limited range of reflective language features and are able to treat programs as data and evaluate reification data at run-time.

Object introspection, in the context of object-oriented programming languages, is the ability to observe and change the state of an object by means of reflection. Introspection is more restricted

than reflection because it adheres to the original syntactical, semantical, and implementational aspects of the source language. It only provides a window to the object states of the current execution of a program and allows changes to be made by means of a uniform gateway to existing legitimate interfaces. For example, using object introspection, one can query, and execute if it exists, an object for a particular method. However, object introspection does not include the ability to add new methods, or modify existing ones, for a class. Therefore, object introspection maintains the semantic integrity of a programming language but opens up its programs to more flexible access. Object introspection allows one to construct applications that are highly dynamic and provides avenues for integration of diverse applications. Open implementations of class libraries [11,12], for instance, will be most natural when object introspection is used.

Some object packaging frameworks, most notably SOM [8,13,14] and COM [9], add object introspection and reflection to the C++ programming language. However each application using such a framework must follow a prescribed class hierarchy. For example, SOM requires that all classes with SOM capability be derived from the SOMObject class, and COM requires that all components with COM capability be equipped with the IUnknown interface. Hence, object introspection cannot be used in applications or class libraries that are not developed using these frameworks. We aim to show that object introspection can be introduced into existing C++ classes without requiring that the classes be derived from, or augmented with, extra declarations.

There are several proposals and projects related to meta object protocols for C++ in the literature (see, for example, [15–17]). They aim to bring full power of reflection to C++. They require customized implementations of the C++ compiler and run-time system since they either add language extensions or change the language semantics. We, on the contrary, seek to develop a framework of object introspection that adheres to the semantics and implementation of C++ (as described in, e.g., [7,18]), and can be used with ordinary C++ compilers.

3. SYSTEM ARCHITECTURE

Providing object introspection for C++ is difficult because C++ objects carry little type information at run-time. The only exceptions are the virtual function mechanism for polymorphic objects and the RTTI facility. Both are rather limited in functionality. Providing a *non-intrusive* object introspection facility for C++ is even more challenging because one is not allowed to augment class declarations such that type information can be automatically attached to each instance to aid introspective operations. Our approach is to define, for each class, a separate meta object that completely captures information about the class for introspection purposes. Introspective operations on instances of the class are then conducted via the corresponding meta object, which has all the necessary information at run-time.

Before discussing further the architecture of such an introspective system, we will examine a typical introspective operation and compare it to the ordinary C++ method invocation. Let `BSTree` be a class whose instances are binary search trees, in which each non-null tree node stores a character string. Suppose class `BSTree` provides a constructor, `new`, for building an empty tree and a public method, `insert`, for adding a node to the tree. Then, the following C++ code segment builds a new tree, `p`, and inserts a node with the character string 'Sinica' into `p`:

```
BSTree *p = new BSTree;  
p->insert("Sinica");
```

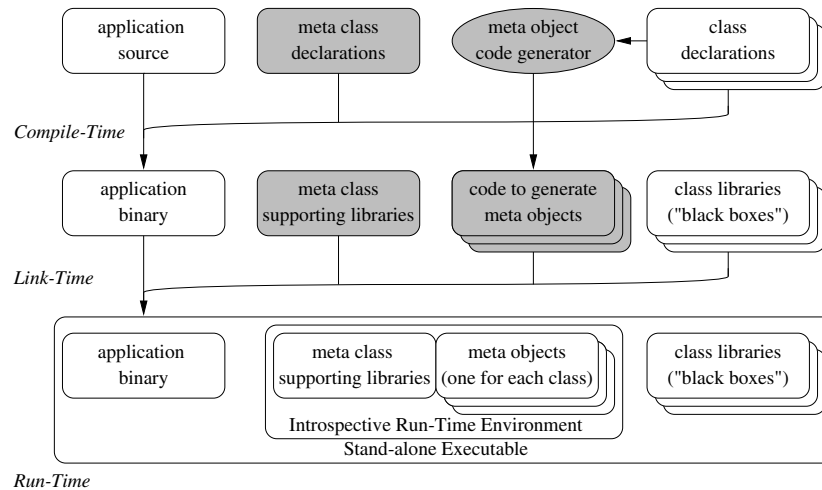


Figure 1. The tightly-coupled model of introspective applications.

Note that, at compile-time, object p is known to be an instance of class `BSTree`, and the `insert` method can be applied to object p .

