XVF: C++ Introspection by Extensible Visitation

[Extended Abstract]

Kurt Stephens
ION, Inc.
5183 Pinetree Drive
Delray Beach, FL, 33484, USA
kurtstephens@acm.org

May 22, 2001 (revised May 16, 2003)

ABSTRACT

Object serialization and object inspector user interfaces are concerns that can be orthogonally implemented using introspection via meta-object protocols (MOP). The C++ language lacks a formal meta-object protocol, although some are available as source pre-processors. A full MOP is not necessary for many classes of problems where basic introspection is useful. This paper describes a technique: extensible visitation of objects as a basic introspection primitive.

1. INTRODUCTION

Introspection, the access of information about a program's construction during run-time, is valuable. Meta-object protocols (MOP) [5] allow object-oriented systems to reify their own definition. Some C++ MOP implementations ([2], [3]) use pre-compilation.

For common concerns, such as object serialization and object inspector user interfaces, a complete MOP is useful but not required. Extensible visitation based on the Visitor design pattern [4] is a mechanism, similar to XDR [9], for interfacing different introspections with little impact to an existing C++ framework.

2. VISITATION AS AN INTROSPECTION PRIMITIVE

Generic programming techniques aim to eliminate boilerplate code [6]. Traversal code is often the largest part of many computations.

Standard C++ `ostream` classes are visitors that produce output as a side-effect of visiting typed data. Developers must create `ostream` visitation methods for each new application class. Developers must create similar visitation methods for other visitors. These methods are specialized for different visitors, yet do mostly the same thing; recursively visit all data members while side-effecting the visitor.

For example:

```
class Foo
{
 private:
  double x, y; // data members
  int i3[3];

 public:
  Foo() { ... }
  Foo(double _x, double _y) { ... }

  // Visitation Methods, one for each type of visitor.

 // ostream dump method.
 ostream &print(ostream &os)
 {
   os << "Foo(" << x ", " << y << ", ";
   os << "{ "
      << i3[0] << ", "
      << i3[1] << ", "
      << i3[2]
      << " }";
   return os << ")";
 }

 // XML output method.
 ostream &print_xml(ostream &os)
 {
   os << "<Foo>";
   os << "<x>"; print_xml(x, os); os << "</x>";
   os << "<y>"; print_xml(y, os); os << "</y>";
   os << "<i3>";
   print_xml(i3[0], os);
   print_xml(i3[1], os);
   print_xml(i3[2], os);
   os << "</i3>";
   return os << "</Foo>";
 }

 // inspector generation method.
 inspector &generate_inspector(inspector &is)
 {
   is.begin_frame("Foo", this);
   is.generate_inspector("x", &x);
   is.generate_inspector("y", &y);
   is.begin_frame("i3[]", &i3);
   is.generate_inspector("0", &i3[0]);
   is.generate_inspector("1", &i3[1]);
   is.generate_inspector("2", &i3[2]);
   is.end_frame(&i3);
   is.end_frame(this);
   return is;
 }

 // XDR support method
 int xdr(XDR *xdr)
 {
   return
   xdr_double(xdr, &x) &&
   xdr_double(xdr, &y) &&
   xdr_int(xdr, &i3[0]) &&
   xdr_int(xdr, &i3[1]) &&
   xdr_int(xdr, &i3[2]);
 }
 // ... other visitor types
}
```

In the example above, more visitation methods are needed as the application classes (visitees) interface with more visitors. Adding, removing or renaming data members becomes a maintenance problem, since each visitation method must be updated. Likewise, adding new visitations becomes difficult as the application grows, since each application class must be updated.

If the number of application classes in a system is `N` and the number of visitor classes is `M`, the complexity of the application-visitor interfaces will be `N * M`, if visitor-specific methods are implemented for each application class. If a single generic visitation method is used for each visitor class, the complexity of the application-visitor interfaces will be `N + M`.

Ideally a single visitation method suffices for most types of visitation. Essentially we want to move the complexity from the visited classes to the visitors, since the number of visitor classes will be small compared to the visited classes. Visitor classes tend to interface auxiliary systems and protocols, such as XML and other I/O systems, and change less frequently than the application classes.

