XTL – The Externalization Template Library

José Orlando Pereira

Departamento de Informática

Universidade do Minho

Address: Dep. de Informática, Campus de Gualtar, 4710-057 Braga, PORTUGAL

Phone: +351 253 604 477 / Fax: +351 253 604 471

E-mail: jop@di.uminho.pt

December 1999

Abstract

The Externalization Template Library (XTL) is a set of template classes designed to ease the task of converting C++ data-structures to a language and machine independent external representation, and from this external representation back to C++ data-structures, as required for communication or persistent storage in heterogeneous computer systems.

As externalization requires the traversal of data-structures, most existing solutions require a special compiler to generate traversal procedures from simple specifications, restricting which data-structures can be externalized. Those that do not, are either inefficient or require extra effort by the programmer.

In contrast, the XTL demonstrates how some features of modern C++ can make the development of traversal procedures for externalization convenient without restricting the programmer to a few data-structures. As the XTL is especially designed for optimizing compilers, the resulting code is also very fast.

1 Introduction

The tasks of converting data-structures to a language and machine independent external representation and from this external representation back to the original data-structures are respectively known as *externalization* and *internalization*[1].

The external representation of a data-structure is a sequence of bytes which exhibits an important characteristic: given the appropriate internalization procedure, it is sufficient to rebuild the data-structure regardless of where it was created. As such, standardized external data formats and the associated externalization and internalization procedures are an absolute requirement for communication or persistent storage in heterogeneous computer systems.

Most of the existing toolkits for externalization require a data-description language and its associated compiler to generate fast conversion procedures from simple specifications of data-structures. This is the case of most distributed programming systems which rely on interface description languages.

However, this solution poses several problems, such as tying externalization to a specific function in a particular distributed computing toolkit and making it unusable for any other purpose. In addition, the mapping from the specification to concrete data-structures is usually also very restrictive in which data-structures of the programming language are supported. In the context of object-oriented programming, it is also a problem that the internal structure of objects has to be exposed.

On the other hand, solutions that do not rely on a special compiler, are either inefficient or require the programmer to perform the tedious task of hand-writing and optimizing externalization and internalization procedures for each data-structure. The lack of a language independent specification is also a severe drawback when interoperability between different implementations is desired.

Nonetheless, by using the target programming language directly, the freedom of the programmer to choose which data-structure is fit for each purpose, regardless of their external format, is preserved, in contrast to being imposed by the data-description compiler and it can be hidden as necessary. In some demanding applications, this added flexibility can be more desirable than a language independent specification, especially if the external format is standardized and well known.

In this context, the Externalization Template Library (XTL), demonstrates how a careful use of modern C++ features [11, 3] can be used to make the development of efficient conversion procedures directly in the programming language convenient to the programmer without compromising on performance. It is also shown how the resulting traversal procedures can also be used as generic traversals for several other purposes.

The rest of this text is structured as follows: Section 2 outlines some design issues and tradeoffs considered in the design of the XTL; Section 3 presents an overview of the architecture; Section 4 depicts the application programmer's interface to the XTL, showing how it is comparable in simplicity to a data-description language, while being significantly more powerful and flexible as is shown in Section 5; Section 6 examines the performance of the XTL, including a benchmark and its evaluation; Section 7 describes some applications of the XTL and discusses the possibility of using procedures written with it as a generic traversal mechanism for C++ data-structures.

2 Design issues

The design of general purpose externalization systems poses two kinds of problems. The first is how to represent each of the data elements externally, so that they can be correctly decoded when being internalized. This problem is not addressed here, as the XTL can be configured to generate and decode several widely accepted external formats.

The second problem is how to traverse data-structures in order to externalize each element contained in them and, when internalizing, how to correctly rebuild complex data-structures from the external "flat" representation.

There are two possible solutions to this problem: the developer either explicitly writes procedures to do the traversal or uses a data description compiler to generate both the data-structure definition and the associated

[1] Externalization is also known as marshaling, serialization, linearization, or pickling.

traversal procedures from a simple specification[2].