For introspective operations, however, the binding between p (the object) and `BSTree` (the class) may not be available at compile-time. It may even be the case that the class declaration for `BSTree` is not even available at compile-time. Furthermore, the name of the class (as a character string, say "`BSTree`") may only be known at run-time. In an introspective environment, the same effect of the previous program segment can be achieved by the following:

```
void *p;
Klass bstree = getClass("BSTree");
p = bstree.new();
```

```
Method insert = getMethod(bstree, "insert");
void* argv[] = { "Sinica", 0 };
bstree.invoke(p, insert, argv);
```

Note that the static type of p is now `(void *)`. The fact that it is an instance of class "`BSTree`" is revealed only at run-time. Furthermore, the method `insert` is invoked on p via the meta object `bstree`, which contains all necessary information about class "`BSTree`" for run-time type-checking.

It is now clear that a non-intrusive introspective environment for C++ has two parts. One is the meta object mechanism, which includes declarations of meta classes (such as `Klass` and `Method` described above) and the associated supporting libraries (implementations of `getClass`, `invoke`, and so on). The other part is the mechanism for generating meta objects for classes in need of introspective

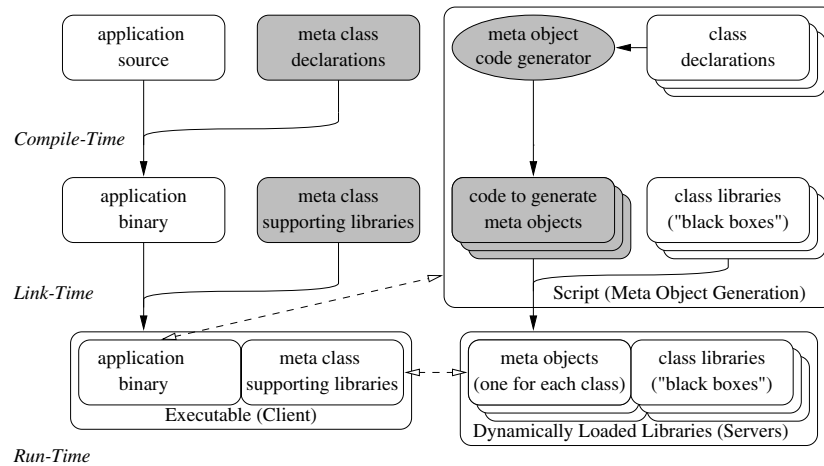


Figure 2. The loosely-coupled model of introspective applications (dashed lines with arrows indicate inter-program interactions at run-time).

operations. The first mechanism is class-neutral and is available at application development time. If all introspective classes are known at application development time, then the code for constructing meta objects can be prepared at compile-time, although meta objects themselves may not materialize until run-time. Generation of the code needed to produce meta objects can be done either manually or automatically. This is shown in Figure 1, where each stand-alone executable includes a self-contained introspective run-time environment. We call introspective applications of this kind tightly-coupled ones. Here, object introspection complements the normal C++ data access and method invocation mechanism.

It may be the case that the classes in need of introspective access are known only at run-time, e.g. in applications that dynamically load class libraries for their functionalities. If so, code generation for the corresponding meta objects can only occur at run-time, and the generation process must be automatic. We call introspective applications of this kind loosely-coupled ones. Here, the binding between an object and its class is dynamic, and introspection is the normal way of interacting with objects. Figure 2 illustrates the loosely-coupled model, where the code for the generation of meta objects is packaged as a script program and invoked by the application on demand (as indicated by the dashed lines with arrows).

In both the tightly-coupled and loosely-coupled models, the mechanism for generating meta objects will need access to the original class declarations but must not modify them. Also, in both cases, applications are developed using the *original* class declarations, with the aid of meta class declarations, which are class-neutral and applicable to all programs. The meta object code generator can be built by using a parser-based analyzer that extracts all necessary type information from class declarations.

4. IMPLEMENTATION ISSUES

As we have shown above, our design of a non-intrusive introspective environment consists of two parts. One is the meta object mechanism, which includes declarations of meta classes and their implementations. The other part is the mechanism for automatic generation of meta objects for classes in need of introspective operations. In the following, we will discuss several implementation issues related to these two parts and the solutions we adopted.

4.1. Meta class interface and library

An application interacts with the introspective environment by means of methods defined in the meta classes. However, the interfaces for interaction often carry less type information than is desirable. As an example, in Section 3, we showed how it is possible to get class information for `BSTree` by passing a *character string* "BSTree" to the introspective environment. A character string says nothing about the class it is associated with (except for its name). To access the method `insert` from the meta object for class `BSTree`, we again use a character string, "insert", as an argument. These interfaces to meta classes are typeless because they must serve requests to all kinds of user-defined classes whose properties are not known to the meta classes in advance. With this in mind, we will now address several issues related to their implementations.

4.1.1. Meta classes for class and method

The meta class for class (`Klass`, as described above in Section 3) must store the name of the class, pointers to meta objects of its base classes, methods for getting names and memory offsets of all its data members, methods for getting names and implementations of all its member functions, and several instance conversion routines between this class and its superclasses and subclasses. Several member functions of the class `Klass` are virtual because they require different implementations for different classes. For example, `getMethod` is a virtual function. Each meta object corresponding to an introspective class will be an instance of a class derived from `Klass`. The derived class then provides the implementation of `getMethod`. The `invoke` method in the class `Method` is virtual as well, and each derived class of `Method` defines its own implementation. Making these methods virtual helps reduce the amount of code that needs to be generated for the meta objects.