3. XVF

XVF (eXtensible Visitation Framework) is a C++ template-based framework for developing visitors and generic visitation methods. In XVF, a generic introspection visitor is messaged with type, naming and layout information about the visited objects. The visitor consumes the introspective messages and controls the recursion during visitation. The visitor's introspection methods are virtual, such that any visitor can be used in any visitation traversal. The set of introspection methods in the visitor closely follows a decomposition of the data types in C++: structures (classes), pointers, arrays and intrinsic types (`char`, `int`, `double`).

3.1 Visitor classes

In XVF, Visitors are sub-classes of `XVFVisitor`. `XVFVisitor` has methods for keeping track of what object is being visited and what members are being visited. Each intrinsic type (e.g. `char`, `int`, `double`) has a visitation method specialized for it in the visitor class.

Visitors are messaged with information about the visitee during visitation traversal.

3.2 Visitee type objects

In XVF, the `XVFType` class provides basic type introspection: type name, type size and allocation and assignment methods.

`XVFType` objects are constructed dynamically by class templates instantiated by template functions. There are five `XVFType` templates: intrinsic types (`char`, `int`, `unsigned long`), abstract classes (non-instantiable classes), concrete classes (instantiable classes), pointers, and arrays.

The `XVFTypeIntrinsic<T>` template creates `XVFType` classes for intrinsic types. The templates `XVFTypeClass<T>` and `XVFTypeAbstract<T>` create type objects for instantiable and abstract classes, respectively. Type objects for intrinsic types are created "by hand" in XVF. Pointer and array type objects are constructed by `XVFTypePtr<T>` and `XVFTypeArray<T>`, respectively. `XVFType` templates are never instantiated by the user of XVF, but are created automatically by the instantiation of `xvf_type(T*)` template functions based on pointers to type `T`.

Each type object implements a virtual `visit(void *data, XVFVisitor *V)` method that is specialized by the type object template to call a template function `xvf_visit((T *) data, V)`;

For example, if `aFoo` is an instance of class `Foo`, the template function call, `xvf_type(aFoo)`, returns the `XVFType` type object for the object instance `aFoo`. Likewise, the static method, `XVFType::type("Foo")`, returns the same `XVFtype` type object for the class named `"Foo"`.

The type objects are used to aid the visitors when visiting each class. For example: a XML input visitor may need to construct a new object based on a XML tag name.

3.3 Visitor/Visitee Interface

Visitee classes are not sub-classes of a special "Visitee" class. They are "glued" to the XVF library by static type polymorphic functions:

`XVFType *xvf_type(T *x)`

returns the `XVFType` object for type `T`.

`T *xvf_alloc(T *dummy)`

allocates an object of type `T`.

```
void xvf_dealloc(T *x)
```

deallocates an object of type `T`.

```
void xvf_visit(T *x, XVFVisitor *V, int opts)
```

recursively visits an object of type `T` with visitor `V`.

Note: XVF does not dispatch using `const T` because then `void` could not be handled in a unified manner and because data are visited by address.

3.4 Visitation Methods

Any type `T` that implements a `xvf_type(T *x)` and `xvf_visit(T *x, XVFVisitor *V, int opts)` (the `opts` argument is explained later) functions can be visited by any `XVFVisitor` object.

XVF supplies macros for constructing a class' visitation method and type object accessor functions.

For example:

```
// Declare xvf_type(Foo *)
XVF_CLASS_BEGIN(Foo);

class Foo {
  double x, y;
  int i3[3];
public:
  Foo() { ... }
  Foo(double x, double y) { ... }

  // xvf_type() support
  XVF_CLASS_METHODS(XVF_);

  // xvf_visit() support
  XVF_CLASS_VISIT_BEGIN_INLINE(Foo)
  {
     XVF_MEMB(x);
     XVF_MEMB(y);
     XVF_MEMB(i3);
  }
  XVF_CLASS_VISIT_END()
};

// Define xvf_visit(Foo *)
XVF_CLASS_END(Foo);

// Define xvf_type(Foo *)
XVF_CLASS_INSTANCE(Foo);
```

3.5 Visitor Messages

The visitor is messaged with information about the visited object during traversal. The visitor can dynamically control recursion for any visitation member. The `XVF_MEMB()` macro above expands into messages to the visitor.

