Marshalgen: A Package for Semi-Automatic Marshaling of Objects

Gene Cooperman[1], Ning Ke and Huanmei Wu[2]
College of Computer and Information Science
Northeastern University, Boston, MA 02115 / USA
{gene,nke,maggiewu}@ccs.neu.edu
Phone: 617-373-8686
FAX: 617-373-5121

***Abstract*— Marshaling or serialization of objects is an important component of both distributed and parallel computing. Current systems impose a significant burden on the programmer for describing the marshaling of complex, recursive data structures. Marshalgen provides a semi-automatic process for marshaling in C and C++. Marshalgen avoids the need for complex IDLs and auxiliary routines. It is based on annotations of the existing source code.**

I. Introduction

Marshalgen is a package for semi-automatically marshaling objects. *Marshaling* is the process of copying associated fields of an object into a contiguous buffer in memory. This is critical for internet computing. It is needed to copy an object across the network. Its applications include parallel and distributed computing. It also extends to the computational grid [1], [2], [3].

Marshalgen is different in concept from such well-known marshaling packages as rpcgen [4], Corba IDL [5], and Java serialization [6]. Its purpose and design are distinguished by the goal of providing marshaling services for *existing* software. This provides ease of maintenance as the existing software is transformed through frequent version upgrades.

Marshalgen has an annotation-based strategy, which allows the original application source code to be used unchanged. Marshalgen requires only the addition of a few comment-based directives. A consequence of this design is that the application writer does not need to learn a new *Interface Definition Language* (IDL). The application writer also does not need to create additional data structures or auxiliary code for the objects to be marshaled. In the simplest and most common case, it suffices simply to write `//MSH_BEGIN` and `MSH_END` around the data structure to be marshaled, and then run the file through the Marshalgen preprocessor. In the more complicated and most general case, Marshalgen provides a triple, (FIELDMARSHAL, FIELDUNMARSHAL, FIELDSIZE), of user-specified code fragments to direct Marshalgen in writing the marshaling stubs. Such triples are discussed in Section III-B.

A. Example: Issues in Marshaling Geant4

The decision to write an extensible, object-oriented, semi-automated marshaling package was motivated by the struggles of the first author in parallelizing Geant4 [7], [8], [1], [2]. Geant4 is a toolkit for simulating particle-matter interaction. It comprises approximately one million lines of code. It was developed by RD44, a world-wide collaboration of about 100 scientists in Europe, Russia, Japan, Canada and the United States, participating in more than 10 collider experiments at CERN. It is important, among other reasons, for determining in advance where to place detectors for collider experiments to maximize the chances of detecting events of interest.

The first author parallelized Geant4 using the high level parallelization tool, TOP-C [9], after some other groups had failed to parallelize Geant4. TOP-C provides support for parallelization, but it views marshaling as an external library, similar in spirit to the relation for the core C language with the C stdio library. Parallelizing the typical Geant4 application requires marshaling complex objects, such as arrays, collections of dynamic extent, pointers to static objects present on all machines (which therefore should not be marshaled), and other objects for which separate marshaling functions had to be written. The marshaling has accounted for 250 of the 450 line parallelization of Geant4. The marshaling code is now being rewritten using Marshalgen and about 20 lines of annotation.

Marshaling the complex Geant4 data structures involves several real-world marshaling issues. For instance, one may only want to marshal array elements with odd index if the even index entries are static data. The array may have dynamic content. In marshaling data members of an object, one must choose between deep shallow copying. A data member may be an index or offset into a static table of definitions already present on all machines. Common types are often aliased using the C++ typedef command. This requires marshaling routines to be aliased. Such real-world issues greatly complicate the use of traditional IDL-based marshaling package.

In the example of Geant4, maintaining a correct IDL across version upgrades would require a tedious and error-prone job

[1]This work was supported in part by the National Science Foundation under Grant CCR-0204113.