4.1.2. Polymorphic object pointers

To correctly access an object, the introspective environment must have the dynamic type information of the object. Note that, however, all object pointers are treated as pointers of type `(void *)` when interfacing with meta objects of their classes. Hence, a function is needed for each introspective class to get the dynamic class names for objects of its class. This function is stored in the meta object of the introspective class. Suppose we have a class `B`; then, the meta object for class `B` will contain a function `dynamicType (pObj)` defined as

```
virtual const char const *dynamicType (void const *self) {
    return typeid(*(B *) self).name(); }
```

that returns the dynamic type name (a character string) of its instance. The function `typeid` above is provided by the C++ RTTI facility.

4.1.3. Base and derived classes

The content of an object consists of its data members and those of its bases as well. In order to access the base's data members, the introspective environment has to perform a 'up cast' operation that adjusts the object pointer. The up cast function is stored in the meta object of the class. For example, the following function will cast an instance of a derived class D of B to an instance of B:

```
static void *fromDtoB(void const *self)
{ return (B *) (D *) self; }
```

If a class has several bases (i.e. multiple inheritance), then all the up cast functions will be stored in the meta object of B. The cast functions will then be in a table and indexed by means of the names of the superclasses.

Similarly, the following 'down cast' routine is stored in the meta object for B as well:

```
static void *fromBtoD(void const *self)
{ return dynamic_cast<D*>((B*) self); }
```

It is used to transform a polymorphic object pointer with known static type (B *) into its dynamic type (D *). The adjustment occurs, e.g., after the dynamic type name "D" is resolved by a call to `dynamicType`. All accesses to data members pointed to by a polymorphic object pointer start with this adjusted pointer.

4.1.4. Other details

C++ provides abstract and virtual base classes. Objects of an abstract class cannot be constructed directly, and base objects of virtual classes share their storage. Information regarding whether a class is abstract, virtual, or neither is recorded in its meta object. This means that the introspective environment has to first check this information when performing data access or object construction.

4.2. Meta object generation and management

For each class in need of introspective operations, the run-time environment needs the meta object of that class. The meta object can be constructed manually or produced automatically from the program text where the class is declared. We use a parser-based program synthesizer that produces program code from user class declarations such that, when the code is executed at run-time with the application code, the necessary meta objects will be generated. Note that, for the tightly-coupled applications shown in Figure 1, code generation and meta object generation are performed separately and at different times. The programming language used for code generation may not even be C++. If it uses C++, it may use a compiler different from the one used to compile the application. Hence, one must take care in generating code that is designed, in turn, to generate meta objects correctly; careful attention should especially be paid to the memory layout of the introspective objects.

4.2.1. Memory layout

The memory layout of an object is calculated by employing a method that relies on a C++ compiler's ability to treat a suitable aligned absolute address as the starting address of any C++ object [19]. For example, the following code computes the offset (in units of `char`) of the data member `a` within an object of class `A`:

```
offset_of_a_in_A = (char *) &(((A *) 64)->a) - (char *) ((A *) 64)
```

where `64` can be any well-aligned absolute address. Note that, in order to calculate offsets of private members, the generator must disable any access control mechanism used by the class. This can be done by altering the original class declaration in several ways. One can delete all the `private` specifiers in the class declaration, hence exposing all data members to the outside world. A better way is to insert a friend function into the class declaration and gain access to all data members without compromising access control of the class too much [20]. Our generator inserts as a friend function an initialization routine into the introspective class in order to calculate data member offsets and produce a meta object for the class. The augmented class declaration is only used for the compilation of friend functions and is discarded afterwards. We discuss in the following additional implementation issues involved in augmenting class declarations.

4.2.2. Nested class declarations

If the class declarations are nested, the initialization routine for the enclosing class, although declared as a friend function, will not be able to access data members of the enclosed classes. The enclosed classes are beyond the scope of the friend function. For these cases, we have the generator insert initialization routines into the enclosed classes as well and have them declared as friend functions in both the enclosed and enclosing classes.

4.2.3. Default constructors

We need a default constructor in order to define the `new` method of the meta class `KLASS` so that blank objects of the right kind can be set up during user program execution. If there is no default constructor in the user's class declaration, we must generate code for it. The generated code for the default constructor has an empty body and does nothing except allocate the necessary memory space and implicitly set up the hidden pointers [21]. If the user's class declaration includes constructors that need additional parameters and initialization procedures, then corresponding friend functions are defined, and made available via the meta object, to facilitate access to and modification of the blank object.