For example: any visitor `V` visiting `FooF` will have the following methods called on it:

```
xvf_visit(&F, V);
```

is equivalent to:

```
V->this_begin(xvf_type(&F), &F, opts);
 if ( V->memb(xvf_type(&F.x), "x", &F.x, opts) ) {
   V->memb_begin(xvf_type(&F.x), "x", &F.x, opts);
    xvf_visit(&F.x, V, opts);
   V->memb_end(xvf_type(&F.x), "x", &F.x, opts);
 }
 if ( V->memb(xvf_type(&F.y), "y", &F.y, opts) ) {
   V->memb_begin(xvf_type(&F.y), "y", &F.y, opts);
    xvf_visit(&F.y, V, opts);
   V->memb_end(xvf_type(&F.y), "y" &F.y, opts);
 }
 if ( V->memb(xvf_type(&F.i3), "i3", &F.y, opts) ) {
   V->memb_begin(xvf_type(&F.i3), "i3", &F.y, opts);
     V->arry_begin(xvf_type(&F.i3),
                   xvf_type(&F.i3[0]),
                   &F.i3, 3, opts);
       V->elem_begin(0, opts);
         xvf_visit(&F.i3[0], V, opts);
       V->elem_end(0, opts);
       V->elem_begin(1, opts);
         xvf_visit(&F.i3[1], V, opts);
       V->elem_end(1, opts);
       V->elem_begin(2, opts);
         xvf_visit(&F.i3[2], V, opts);
       V->elem_end(3, opts);
     V->arry_end(xvf_type(&F.i3),
                 xvf_type(&F.i3[0]),
                 &F.i3, 3, opts);
   V->memb_end(xvf_type(&F.i3), "i3" &F.y, opts);
 }
V->this_end();
```

4. HANDLING CONST-NESS

The `opts` argument to the `visit` functions relays `const`-ness to the visitor without requiring different routines and type objects for `const` and non-`const` visitation.

If `opts == XVF_NON_CONST` in `V->visit(T *data, int opts)`, `V->visit()` has license to modify `*data`.

For example: an inspector user-interface visitor may generate an editable or non-editable user-interface field depending if `opts == XVF_NON_CONST` or `opts == XVF_CONST`.

5. HANDLING VISIT-BY-VALUE

Sometimes the visitation semantics of a class' members should be through a pair of accessor methods or functions, rather than directly on an object data member.

For example:

```
XVF_CLASS_BEGIN(Temperature);
class Temperature {
  private:
    double _f; // Fahrenheit: f = (c * 1.8) + 32
    double _c; // Celsius:    c = (f - 32) / 1.8

  public:
  Temperature() { celsius(0); }

  // Getter, setter.
  double fahrenheit() const { return _f; }
  void fahrenheit(double x) {
    _f = x; _c = (_f - 32) / 1.8;
   }

  // Getter, setter.
  double celsius() const { return _c; }
  void celsius(double x) {
    _c = x; _f = _c * 1.8 + 32;
  }

  // Getter, setter.
  // Note: different setter name style
  double kelvin() const { return _c + 273.15; }
  void set_kelvin(double x) {
    celsius(x - 273.15);
  }

  XVF_CLASS_METHODS(XVF_);
  XVF_CLASS_VISIT_BEGIN_INLINE(Temperature)
  {
     // Visit by accessor code
     XVF_ACC_T("fahrenheit", // name
               double,       // type
               _xvfv_this->fahrenheit(),
                             // getter
               _xvfv_this->fahrenheit(XVF_TEMP)
                     // setter
     );

     // Visit by accessor names using
     //   shorthand for:
     // getter: _xvfv_this->celsius()
     // setter: _xvfv_this->celsius(XVF_TEMP)
     XVF_ACC_T_M("celsius",
               double,
               celsius,
               celsius
     );

     XVF_ACC_T_M("kelvin",
               double,
               kelvin,
               set_kelvin
     );

  }
  XVF_CLASS_VISIT_END()
};
XVF_CLASS_END(Temperature);
XVF_CLASS_INSTANCE(Temperature);
```

If any visitor updates the `"fahrenheit"`, `"celsius"` or `"kelvin"` members of a `Temperature` object, the Fahrenheit, Celsius and Kelvin values will be synchronized through the getter and setter methods in `Temperature`.

6. VISITATION ATTRIBUTES

A visitation method can relay visitor-specific information using visitation attributes.