Using a data description language to describe data-structures is usually easier for the programmer and the resulting externalization code, being machine generated, can sacrifice readability and be optimized for efficiency.

However, the usage of a data description language has some drawbacks, especially in the context of object-oriented programming and for persistence. The first of these is that usually they are tightly coupled to a particular distributed programming toolkit and are not usable for other purposes, as for instance, persistent storage.

The second problem is that data-structure definitions must be generated by the special compiler and as such, the programmer looses the freedom to hide them inside class definitions. In an object-oriented environment this is not desired but the alternative is to use an intermediate data-structure which is not encapsulated and then convert it back and forth to the encapsulated structure. This extra conversion step is both awkward and inefficient.

Because data description languages aim to be independent from specific programming languages, there is also the problem of mapping high-level specifications in the data description language to low-level data-structures in a programming language. For instance, a sequence in the CORBA interface definition language is usually mapped to an array [1]. However, this seldom is satisfactory for the code that processes it, which might require something different, for example, an Standard Template Library (STL) container [10]. Note that selecting the STL container, or any other data-structure, for that matter, wouldn't solve the problem, as there are situations where the array is precisely what is required.

A possible solution to this problem is to complement the data description with programming language specific configuration. The configuration language conveys further information about the mapping from abstract specifications to concrete constructs in a specific programming language. Although this would be an improvement over a simple data description language, it would introduce yet another degree of complexity and somewhat defeat the purpose of language independence in data-descriptions. It can also be pointed out, that in the extreme, this configuration language would have to be as powerful as the target programming language, to allow for every possible mapping.

Hand-written externalization procedures do not suffer from any of these limitations. On the other hand, they have some of their own. The most notable of these is the dependence on a specific programming language, which is especially important when externalization is used for heterogeneous distributed programming. Nonetheless, this limitation can be somewhat circumvented by using standard external formats.

In addition, writing carefully hand-optimized code to produce fast externalization and internalization procedures, is usually a complex, error-prone and time consuming task, especially if separate externalization and internalization procedures must be written for each data-structure, as happens, for instance, with the CORBA Externalization Service [2] and with almost all others.

On the other hand, if some framework that enables the programmer to write simple externalization procedures is available, it may severely impact the possibility for compiler type-checking and optimization. This is the case of the original XDR library by Sun [4], which has predefined procedures for the most common composite data-structures and where the same procedure can be used both for externalization and internalization.[3].

In short, the conflicting design goals are ease of use, performance, flexibility and interoperability. By using a special compiler, it is possible to achieve most of them, except flexibility as it restricts which data-structures can be used, what is the purpose of externalization and poses several problems in the context of object-oriented programming.

Hand-written externalization procedures provide maximum flexibility. However, easy to use frameworks tend to be inefficient and interoperability is dependent on the standardization of external representations.

In this context, the XTL provides a way to write externalization procedures directly in the C++ programming

[2] In fact, there is a third solution which is valid only for languages with reflection [6], where the traversal procedures can be programmed at the meta-level as happens with Java serialization [7]. However, as this is not valid for C++ it is not further discussed here.

[3] Most interestingly, the associated compiler `rpcgen`, outputs special inline code to improve performance. As a result, the generated code looses the simplicity and elegance of its less efficient counterpart.

language, which are very simple to write but efficient. It also supports most features of C++, including some widely used programming idioms and several standard external data-formats.

3 Architecture

The XTL is composed of three layers. The bottommost layer is the *buffer driver* which is responsible for storing the external representation produced by the middle layer[4]. Currently, there are three options for this layer, which store data in memory buffers, C-style file streams or a C++-style streams.

The middle layer is the *format driver* and defines the external data format. It accepts requests to externalize each of the basic data types (e.g. numbers) and provides some primitives to allow the description of complex data-structures. Note that these complex data structures are abstract data-structures, such as fixed or varying length sequences. The available formats are: CORBA Generic Inter-ORB Protocol Common Data Representation (GIOP CDR) [1], CORBA Externalization Service [2], RFC1832 External Data Representation (XDR) [9] and plain text

The mapping of these high-level concepts to concrete data-structures in the C++ language is done at the topmost layer, the *object stream*. For instance, it is possible to map sequences to arrays or containers.