[2]This work was supported in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821).

```
#include MYCLASS.h              //MYCLASS is a user defined application class

main() {                        //MARSHAL OBJECT FOR SENDING
  MYCLASS obj1();               // Construct an instance, obj1, of MYCLASS
  MarshaledMYCLASS mObj1(obj1); // Marshal it into marshaled object, mObj1
  SendBuffer( mObj1.getBuffer() ); //Send the marshaled buffer to remote host
  ...
                                //RECEIVE A REMOTE MARSHALED OBJECT
  char *mbuf = ReceiveBuffer(); // recv marshaled buffer from remote host
  MYCLASS obj2;                 // obj2 is uninitialized instance of MYCLASS
  MarshaledMYCLASS::unmarshal(mbuf, obj2); // Unmarshal mbuf into obj2
}
```

Fig. 1. `main.cpp` (invocation of marshaling routines)

of rewriting the IDL with each new version. In the Marshalgen alternative, the programmer need only add simple annotations from a small set of approximately ten keywords.

A more subtle motivation for semi-automatic marshaling occurs when one does not have access to the source code. One must then write additional marshaling routines, without the luxury of modifying the source code of the target application. For example, some vendors do not distribute source code. Such an application consists of pre-compiled libraries plus include files with the necessary class declarations. In the case of Geant4, one prefers to use pre-compiled libraries to avoid the burden of compiling a large package. The source files of Geant4, version 5.1, consist of 8.25 Megabytes of source files after compression. They also distribute pre-compiled Geant4 libraries consisting of 16.77 Megabytes.

B. *Organization of Paper*

The paper is organized as follows. Section II briefly discusses related work. Section III describes Marshalgen as seen by the end user. Section IV describes the intermediate IDL generated by Marshalgen. This IDL is then used to generate the traditional stub or skeleton code for marshaling. Future work summarized in Section VI.

II. Related Work

Previous well-known marshaling systems include rpcgen [4], Corba IDL [5], and Java serialization [6] as part of the Java RMI (Remote Method Invocation) facility. The packages rpcgen and Corba IDL are both IDL-based. (The .x file of rpcgen acts as the IDL file.) The packages are not as extensible as Marshalgen. They allow the programmer to marshal compound data structures by specifying the components. However, either all or none of the data structure must be marshaled. Further, there is no provision for marshaling pointers to other objects. In addition, there are numerous marshaling packages tied to a particular software package. Microsoft has designed its own marshalling packages, such as MIDL and DCOM [10]. With the rise of XML, there are now also many packages to marshal data into XML. Foremost among these is XML-RPC [11], a variation of RPC using XML for the marshaled representation. Other XML marshaling packages include JAXB, Castor XML and a lot of others.

Much of the work on marshaling has been concerned with more highly optimized packages for efficiency [12], [13], [14], [15]. Such packages are typically based on marshaling all or none of a data type, and do not allow for pointer members. The Universal Stub Compiler (USC) [15] optimized copying based on user specification.

III. Overview of Marshalgen

We demonstrate the simplicity of the Marshalgen approach through a running example in marshaling linked lists (see Figures 1, 3, 4 and 5). For each application class, `MYCLASS`, Marshalgen produces a new class, `MarshaledMYCLASS`, and a constructor, `MarshaledMYCLASS()`. An instance of `MarshaledMYCLASS` contains a marshaled buffer of `MYCLASS`. The buffer can be unmarshaled by calling a member function `MarshaledMYCLASS::unmarshal()`. An intuitive C++ binding makes the usage easy for the end user. A typical invocation from `main()` is shown in Figure 1.

A. *Marshalgen Framework*

The layers of Marshalgen and the process for building an application using Marshalgen are shown in Figure 2 (a) and (b), respectively.

B. *Annotations*

Marshalgen is distinguished from other marshaling systems in that the end user need only annotate the existing object declaration in the included files. A Marshalgen annotation is specified as a triple, (FIELDMARSHAL, FIELDUNMARSHAL, FIELDSIZE), for each data member to be marshaled. Each element of the triple is a code fragment specifying how to marshal a field, unmarshal a field, or determine the size of the marshaled field. Conceptually, Marshalgen already knows how to marshal primitive data types. So, it suffices to describe how to marshal compound data types. Then Marshalgen can marshal any object by recursively marshaling each data member.