4.2.4. Class templates

Since the generator analyzes only class declarations, not application programs, it does not know how a class template will be instantiated in an application program. We require the user to give hints on how a class template will be instantiated. As an example, as discussed in Section 3, if `BSTree` is a class template (with the type of the tree node used as a parameter), then one must explicitly specify

```
class Name {
private:
    char *name;
public:
    ... constructor and method declarations ...
};

class BSTP {
private:
    char *bstpName;
public:
    ... constructor and method declarations here ...
};

class BSTree: public BSTP {
public: // below is an example of nested class
    struct node {
        int id;
        Name nam;
        struct node *l, *r;
        ... constructors and methods here ...
    };
private:
    struct node *head, *z;
public:
    ... constructor and method declarations here ...
};
```

Figure 3. Code fragments of sample class declarations (see also Figures 4 and 5).

that `BSTree` will be instantiated with type `int`, in order for the generator to produce the meta object for class `BSTree<int>`.

4.2.5. Object compatibility

In the above, we have shown how to augment a class declaration in order for the meta object generators to work. For example, we insert friend functions into class declarations as well as add non-virtual member functions for other purposes. However, we never add data members or virtual functions to the class declarations. This ensures that objects produced by the original class declaration and the augmented one will always have the same memory layout under the usual C++ object model [18]. Furthermore, the augmented class declaration is used only by the generated code and is inaccessible otherwise to the application developers. Developers continue to use the original class declarations and are not aware of the augmented versions.

4.3. Sample code for meta object generation and initialization

To better illustrate the details of meta object generation and initialization, we show code fragments in Figure 3 used as sample class declarations. The classes are used to construct binary search trees with

```

static void *createBSTree(void) { return new BSTree; }
static const char const *dynamicTypeBSTree(void const *self) {
    return typeid(*(BSTree *) self).name(); }
static void *fromBSTreeToBSTP(void const *self) { return (BSTP *) (BSTree *) self; }
static void *fromBSTPToBSTree(void const *self) {
    return dynamic_cast<BSTree *>((BSTP*) self); }

typedef void *(*CastFuncPtr)(void const *); // aux. type def. for "casting".

// The meta object initialization function for BSTree
static int initBSTree(void) {
    const char const *bases[] = { typeid(BSTP).name(), NULL };
    const unsigned short virtualBaseFlag[] = { 0, NULL };
    const char const *members[] = { "head", "z", NULL };
    const char *members_type[] = { typeid(BSTree::node *).name(),
        typeid(BSTree::node *).name(), NULL };
    const size_t offsets[] = {
        (char *) &((BSTree *) 64)->head - (char *) ((BSTree *) 64),
        (char *) &((BSTree *) 64)->z - (char *) ((BSTree *) 64),
        NULL };
    const unsigned int members_count[] = {1, 1, NULL };
    const size_t members_size[] = {
        sizeof(BSTree::node *), sizeof(BSTree::node *), NULL };
    CastFuncPtr ups[] = { fromBSTreeToBSTP, NULL };
    CastFuncPtr downs[] = { fromBSTPToBSTree, NULL };
    ... more function pointers to access BSTree methods ...

    // Add the meta object for BSTree into the meta object dictionary
    addClassDictionary(new Klass(
        typeid(BSTree).name(), // Name of this class
        bases, // Names of the base classes
        virtualBaseFlag, // True if the base is virtual;
        // one flag per base in this array
        members_type, offsets, members, // Types/offsets/names of data members
        members_count, // 1 if the data member is not an array,
        // otherwise the array element size
        members_size, // Sizes of the data members (per element
        // if the data member is an array)
        ups, downs // Pointers to "up/down cast" functions
        dynamicTypeBSTree, // Function that returns class name
        createBSTree, // Function to "new" an object of this class
        ... more function pointers to access BSTree methods ...
    ));
    return 0;
}

// The meta object is added to the meta object dictionary at link time.
static int thisVariableIsNeverUsedBSTree = initBSTree();

```

Figure 4. Code fragments for meta object generation and initialization (cf. Figure 3).

string node values. The classes will also be used later on in Section 5 to further illustrate the need for introspective objects in C++ applications.

For each class declaration *C* shown in Figure 3, including the structure node which is nested inside class *BSTree*, a static function *initC* is produced by the meta object code generator so that the meta object for *C* can be generated at run-time. Also included in the generated code is the definition of a static ‘don’t care’ variable that is initialized by a call to function *initC*:

```
static int thisVariableIsNeverUsedC = initC();
```

If the generated code is linked with a user program, then function *initC* will be automatically invoked at the start of program execution. Only then is the meta object for class *C* generated.