The object stream layer is also the visible programming interface to the XTL and defines how "easy" is to write externalization procedures. In this context, "easy" means several different things:

- how much code the programmer has to write;
- how much the programmer has to learn to write the code;
- how much help the programmer gets from the compiler to ensure the correctness and efficiency of the code.

[4] And for retrieving the external representation and feeding it to the upper layer, when internalizing. For the sake of simplicity, we describe the architecture solely in terms of externalization. The internalization process is symmetrical and can easily be inferred.

The interface of the XTL is designed to target all these. First, as the programming interface is somewhat modeled upon the original XDR library, a single procedure for each data type is enough for both externalization and internalization. This happens because externalization procedures are in fact general purpose traversals of the data-structure, which can be used for other purposes. For instance, the XDR library also uses them to free memory.

The resemblance to XDR will also help many programmers accustomed to it to quickly understand how to write externalization procedures with the XTL. However, the XTL takes advantage of function overloading to make it even simpler than XDR, as all basic C++ data types and most of the composite data types are externalized with the same method name. Different method names are only necessary when there is the need to disambiguate a C++ construct that has several distinct meanings. For instance, `char*` can be a reference to a single character; an optional character, absent if `NULL`; a zero-terminated string or even raw data.

The use of template functions, instead of generic pointers and function parameters, allows the compiler to further check the correctness of the code, ensuring that each data-structure is accessed only by the correct externalization procedures[5].

Finally, the XTL directly supports most of the features of C++, such as template classes; externalization through pointers to, possibly abstract, base classes, STL containers and container-style strings. In addition, as externalization procedures are methods of the respective classes, no internal structure needs to be shown and encapsulation is preserved[6].

4 Programmer's interface

4.1 Streams

The externalization process is initiated by creating a *stream*. This is done by selecting the appropriate buffer

[5] Templates are also essential for efficiency. See Section 6 for further discussion of this topic.

[6] There is however the possibility to write the externalization procedures as global functions, which is useful for externalizing objects of classes which must not be modified. See Section 5.1 for an example.

and format drivers and by stacking them together. For example, a CORBA GIOP CDR output stream to memory is declared as:

```
char buf[SIZE];
mem_buffer mb(buf, SIZE);
GIOP_format<mem_buffer> gfmb(mb);
obj_output<GIOP_format<mem_buffer> >
    stream(gfmb);
```

An XDR input stream from a file is declared as:

```
FILE* fp=fopen("stuff", "r");
cfile_buffer fb(fp);
XDR_format<cfile_buffer> xffb(fb);
obj_input<XDR_format<cfile_buffer> >
    stream(xffb);
```

A stream can then be used to externalize, or internalize, any data-structure, by invoking the appropriate methods.

4.2 Numbers

All native C++ data types (numbers and booleans) are externalized and internalized with method `simple`. For instance, an integer declared as:

```
int i;
```

can be externalized or internalized by:

```
stream.simple(i);
```

The kind of stream, `obj_output` or `obj_input`, determines whether the operation performed is externalization or internalization.

4.3 Composites

To externalize composite data-structures (i.e. structures and classes) a *filter* must be defined. A filter is just a public template method of the structure or class and is used both for externalization and internalization. The body o the filter method enumerates and describes each of the members of the composite. As each method of a stream returns a reference to itself, invocations on the same stream can be cascaded, as in:

```
struct S {
    int a, b;
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(a).simple(b);
    }
};
class C {
    float a, b;
 public:
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(a).simple(b);
    }
};
```

After defining appropriate filters, instances of classes and structures can be externalized and internalized as native data types with method `simple`:

```
S s; C c;
stream.simple(s).simple(c);
```

Furthermore, inheritance is supported by recursively calling the externalization method on all base classes:

```
class D: public C {
    int d, e;
 public:
    template <class Stream>
    void composite(Stream& stream) {
        C::composite(stream);
        stream.simple(d).simple(e);
    }
};
```

However, this does not work if externalization is called through a pointer to the base class. In this situation, the runtime type of the object needs to be used, as described in Section 4.10.