Annotations of a struct or class must be surrounded by `//MSH_BEGIN` and `//MSH_END`. Real-world code often uses typedef to provide a simple name, NEWTYPE, for a common data type. Such a case must also be surrounded by `//MSH_BEGIN` and `//MSH_END` in order for Marshalgen

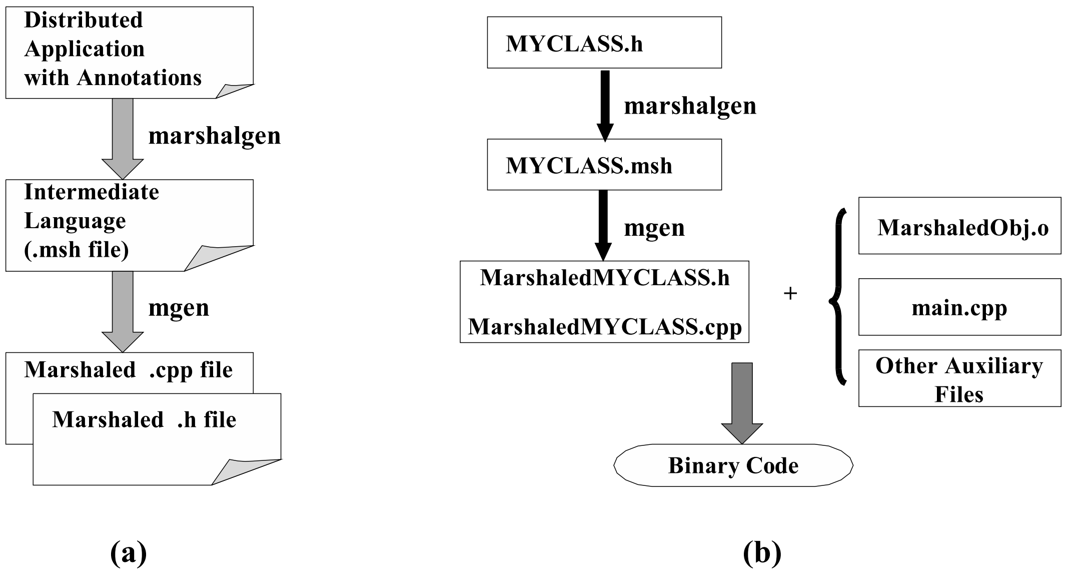

Fig. 2. Overview of Marshalgen (a) The layered view of marshalgen; (b) Building an application using Marshalgen.

```
#include <stdio.h>

//MSH_BEGIN --- beginning of marshaled block
class LinkedList
{
public:
    int head;          //MSH:  primitive
    LinkedList *next;  //MSH:  predefined_ptr
public:
    LinkedList(int = 0 , LinkedList* = NULL);
    bool operator==(LinkedList l);
    bool operator!=(LinkedList l);
};
//MSH_END --- end of marshaled block
```

Fig. 3. `LinkedList.h`: original application file with Marshalgen annotations; The annotations for `head` and `next` are optional, since Marshalgen already knows how to marshal an int or to recursive call itself to marshal a recursive data structure

Default Annotations	**Explanations**
//MSH: primitive	For int, double, char, float and other primitive data types Use built-in marshaling routines
//MSH: primitive_ptr	For int *, double *, and other points to the primitive data Use built-in marshaling routines
//MSH: predefined	For instances of a previously annotated struct or class. Use previously defined MarshaledMYCLASS
//MSH: predefined_ptr	For pointers to previously annotated struct or class Use previously defined MarshaledMYCLASS
//MSH: array	For array with element type from four cases above Use array of marshaled elements

TABLE I
OPTIONAL ANNOTATIONS: ONE OF FIVE DEFAULT CASES, DETERMINED BY PARSING DATA TYPES

to provide a marshaling class with the corresponding name MarshaledNEWTYPE. All the annotations are in the format of *//MSH: annotation_type*. As new cases appear, it is easy to add new types to the design of annotations.