We show in Figure 4 the code fragment that is generated for class *initBSTree*. The code for *initBSTree* is straightforward: after computing all the necessary information to be stored in the meta object, function *initBSTree* creates the meta object by invoking a constructor of class *Klass*—the class of all meta objects—and inserts the meta object into a global meta object dictionary with which user programs can later query to retrieve the desired meta object:

```
addKlassDictionary(new Klass( ... ));
```

Once the meta object is retrieved, introspective operations become available via methods of the meta object.

5. APPLICATIONS

We will describe two applications based on non-intrusive introspective C++ environments. The first is a system called *ObjectStream* that provides automatic I/O support for complex C++ objects. The other is a class exerciser called *RunClass* that allows interactive execution of dynamically loaded C++ class libraries. *ObjectStream* is a system for building applications that are tightly-coupled with their introspective classes, while *RunClass* itself is a loosely-coupled introspective application. Work on *ObjectStream* and *RunClass* has been reported elsewhere although not from the viewpoint of object introspection [22,23].

5.1. ObjectStream

The C++ standard I/O stream library, by overloading the << and >> operators, provides a convenient and type-safe interface for I/O. However, it is only applicable to fundamental types (such as *char*, *int*, etc.). If one wants to I/O objects of user-defined classes, one must either extend the stream I/O library by overloading the << and >> operators or define I/O operations as member functions of those classes. Most C++ development environments (such as the Microsoft Foundation Classes (MFCs)) in use today take the latter approach. They treat the I/O facility as part of a pre-defined application framework. Programmers are required to define I/O operations for user-defined classes by following some prescribed procedures—such as deriving user-defined classes from a ‘persistent’ base class and implementing certain virtual functions for the purpose of object I/O. This practice requires considerable learning time. Furthermore, programmers have to write code to construct/traverse an object’s internal structure in order to perform I/O correctly. When complex objects are involved, this task can be tedious and error-prone. By requiring that all user-defined classes be derived from a persistent base class,

pre-built class libraries are also prevented from performing object I/O. Pre-built class libraries, such as those provided by library vendors but supplied with no source code, often have a pre-defined class hierarchy and cannot be re-derived from the persistent base class.

In `ObjectStream`, we provide a generic I/O library that interacts with the introspective run-time environment to automatically traverse an object's internal structure for I/O purposes. The meta class declarations include the follow template I/O operators:

```
template <class T>
Uostream &operator<<(Uostream &os, T &obj) {
    Uwrite(os, (void*) &obj, typeid(T).name());
    return os; }

template <class T>
Uistream &operator>>(Uistream &is, T *&obj) {
    Uread(is, (void *) obj, typeid(T).name());
    return is; }
```

where `Uistream` and `Uostream` are the class names for the universal streams that are capable of inputting/outputting objects of all classes. The programmers can then use `>>` and `<<` to I/O objects of user-defined classes.

Note that the object reference is passed as a void pointer, and that the class information is passed as a string containing the static class name of the object. `Uwrite` and `Uread` are generic read/write functions that interact with the introspective environment to traverse objects for I/O. The object traversal algorithm used by the library is a depth-first search that looks inside an object for its bases and data members. The traversal algorithm is not that different from those used for garbage collection. Since bases and data members are objects as well, the search is recursive in nature. When the search encounters objects of fundamental types, it calls the corresponding primitive routines for I/O. A major issue in the search is to avoid duplication of I/O for objects that have already been visited. For this purpose, we maintain a dictionary of object references that maps between an object's internal memory address and its external object ID. The library first checks with the object reference dictionary using the object's memory address as a key to see whether the object has been output or not. If it has, only the object's ID is output. Otherwise, a new ID is assigned to the object and a new entry of (address, ID) pair is entered into the dictionary. Similarly, when the input operation is performed, if the library sees only an object ID, then it is used as a key to look for the object's memory address in the dictionary.

In Figure 5, we show a program fragment to illustrate how user-defined objects can be restored/stored from/to universal streams. The program constructs a binary search tree, outputs the tree to a file using the `<<` operator, and reads the same tree back from the file using the `>>` operator. `ObjectStream` also comes with an 'object stream browser' that can be used to open an object stream and display objects in the stream. Figure 6 shows a screenshot in which the object stream browser is used to view a binary search tree that is stored in an object stream.