4.4 Strings

There are several representations of strings in C++, such as C-style zero-terminated strings and container-style strings. Zero-terminated C-style strings are externalized with method `cstring`. If space for the string is statically allocated, the XTL must be informed of the maximum size:

```
char s[LEN];
stream.cstring(s, LEN);
```

If the string is dynamic, the XTL will always reserve the necessary space for the string. The previous value will be deleted with `delete []` if not `NULL`, so is up to the programmer to ensure that it is properly initialized prior to internalization.

```
char* d=new char[LEN];
stream.cstring(d);
```

Container-style C++ strings are internalized and externalized with `simple`, just as any other composite:

```
string s;
stream.simple(s);
```

The external representation of all kinds of string is the same, and as such, interchangeable (e.g. a string can be externalized from a C string and internalized to a C++ string).

4.5 Arrays

The XTL supports two distinct kinds of arrays: statically allocated with fixed size or dynamically allocated with variable size. Fixed size arrays can be externalized and internalized with `vector`:

```
int a[10];
stream.vector(a, 10);
```

Variable sized arrays, composed of a scalar size and the array itself, are externalized and internalized with `array`:

```
int size=LEN;
int* a=new int[size];
stream.array(a, size);
```

As happens with dynamic C-style strings, the XTL will always reserve the necessary space for the array. The previous value will be deleted with `delete []` if not `NULL`, so is up to the programmer to ensure that it is properly initialized prior to internalization.

The XTL supports only arrays of elements which can be externalized and internalized with `simple`. If that is not the case, data must be wrapped within a structure with an appropriate filter.

Arrays of bytes which are already in a suitable external representation (e.g. an well known graphics format), don't need to be converted. In this situation, `vector` and `array` should be replaced with `opaque` and `bytes`, respectively.

4.6 Pointers

Pointers in C++ can express references to data items which are always present or optional data items, depending on the pointer being or not allowed to be null.

References to data items which are always present can be internalized and externalized as:

```
int* a=new int(10);
stream.reference(a);
```

Optional data items can be internalized and externalized as:

```
int* a=(int*)0, *b=new int(10);
stream.pointer(a).pointer(b);
```

As happens with dynamic arrays and C-style strings, the XTL will always reserve the necessary space for the pointed structure. The previous value will be deleted with `delete` if not `NULL`, so is up to the programmer to ensure that it is properly initialized prior to internalization. As with dynamic arrays, the referenced or optional data item must be externalizable and internalizable with `simple`.

4.7 Templates

Template types are supported by the XTL without any further complexity:

```
template <class T>
class C {
    T a, b;
 public:
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(a).simple(b);
    }
};
```

and are used just as any other composite data-type:

```
C<int> ci;
C<float> cf;

stream.simple(ci).simple(cf);
```

Notice that if `C` is instantiated with a type that is not externalizable with `simple`, it will result in a compilation error.

4.8 Containers

A special case of composites are the STL containers which do not have the required filter methods. However, as they can be accessed through their external interface it is possible to externalize them with:

```
list<int> li;
stream.container(li);
```

This filter may also be used for any other standard or even user-defined containers which support a forward iterator. The elements of the container must be externalizable with `simple`. In order to externalize `maps` you have to define an appropriate overload of global function `composite` for each instanciation of maps.

If supported by the compiler[7] all containers, including maps, can be externalized with `simple`, without the need for defining filters for specific pairs.

4.9 Unions

Unions in XTL must always be discriminated by an integer value to select the appropriate externalization procedure:

```
int discr;
union {
    int i;
    float f;
} val;
```

Externalization is accomplished by calling `choices`:

```
stream.choices(discr, val.i, val.f);
```

where the first element is the discriminator and the following variable number of elements are the alternative values of the union. The alternatives are numbered from 0 for the leftmost and counting up, that is, the preceding example, 0 indicates an integer value and 1 a floating point value.