Conceptually, each data member of the object falls into one of five categories below. These five categories are described

```
%{
#include <string.h>
#include "LinkedList.h"
%}

marshaling class MarshaledLinkedList (LinkedList *__obj)
{
  int head;
  // The triple (FIELDMARSHAL, FIELDUNMARSHAL, FIELDSIZE) is omitted
  //    for int, since mgen already knows how to marshal an int.

  LinkedList *next;
  // FIELDMARSHAL:
  { MarshaledLinkedList __m_obj(__obj->next);
    memcpy($BUFFER, __m_obj.getBuffer(), __m_obj.getBufferSize()); }
  // FIELDUNMARSHAL:
  { MarshaledLinkedList __m_obj($BUFFER);
    __obj->next = __m_obj.unmarshal(); }
  // FIELDSIZE:
  { MarshaledLinkedList __m_obj(__obj->next);
    $SIZE = __m_obj.getBufferSize(); }
}
```

Fig. 4. `LinkedList.msh`: IDL generated from annotated `LinkedList.h` file: The recursive definition of MarshaledLinkedList reflects the recursive definition of the original LinkedList class.

to provide a general framework. The actual annotations used by Marshalgen are different, and are described later in Table I. Simple data members need no special annotations for marshaling. There is a default routine for marshaling such data members. Hence, the last four of the five categories are present to handle certain real-world issues that are sometimes omitted in simple demonstrations of marshaling.

1) **default:** A tree traversal strategy based on member data types is employed for marshaling. Refer to Table I for details.
2) **transient:** Don't marshal. Set to default value (e.g. `NULL`) on unmarshaling.
3) **ptr_shallow_copy:** Copy pointer only. This works only on homogeneous architectures. It assumes that the executable is loaded at the same virtual memory address on such homogeneous architectures.
4) **ptr_to_static_table(TYPE TABLE):** Convert pointer into index into static table. The table and its type are specified as a parameter. Table is same on source and destination.
5) **manual(FIELDMARSHAL, FIELDUNMARSHAL, FIELDSIZE):** This is for special application-specific cases not handled above. Code fragments are provided for FIELDMARSHAL, FIELDUNMARSHAL and FIELDSIZE.

The source file Marshalgen is a C++ file with appropriate annotations. As shown in Figure 3, the class is declared in some file, `MYCLASS.h`. Here, we assume that `MYCLASS` is a class for defining linked lists. The Marshalgen annotations are described in Table I.

Marshalgen also has a variation of the syntax of Figure 3 for marshaling more than one original object as a single marshaled object. Marshalgen then produces a new marshaled class with each marshaled object as its field.

IV. Intermediate Language

Marshalgen currently translates the annotations into a `.msh` file. This `.msh` file is then translated into C++ stub code. We directly generate C++ stub code for ease of experimentation. A future version may translate into an IDL file, and data structures and auxiliary functions for any of rpcgen, CORBA or Java serialization.

The grammar of the intermediate language for specification of single object marshaling is as follows.

```
%{
//anything to be included or defined
INCLUDE\_MACROS
%}

marshaling class MTYPE (TYPE OBJ) {

  TYPE1 FIELD1;
    { FIELDMARSHAL }
    { FIELDUNMARSHAL }
    { FIELDSIZE }

  TYPE2 FIELD2;
```

```
class MarshaledLinkedList : public MarshaledObj {
  MarshaledLinkedList(LinkedList source);
  ~MarshaledLinkedList();
  void unmarshal(MarshaledLinkedList m_obj, LinkedList dest);
}
```

```
class MarshaledObj {
public:
  char *msh_buffer; // same as $BUFFER
  char *msh_size;   // same as $SIZE
  char *msh_cursor;
  ...
  inline int getBufferSize() { return msh_size; }
  inline char *getBuffer() { return msh_buffer; }
}
```

Fig. 5. `MarshaledLinkedList.h` and `MarshaledObj.h`: MarshaledLinkedList.h and MarshaledLinkedList.cpp are stub files generated from `LinkedList.cpp` by mgen. The class MarshaledObj is not generated. It is a fixed class in the Marshalgen package.