5.2. RunClass

Object-oriented development is often characterized by the development and use of a large set of reusable classes. Current application development environments facilitate software reuse by providing

```

#include <ostream.h>    // Declarations of I/O operators and universal streams.
#include "bstree.h"     // Declarations of BSTree (for Binary Search Tree).

main () {
  AsciiOut  ascout;    // Use ASCII format for output.
  ascout.precision(16); // Precision is 16 digits wide.
  Uofstream ofile("tree.dat", ascout); // Create an output stream "tree.dat"
                                     // on file medium with ASCII format.

  BSTree    myTree;    // The place to hold a BSTree object.
  ...       // Insertion of elements into myTree omitted here.
  ofile << myTree;     // Store myTree to ofile.
  ofile.close();      // Close the file stream.

  AsciiIn   ascIn;    // Use ASCII format for input.
  Uifstream ifile("tree.dat", ascIn); // Use file "tree.dat" as the input
                                     // stream. The format is ASCII.

  BSTree    *pTree = 0; // The pointer to hold a restored BSTree object.
  try {
    ifile >> pTree;    // Restore a BSTree object from ifile. After this,
                       // *pTree and myTree contain the same tree.
  } catch (IOError error) { error.printMessage(); } // Catch errors, if any.
  ifile.close();      // Close the input file stream.
}

```

Figure 5. ObjectStream: an illustrating example (see Figure 3 for classes declared in bstree.h).

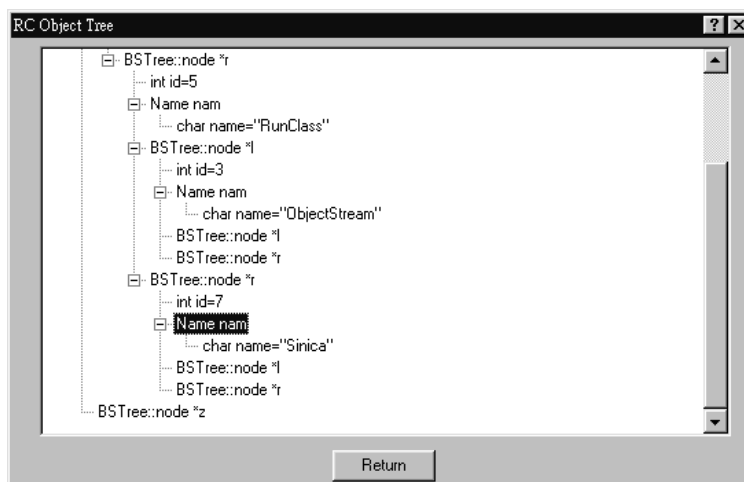


Figure 6. Browsing an object stream.

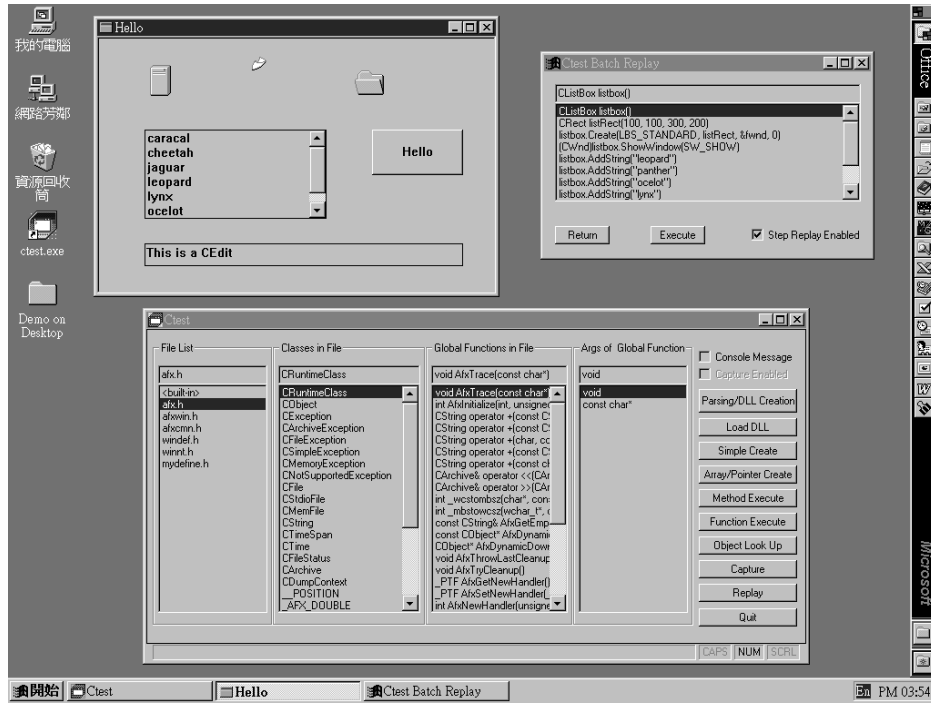


Figure 7. RunClass: exercising the MFCs.

tools to inspect and access class libraries. Using these tools, one can easily locate desired classes and retrieve information of interest. However, invocation of classes (including instantiation of objects from classes and invocation of methods upon objects) is carried out in the form of writing and executing programs. That is, one has to write a short testing program simply in order to test a class' functionality. This is fine if the application programmer really wants to use the class in program development. However, in many situations, one may only want to invoke a class to evaluate its functionality, to see an undocumented feature, or to verify its correctness. For these situations, an interactive, easy-to-use environment for 'exercising' classes appears to be more convenient than the traditional edit-compile-run programming environment.