Even if the use of unions is not very common in C++, they are useful in the context of externalization to set a version number on the externalized data and then to recognize multiple versions upon internalization.

[7] As of release 1.3, G++ 2.95 only.

4.10 Objects

Pointers to base classes are in the XTL considered an implicitly discriminated union of pointers to all possible derived classes of the pointer type. For instance, in the following example:

```
class B {
 public:
    virtual void f()=0;
};
class D1: public B {
    int a;
 public:
    virtual void f() {
        // something with a
    }
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(a);
    }
};
class D2: public B {
    float b;
 public:
    virtual void f() {
        // something with b
    }
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(b);
    }
};
```

instances of both `D1` and `D2` can be externalized through a pointer to `B` with:

```
B* p=new D1;
stream.object(p, (D1*)0, (D2*)0);
```

Notice that the first parameter is the pointer itself and the following are all possible concretizations of `B`. As a consequence, if `B` was not abstract, it would have to be listed as a possible concretization of itself:

```
stream.object(p, (B*)0, (D1*)0, (D2*)0);
```

All parameters but the first are used just to instantiate the template, so any value is correct, as long as it is of the correct type.

5 Advanced features

5.1 Third-party classes

When the externalization of some data-structure with `simple` is required but it is not possible to add the filter method, it is possible to achieve the same result by overloading the global template function `composite`.

For example, to allow the externalization of STL lists with `simple` all it takes is:

```
template <class Stream, class T>
void composite(Stream& stream,
               list<T>& list) {
    stream.container(list);
}
```

Overloading of this global function can also be used to override default filters defined as methods.

5.2 Separating input and output

Sometimes it is not desired, or even possible, to use the same filter for both externalization and internalization. For instance, with structures which have a peculiar internal representation that does not map directly to the desired external format.

One example is the externalization of the XTL `mem_buffer`, which is internally represented by three pointers into a buffer:

- `buffer` points to the start of the data;
- `ptr` points to the next position to be read or written;
- `lim` points to the first position past the end of the buffer.

The adequate external representation is obviously the same as the one produced by `bytes`[8] plus an index. However, it is impossible to use that filter because the size is implicit in `lim-buffer`. To circumvent the problem, when externalizing, a temporary variable must be used to compute the size. When internalizing, the size is used to compute `lim`.

This can be done by defining filter methods separately for input and output streams:

[8] See Section 4.5.

```
template <class Format>
void composite(obj_input<Format>&
                          stream) {
    int size, idx;
    stream.bytes(buffer, size)
        .simple(idx);
    ptr=buffer+idx;
    lim=buffer+size;
}
template <class Format>
void composite(obj_output<Format>&
                            stream) {
    stream.bytes(buffer, lim-buffer)
        .simple(ptr-buffer);
}
```

5.3 Pointer aliasing

Pointer aliasing is quite common in C++ data-structures. If this possibility is not anticipated by the externalization code, multiple copies of objects will be written to the external representation and internalized as duplicates. Besides being a waste of space, it can also be an error if the program depends on the aliasing for sharing, which is often the case.

Even worse, if pointer aliasing results in a cyclic data-structure (e.g. a circular linked list), an attempt to externalize it without taking aliasing into consideration will result in an endless loop.

To circumvent this problem, the XTL object stream can be parameterized with a strategy class [5] `graph_refs` which intercepts operations with pointers, searching for duplicates upon externalization and correctly aliasing pointers upon internalization:

```
obj_output<text_format<mem_buffer>,
                   graph_refs> os(mb);
```

5.4 Smart-pointers

Reference counting with smart-pointers is a common strategy for managing pointer aliasing in C++. The XTL supports the externalization of these objects by simultaneously using the techniques described in the previous two sections.