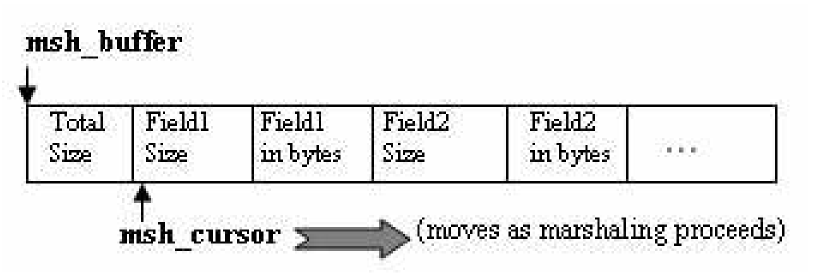

Fig. 6. Internal Architecture of the Marshaled Buffer

```
    { FIELDMARSHAL }
    { FIELDUNMARSHAL }
    { FIELDSIZE }
  ...
}
```

Here, TYPE is the original type (or class or struct) that is to be marshaled. MTYPE is the name of the class that contains the marshaled object. By convention, if TYPE is Foo, then MTYPE is MarshaledFoo.

The Marshalgen IDL is easily readable by a non-expert, allowing for easy customization by the end-user. Figure 4 shows a sample IDL file for a linked list. It corresponds to a Corba IDL file or to a `.x` file in rpcgen. The variable $BUFFER is of type `(char *)`, pointing to a marshaled buffer, and $SIZE is a variable specifying the size of the marshaled data. FIELDMARSHAL and FIELDSIZE must set $BUFFER and $SIZE, respectively, while FIELDUNMARSHAL may read from $BUFFER and $SIZE.

Finally, Marshalgen produces two stub files, `MarshaledLinkedList.h` and `MarshaledLinkedList.cpp` (see Figure 5 below). This is implemented using GNU bison and flex for parsing. The corresponding class is derived from `MarshaledObj`, which maintains an internal buffer $BUFFER, as shown in Figure 6. The next section describes the internals for writing to that buffer.

V. INTERNALS

Internally, marshaling is doing nothing more than converting an object (or a number of objects and values) into a byte array, which is suitable to be sent over the network or be written to a file. Several internal files are part of the Marshalgen package. Two files, `marshalgen.y` and `marshalgen.cpp`, are there to read the marshalgen language. There are two other library files. The file `genfiles.c` is part of the compile-time libray, while `MarshaledObj.cpp` is part of the run-time library which, is a base class for all marshaled objects.

The files `MarshaledObj.h` and `MarshaledObj.cpp` contain code for the basic marshaling. The base class MarshaledObj contains a buffer that stores the marshaled objects. This field of MarshaledObj is called `msh_buffer`. There is also a field called `msh_cursor`, which points to the next field of the object to be marshaled. The pointer `msh_cursor` moves along `msh_buffer` as the object or objects are marshaled. Figure 6 is a diagram of `msh_buffer`.

The sequence in which objects are marshaled is not random. In single-object marshaling, the sequence in which fields of an object are marshaled is the same sequence the fields are specified in the field specification. In multiple-object marshaling, the sequence in which objects are marshaled is the same as the order in which objects are listed in the parameter list.

The sequence in which objects are marshaled can be important. For example, suppose one wishes to marshal an array of objects. The size of the array must be marshaled along with the array. In this case, the size of the array must be marshaled before the array is marshaled, since the size of the array must be known before the array can be unmarshaled. Therefore, the size of the array must precede the array itself in the .msh file specification.

Marshalgen can also gracefully handle marshaling of mutually recursive data structures. This is a slight generalization of the MarshaledLinkedList example. If an object of class A contains a pointer to another object of class B, then marshaling of an object A depends on the marshaling of another object B. But an object of class B may simultaneously contain a pointer to an object of class A. In the `.msh` file, one can use `marshaledB` to specify how A should be marshaled. This case is marshaled automatically using our standard annotations.