RunClass is a class exerciser that, when a set of classes is taken as input, allows a user to create objects for these classes, execute methods upon the objects, and examine their contents interactively. RunClass provides an easy-to-use graphical interface, where a class library is presented to a user by means of a list of classes in the library and a list of methods for each class. A user can then select and invoke a desired class or method using a pointing device without memorizing their names. RunClass also manages objects that have been created. Objects are grouped according to their classes for the user's convenience. We have used RunClass in several applications. It has been used as a

demonstration tool, interactively showing the functionality of a class library. It has also been used for testing and maintenance purposes because it allows easy access to classes that are unfamiliar to programmers.

Implementation of `RunClass` is straightforward in an introspective C++ environment. It needs a graphical user interface to receive commands from users and pass them to the underlying introspective environment. Results from commands are then displayed for users to interpret. It also needs a command manager that can capture a sequence of commands and log their effects. The command sequence can then be replayed later for regression testing on newer versions of the class library.

Figure 7 shows a snapshot of using `RunClass` to exercise the MFCs [24]. On the right-hand side is the replay dialog, which shows some previously captured operations. They appear in the form of C++ statements. On the left-hand side are a window and four MFC 'controls' (animate control, list box, push button, and edit control) inside the window, which are all constructed by replaying the captured operations. By using the dialog at the bottom, users can manipulate the windows and controls, such as by pausing and playing the animate control, adding items to the list box, and so on via interactive method invocations. `RunClass` used as an exerciser of MFCs has proven to be an attractive development tool for novice MFC programmers.

6. LIMITATIONS

There are several limitations in our approach to non-intrusive object introspection and in the applications so far developed. Most are caused by ambiguities in the C++ language. C++ allows pointers to be of type `(void *)` and allows data members to have `union` types. When our code generator parses class declarations with such data members, the information it collects and puts into the meta objects is ambiguous and useless for the meta class supporting libraries. For example, how should `ObjectStream` output a data member of type `(void *)`? Usually the type name is available as a string, say "BSTree", in an introspective environment. Now, we have "`(void *)`", which says almost nothing. Similarly, if a data member is a pointer, then it can serve as the address of an object or as the starting address of a dynamic array. If the data member is a fix-sized array, it will be correctly handled. Note, however, that these restrictions can always be resolved by using user-defined I/O operations, which are supported by `ObjectStream`. At the present time, `ObjectStream` either does not support such data members (default) or require users to provide customized I/O routines, which it hooks up to the supporting library automatically.

The very nature of late-binding applications can cause some problems as well. For example, the late-binding method invocation (described in Section 3)

```
bstree.invoke(p, insert, argv);
```

requires several run-time checks to ensure type-safety. This is because object `p` is not necessarily of the type `bstree`, the method `insert` is not necessarily defined in the class `bstree`, and so on. Standard C++ method invocations, on the other hand, do not require these run-time checks. Introspective applications, then, suffer from this and other meta object overheads required to ensure that introspective operations are safe. From our experience, such overheads are tolerable as they do not affect overall application performance in a major way. Note that, when our approach is used, all

non-introspective accesses to introspective objects still use the usual C++ object model and are just as efficient.

Our implementation of non-intrusive object introspection in C++ assumes use of the common C++ object model (as described in, e.g., [18]). For example, data members of an object use consecutive memory address in the object's memory space, and non-virtual member functions do not occupy object memory and are resolved at compile-time. There are other object models, such as IBM's SOM [8], that make no such assumption. However, our approach to non-intrusive object introspection—automatic generation of C++ code for the creation of meta objects at run-time—remains applicable to other object models. Based on detailed information about a data model, our meta object code generator can be modified so that the correct meta objects are created at run-time.

7. CONCLUSIONS

We have presented the concept of non-intrusive object introspection as well as its models and implementations in C++. Our approach works with standard compilers and existing class libraries. Several important implementation issues have been discussed, and two substantial applications have been described. It has been shown that non-intrusive object introspection can be added to C++ in a rather straightforward way, and that, once added, it facilitates application reuse and integration.

The introspective framework can be implemented in a compiler- and machine-independent manner. One only needs to ensure that the meta object code generator will produce C++ code that can be compiled with different C++ compilers, and that it can be correctly executed on different platforms. Our meta object code generator, in fact, is a modified G++ front-end of the GNU Compiler Collection (GCC). The modified G++ front-end extracts from user-defined class declarations information necessary for the generation of meta object code. Notice that the user program code, the meta object supporting library source, and the generated meta object code may be compiled using non-G++ compilers. This implies that the modified G++ front-end may need to process non-G++-compliant class declarations supported by non-G++ C++ dialects, including those required by the C++ standard library. This was especially the case in the early stage of C++ language evolution. It is for this reason that we had to make frequent adjustments to the meta object code generator in order for them to fit the C++ compilers used by the source programs while the implementation of the G++ front-end was, at the same time, evolving.