For example, typical externalization methods to be defined as part of a smart-pointer to count references to ob-

jects of class C would be:

```
template <class Format>
void composite(obj_input<Format>&
                         stream) {
    if (ptr && !ptr->decref())
        delete ptr;
    ptr=(C*)0;
    stream.pointer(ptr);
    if (ptr) ptr->incref();
}
template <class Format>
void composite(obj_output<Format>&
                          stream) {
    stream.pointer(ptr);
}
```

The extra work of the internalization procedure ensures that the reference counts are correctly updated and avoids erroneous deletions of shared objects.

5.5 Polymorphic streams

Although the XTL defaults to statically composed and optimized streams which result in fast and fairly compact code, in certain situations, it is desirable to take a different approach.

For instance, if many different combinations of buffer and format drivers are used within the same program, the generated code grows accordingly. In these situation, polymorphic wrappers for buffer drivers reduce the size of the generated code to be proportional to the number of distinct components and not the number of distinct combinations. On the other hand, this wrapper reduces the possibility of compiler optimizations and as such has some impact on performance.

5.6 Generalized objects

Pointers to base classes as considered in Section 4.10 are only adequate for small inheritance hierarchies. There is also the possibility of using bigger hierarchies, although there is a small performance penalty. To use this mechanism, all classes have to be annotated with `decl_externalizable(B)`. For instance, using the same example class hierarchy:

```
class B {
    decl_externalizable(B);
 public:
    virtual void f()=0;
};

class D1: public B {
    decl_externalizable(D1);
    int a;
 public:
    virtual void f() {
        // something with a
    }
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(a);
    }
};
class D2: public B {
    decl_externalizable(D2);
    float b;
 public:
    virtual void f() {
        // something with b
    }
    template <class Stream>
    void composite(Stream& stream) {
        stream.simple(b);
    }
};
```

In addition, in an implementation file you need to create an index of classes that can be externalized:

```
externalizable_index idx;
impl_externalizable(D1, 1, idx);
impl_externalizable(D2, 2, idx);
```

Instances of both `D1` and `D2` can be externalized through a pointer to `B` with:

```
B* p=new D1;
stream.auto_object(p);
```

6 Performance

As the XTL does not rely on a special compiler to generate efficient externalization procedures, some care must be taken to ensure that the code written by application programmers is efficient.

	optimized	not optimized
memcpy	4.0	5.0
SunXDR	35.0	36.6
no conversion	10.1	63.7
XDR	14.2	73.7
GIOP CDR	12.8	89.2

Table 1: Performance measurements. All times in microseconds.

This goal is achieved by making sure that, unless the programmer explicitly requires otherwise, all variables and function calls are bound at compile time. As such, even though the XTL code is deeply nested, all functions can be inlined, which means that deep invocations can be collapsed to sequential code.

As a consequence, the compiler has the opportunity to perform extensive optimization, both for speed as well as for code size, by reordering instructions and removing unused code.

The result is that externalization code written with the XTL is very fast, when compared to a similar library where these optimizations are not possible. Table 1 shows measurements of the time necessary to externalize a deeply nested complex data-structure using different format drivers. The tests were run on a Pentium Pro 200MHz machine running Linux 2.0 and using the `egcs` 1.0.2 compiler both with and without optimization. The size of the external representation of the data-structure used for testing is about 400 bytes, depending on the format used.

As it can be seen, using the XTL with no format conversion results in times very close to the theoretical maximum, measured by copying the same amount of data with `memcpy`.

It is also notable that when converting to XDR format, the optimized XTL code is more than twice as fast as the code produced by `rpcgen` for the same structure. Notice that the XTL version required absolutely no effort by the programmer to optimize it while the code produced by `rpcgen` already uses some explicit inline macros, which makes it more complex.

Finally, if we compare the effect of compiler optimization, it can be seen that the code produced by `rpcgen` is only marginally affected by optimization, which is the consequence of dynamic binding, while with XTL the effect is as much as six fold.

These facts confirm that C++ is adequate for performance sensitive system-level operations, if the developer has the knowledge to select which of the features of the language are appropriate.