Imagine a visitor that needs to know how each member is protected in a C++ class:

```
...
class Bar {
private:
  int a; // a is private.
protected:
  float b; // a is protected.
public:
  char *c; // c is public.

  Bar(...)

  XVF_CLASS_VISIT_BEGIN_INLINE(Bar)
  {
    // Save and set attribute value.
    XVF_ATTR_BEGIN("C++:protection", "private");
    XVF_MEMB(a);

    // Set attribute value.
    XVF_ATTR("C++:protection", "protected");
    XVF_MEMB(b);

    XVF_ATTR("C++:protection", "public");
    XVF_MEMB(c);

    // Restore attribute value.
    XVF_ATTR_END("C++:protection");
  }
  XVF_CLASS_VISIT_END();
};
...
```

The specialized visitor will check the attribute named `"C++:protection"` for its value and produce results accordingly.

7. EXAMPLES

Below are sample uses of visitor objects using the same visit methods for class `Foo`. The visitor code is not presented here.

7.1 A Structured Data Dumper

A structured data visitor writes a human-readable representation to an `ostream`.

```
XVF_Dump_Out<ostream> xvf(cout);
Foo p(1, 2);

xvfv << p;
```

7.2 An Archiver

An archiver visitor writes a machine-readable representation to an `ostream`.

Figure 1: TK widget generated by XVF˙TK˙Inspector.

```
XVF_Archive_Out<ostream> xvfv(cout);
Foo p(1, 2);

xvfv << p;
```

7.3 XML I/O

An XML output visitor writes objects as an XML stream.

```
XVF_XML_Out<ostream> xvfv(cout);
Foo p(1, 2);

xvfv << p;
```

An XML input visitor reads an XML stream into a object.

```
XVFV_XML_In<istream> xvfv(cin);
Foo p;

xvfv >> p;
```

7.4 Dynamic GUI generation

An inspector visitor generates TCL/TK code to create an inspector GUI.

```
Tcl_Interp *interp;
XVF_TK_Inspector xvfv(interp);
Foo p(1, 2);

xvfv << p;
```

7.5 Reflective Accessors

These accessor visitors apply getter and setter methods by name.

```
XVF_Getter xvfg;
Foo p(1, 2);
double py;

xvfg(&p, xvf_type(&p), "y", &py); // py = p.y

XVF_Setter xvfs;
xvfs(&p, xvf_type(&p), "x", &py); // p.x = py
```

7.6 XDR Bridge

This visitor provides XDR services.

```
XDR *xdr;
XVF_XDR xvfv(xdr);

xvfv << p;
```

8. RELATED WORK

External Data Representation (XDR) [9] uses generic visitation for encoding, decoding and freeing C objects. XDR visitors cannot be sub-classed directly and do not provide any mechanisms for XDR visitors to determine member names. XDR is used primarily for Remote Procedure Calls (RPC) [7].

OpenC++ [3] provides introspection using pre-compiler techniques. Iguana [2] seems to be based on pre-compiler techniques as well, but no public implementations are offered.

[6] discusses generic traversal combinator generation for generic programming support.

9. FUTURE WORK

Namespaces, enumerations, bit-fields, unions, function pointers and methods are not handled. Class versioning for backward compatibility is not handled. A pre-processor to parse members from class definitions to generate and insert `xvf_visit()` methods would be helpful; OpenC++ or SWIG [1] might be applicable. A C interface for legacy code may also be useful.

10. CONCLUSION

This paper describes a method for basic introspection in C++ using the Visitor design pattern. A prototype implementation of XVF is available at URL: `http://kurtstephens.com/pub/xvf`.

11. REFERENCES

[1] D. Beazley. *Simplified Wrapper and Interface Generator Homepage*, URL: `http://www.swig.org`, 2003.

[2] V. Cahill. *The Iguana Project Homepage*, URL: `http://www.dsg.cs.tcd.ie/~coyote`, 2003.

[3] S. Chiba. *OpenC++ Homepage*, URL: `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`, 2003.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns*, Addison-Wesley, 1995.

[5] G. Kiczales. *The Art of the Metaobject Protocol*, MIT Press, 1991.

[6] R. Lammel and S. Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26-37, ACM, 2003.

[7] R. Srinivasan. *RFC 1831: RPC: Remote Procedure Call Protocol Specification Version 2*, URL: `http://www.ietf.org/rfc/rfc1831.txt`, 1995.

[8] K. Stephens. *XVF Package Homepage*, ION, Inc., URL: `http://kurtstephens.com/pub/xvf`, 2001.

[9] Sun Microsystems, Inc. *RFC 1014: XDR: External Data Representation Standard*, URL: `http://www.ietf.org/rfc/rfc1014.txt`, 1987.