A future direction of introspective/reflective execution of high-level programs concerns open compilers. An open compiler exports a well-defined, semantics rich, external representation of the source program at each stage of the compilation process. An example of external representation is the Register Transfer Language (RTL) intermediate form used in the GCC common back-end. Such external representations will allow the source program to be specialized for other functions with the aid of other programs, e.g. meta object code generators, non-standard debugging tools, performance monitors, etc. Results of specialization could even be used in later stages of the same compilation process.

ACKNOWLEDGEMENTS

This paper is the result of ideas and experience accumulated over the last few years in the Software Methodology Laboratory at the Institute of Information Science, Academia Sinica. `ObjectStream` was mainly implemented by

Chuan-Chieh Jung and Wen-Min Kuan, and RunClass was implemented by Chin-Chuan Hsu and Wei-Hsueh Lai. We thank them for their hard work.

REFERENCES

1. Paepcke A (ed.). *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.
2. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley, 1996.
3. Sun Microsystems. *Java Core Reflection—API and Specification*, January 1997.
4. Sun Microsystems. java.lang.reflect. As a package in *Java 2 Platform API Specification*, 2000.
5. NeXTSTEP. *Object-Oriented Programming and the Objective C Language*. Addison-Wesley, 1993.
6. Goldberg A, Robson D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1993.
7. Stroustrup B. *The C++ Programming Language* (2nd edn). Addison-Wesley, 1993.
8. International Business Machines. *SOMobjects Developer's Toolkit, Programmer's Guide* (2nd edn), vol. 1–2, 1996.
9. Rogerson D. *Inside COM*. Microsoft Press, 1997.
10. Malenfant J, Jacques M, Demers F-N. A tutorial on behavioral reflection and its implementation. *Proceedings of the Reflection '96 Conference*, San Francisco, CA, Xerox Palo Alto Research Center, April 1996, Kiczales G (ed.). 1996; 1–20.
11. Kiczales G. Beyond the black box: Open implementation. *IEEE Software* 1996; **13**(1):8–10.
12. Kiczales G, Lamping J, Lopes CV, Maeda C, Mendhekar A, Murphy G. Open implementation design guidelines. *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997. IEEE Press, 1997; 481–490.
13. Danforth S, Forman IR. Reflection on metaclass programming in SOM. *Conference Proceedings of Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, USA, October 1994. ACM Press, 1994; 440–452.
14. Hamilton J, Klarer R, Mendell M, Thomson B. Using SOM with C++. *C++ Report* 1995; **7**(6):40–45.
15. Chiba S. A metaobject protocol for C++. *Conference Proceedings of Object-Oriented Programming Systems, Languages and Applications*, Austin, TX, October 1995. ACM Press, 1995; 285–299.
16. Gowing B, Cahill V. Meta-object protocols for C++: The Iguana approach. *Proceedings of the Reflection '96 Conference*, San Francisco, CA, Xerox Palo Alto Research Center, April 1996, Kiczales G (ed.). 1996; 137–152.
17. Ishikawa Y, Hori A, Sato M, Matsuda M, Nolte J, Tezuka H, Konaka H, Maeda M, Kubota K. Design and implementation of metalevel architecture in C++—MPC++ approach. *Proceedings of the Reflection '96 Conference*, San Francisco, CA, Xerox Palo Alto Research Center, April 1996, Kiczales G (ed.). 1996; 153–166.
18. Lippman SB. *Inside The C++ Object Model*. Addison-Wesley, 1996.
19. Mecklenburg R, Clark C, Lindstrom G, Yih B. A dossier driven persistent objects facility. *USENIX 6th C++ Technical Conference*, Cambridge, MA, April 1994. USENIX Association, 1994; 265–281.
20. Tichy WF, Heilig J, Paulisch FN. A generative and generic approach to persistence. *C++ Report* 1994; **6**(1):22–33.
21. Biliris A, Dar S, Gehani NH. Making C++ objects persistent: The hidden pointers. *Software—Practice and Experience* 1993; **23**(12):1285–1303.
22. Chuang T-R, Jung C-C, Kuan W-M, Kuo YS. ObjectStream: Generating stream-based object I/O for C++. *Proceedings 24th International Conference on Technology of Object-Oriented Languages and Systems*, Beijing, China, September 1997, Chen J, Li M, Mingins C, Meyer B (eds.). IEEE Computer Society Press, 1997; 70–79.
23. Wang C-M, Kuo YS. Class exerciser: A basic tool for object-oriented development. *Proceedings of the 1995 Asia Pacific Software Engineering Conference*, Brisbane, Australia, December 1995; 108–116.
24. Microsoft. *Microsoft Foundation Classes*, version 4.2.1, 1997.