7 Applications

Most of the experience with the XTL is based on a previous version of the library. Although it supported only text format and had the lower two layers written in C, the upper layer was already written in C++ and closely followed the current XTL, making them differ mostly on performance.

This library has been used in several projects, both for persistence and communication. For persistence, it has been used to store configuration and data in an object replication system targeted for mobile hosts and to hold the cache for a directory service proxy for disconnected operation [8]. In both these applications, the flexibility and power of the XTL interface have clearly offset the disadvantages of not using a data-description language compiler.

For communication, it has been used as the basis for a remote procedure call library. The use of a human readable external data format has proved to be very useful when debugging, as it was possible to interactively establish a connection to a server, manually issue procedure invocations and immediately examine the results. The XTL makes it possible do the same while developing and then switch to a more compact data format without compromising on efficiency.

The XTL is being used in the development of a reliable multicast protocol. Besides being directly used in the conversion of messages to its external representation, the concepts that were applied to the XTL are also being used to allow that highly configurable protocols are easily assembled by application programmers without incurring in unacceptable overheads.

The XTL is also being used in the LyX project[9] to han-

[9] `http://www.lyx.org`

dle communications between the graphical user interface and the core.

Other application being considered for the XTL is to provide light-weight and high-performance interoperability with CORBA systems at low-level, both with GIOP based protocols, such as the Internet Inter-ORB Protocol, as well as with the Externalization Service.

The XTL could also be used as an efficient multi-format back-end to simplify the development of a data description compiler, which would not require optimization of externalization procedures and would easily work with hand-written code when necessary.

It is also possible to use custom streams, format or buffer drivers which use the traversal procedures for purposes other than externalization. An example is the description format driver, which outputs an abstract description of the structure as text. This feature allowed a remote procedure server to describe itself to a client, which would then generate stubs on-the-fly. Other examples are buffer and format drivers that skip some input or calculate the size of a data-structure in a specific format, making it possible to perform random access to externalized data. Another possible application, but that has not been implemented, would be to use the traversal to mark reachable structure for garbage collection.

These applications show that externalization procedures written with the XTL are in fact generic traversals of data-structures and as such can be used as an implementation of the visitor pattern [5] for purposes other than externalization.

References

[1] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1996.

[2] *CORBAservices: Common Object Services Specification*, volume 1, chapter 8, Externalization Service Specification. Object Management Group, 1996.

[3] ANSI document X3J16/95–0087, ISO document WG21/N0618. *Draft Proposed International Standard for Information Systems: Programming Language C++*, April 1995.

[4] John Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Call*. Springer, Berlin, 1990.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1994.

[6] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

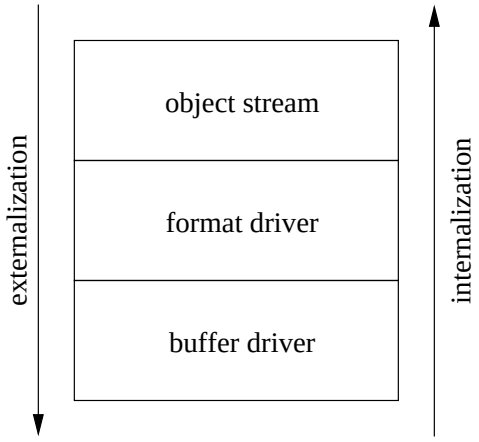
[7] R. Riggs, J. Waldo, A. Wollrath, and K. Bharath. Pickling state in the Java system. *Usenix Computing Systems*, 9(4):291–312, Fall 1996.

[8] A. Sousa, C. Baquero, J. Pereira, R. Oliveira, and F. Moura. A human centered perspective for mobile information sharing and delivery. In *Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, page 412.

[9] Raj Srinivasan. XDR: External Data Representation Standard. RFC 1832, Sun Microsystems, August 1995.

[10] A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[11] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, Reading, Mass., 1997.


externalization
object stream
format driver
buffer driver
internalization