The efficiency of Marshalgen with respect to CPU time and memory tends to be excellent. This is because the marshaling code is generated at compile time and targeted specifically at the original annotated class. So there is no run-time overhead, and no padding is neede in writing to the marshaling buffers.

VI. Future Work

This paper presents a new semi-automatic and extensible, object-oriented marshaling package based on annotations. A future version of Marshalgen will further simplify the annotations. More types of annotations will be optional, and determined by parsing.

Two particular issues remain to be addressed before Marshalgen can be applied to large, complex applications, such as Geant4. These are (i) private data members and (ii) class hierarchies containing templates or derived classes. The first issue would arise if the class MarshaledLinkedList had to access private data members of LinkedList. The second issue of class hierarchies is important since template instantiations and derived classes need information from a base class.

We foresee Marshalgen as being added to existing systems. Marshalgen has a small footprint, and is trivial to bring up in a new operating system. Although the current prototype of Marshalgen has been written for homogeneous architectures, it is easy to extend to heterogeneous architectures. One option is to use a package such as XDR (eXternal Data Representation) [16]. Alternatively, Marshalgen can be based on top of CORBA, RPC, Java Serialization (RMI) to gain their support for heterogeneous architectures.

Acknowledgements

We gratefully acknowledge Victor Grinberg and Viet Ha Nguyen for useful discussions and suggestions.

References

[1] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel, “Using TOP-C and AMPIC to port large parallel applications to the computational grid,” in *Proc. of 2^{nd} IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, H. Bal, A. Reinefeld, and P. Lohr, Eds. IEEE Press, 2002, pp. 120–127.

[2] J. H. G. Cooperman, H. Casanova and T. Witzel, “Using TOP-C and AMPIC to port large parallel applications to the computational grid,” *Future Generation Computer Systems (FGCS)*, vol. 19, pp. 587–596, 2003.

[3] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[4] Sun Microsystems, Inc., “Onc+ developer’s guide,” Nov. 1995.

[5] Object Management Group, “The common object request broker: Architecture and specifi cation,” Feb. 1998.

[6] D. Reilly, “Introduction to remote method invocation,” Oct. 1998, online at http://www.davidreilly.com/jcb/articles/javarmi/javarmi.html.

[7] A. D. Acqua *et al.*, “Geant4: A simulation toolkit,” *Nuclear Instruments and Methods in Physics Research Section A*, 2003, (over 100 authors, incl. G. Cooperman), to appear.

[8] Geant4 webpage. [Online]. Available: http://wwwinfo.cern.ch/asd/geant4/geant4.html

[9] G. Cooperman, “TOP-C: A Task-Oriented Parallel C interface,” in 5^{th} *International Symposium on High Performance Distributed Computing (HPDC-5)*. IEEE Press, 1996, pp. 141–150, software at http://www.ccs.neu.edu/home/gene/topc.html.

[10] N. Brown and C. Kindel, *Distributed Component Object Model Protocol — DCOM/1.0*. Microsoft Corporation, Redmond, WA, 1996.

[11] XML-RPC. [Online]. Available: http://www.xmlrpc.com/

[12] T. Braun and C. Diot, “Automated code generation for integrated layer processing,” in *Proc. of IFIP Protocols for High Speed Networks*, Sophia-Antipolis, France, Oct. 1996.

[13] M. Hof, “Just-in-time stub generation,” in *JMLC’97 — Joint Modular Languages Conference*, Linz, Austria, Mar. 1997, pp. 197–206.

[14] P. Hoschka and C. Huitema, “Automatic generation of optimized code for marshaling routines,” in *IFIP TC6/WG6.5 International Working Conference on Upper Layer Protocols, Architectures and Applications*, M. Medina and N. Borenstein, Eds., 1994, pp. 131–146.

[15] S. W. O’Malley, T. A. Proebsting, and A. B. Montz, “USC: A universal stub compiler,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, Aug. 1994.

[16] Sun Microsystems, Inc., “XDR: External Data Representation,” June 1987, RFC 1014.

[17] C. Queinnec, “Marshaling/demarshaling as a compilation/interpretation process,” *IPPS/SPDP*, pp. 616–, 1999.