Marshalgen: Marshaling Objects in the Presence of Polymorphism

Gene Cooperman[1,2] and Viet Ha Nguyen[1,2]
College of Computer and Information Science, 161 CN
Northeastern University, Boston, MA 02115 / USA
{gene,vietha}@ccs.neu.edu
Keywords: middleware, marshaling, serialization, automatic parallelization, polymorphism

***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 support for retrofitting legacy and complex software with marshaling features. The original version of Marshalgen provided a semi-automatic process for marshaling in C and C++, based on annotations of the existing include files. The new version reported on here provides direct support for class inheritance, templates and other important features of an object-oriented programming style.**

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].

Marshalgen is intended to support marshaling of *legacy* and complex software. The goal of Marshalgen is to simplify human-computer interaction. It does so by requiring the user only to add annotations (comments) to the include files. By not requiring the modification of the source code (other than include files), Marshalgen provides two benefits.

1) It supports software libraries distributed as binary only, with include files to support the API.
2) It allows one to marshal large, complex software by minimally invasive modifications to the existing code. This results in more maintainable software.

Version 1 of Marshalgen was previously reported on elsewhere [3]. Loosely speaking, although version 1 of Marshalgen was targeted toward C++, like most other marshaling packages, Marshalgen primarily addressed marshaling of an isolated class. Classes were marshaled in isolation. In cases of inheritance and templates, marshaling was difficult or impossible. Version 2 of Marshalgen addresses these issues of polymorphism.

Polymorphism concerns itself with abstractions that operate uniformly on values of different forms. Polymorphism is supported in object-oriented programming through two main abstractions: inheritance (subtype or inclusion polymorphism [4]) and genericity (templates; parametric polymorphism).

Our motivation was to have sufficient features so as to automatically marshal Geant4, a package for particle-matter interaction. For this purpose, we address marshaling issues closely associated with object-oriented programming. Version 2 adds marshaling support for four language features:

1) Genericity (C++ templates)
2) Polymorphic Access (An object has a compile-time type B, but a run-time type D, where D is one of the derived classes of B.
3) Inheritance (A derived class D inherits data members from a base class B. An object of type D must marshal data members declared directly in D, and also those additional data members declared in B.)
4) Visibility of data members (What does one do with a data member that is protected or private?)

We specifically do not address multiple inheritance. In many cases (including Geant4), C++ programmers try to avoid multiple inheritance. Where they do use it, they use it in a restricted context that can best be served in Marshalgen by user-specified annotations suitable for the specific context.

Cyclic data structures, such as doubly linked lists, represent a special issue. That issue will be addressed in a future work.

Section II provides an overview of Marshalgen. The new support for handling polymorphism and visibility are described in Section III. Section IV concludes with an example motivated by Geant4.

A. Taxonomy of Issues for any Marshaling Package

It is useful to provide a taxonomy of the cases that any marshaling package (not just Marshalgen) will encounter in the real world in marshaling compound types. We illustrate for a class, although the ideas are also valid for an array. It is assumed that primitive types (int, float, etc.) are always marshaled. Pointer types are marshaled along the same lines as class data members that are of pointer type.

1) **Deep copying of data members (default for Marshalgen):** (Marshalgen annotations are optional in this case.)

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

[2]This work is supported by the Institute for Complex Scientific Software (www.icss.neu.edu) of Northeastern University.

a) **pointer member pointing to class, pointer, or primitive type for which a marshaling routine is available**
b) **class member:** invoke the marshaling routine for the class
c) **array member of declared array size:** iteratively marshal each element of the array using the appropriate marshaling routine
d) **dynamically allocated arrays:** annotation includes a parameter to determine the array size
e) **user-written stub functions:** This is for cases not handled above. (Marshalgen provides additional support for this case, by requiring the application writer only to annotate a triple of code fragments for marshaling, unmarshaling and the size of a field: (FIELDMARSHAL, FIELDUNMARSHAL, FIELDSIZE). See [3] for further information.)

2) **Shallow copying of data members:**
a) **transient member:** Don't marshal. Set to default value on unmarshaling. (For example, if a pointer is not used by the remote process, then it suffices to marshal the pointer to the null pointer. An array acts as a pointer in this case. If a class member is not used, the it suffices to marshal an object produced by the default constructor.)
b) **pointer to data that is common to all processes:** Copy pointer only. This works only on homogeneous architectures. It assumes that the pointer points to preinitialized constant data. It assumes that the executable is loaded at the same virtual memory address on such homogeneous architectures.
c) **pointer to array of data that is common to all processes:** Convert pointer into index into array. Array is same on source and destination. Only the index need be marshaled.

Potentially, any case can be handled by a hand-coded stub function (Case 1e above). Version 1 of Marshalgen directly supported all cases except Case 1d, which was handled as a stub function. Version 2 includes direct support for Case 1d. The newer, novel issues of support for object-oriented programming are described in Section III.

B. 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 [5], [6], [1]. Geant4 is a C++ toolkit for simulating particle-matter interaction. It comprises approximately one million lines of code developed by an international consortium centered around CERN. Geant4 is used, among other purposes, to simulate collider experiments, in order to determine where to place particle detectors for the greatest sensitivity. As a toolkit, Geant4 includes libraries only, similarly to many other scientific subroutine libraries, such as LinPACK [7]. It is up to the end-user to write a main routine.

The first author parallelized Geant4 using the high level parallelization tool, TOP-C [8]. TOP-C provided support for parallelization, but it viewed marshaling as an external library, similar in spirit to the relation of the C stdio library to the core C language. The parallelization of Geant4 originally included manual coding of the marshaling routines. That hand-coded marshaling code accounted for 250 lines of the 450 line parallelization of Geant4.

The example of Geant4 demonstrates two motivations for an annotation-based marshaling package, such as Marshalgen.

1) A common mode of distribution for a toolkit is to provide users with pre-compiled binaries, along with include files (.h files). (The pre-compiled binaries for Geant4 are larger than 15 MB, and it would be a great inconvenience to re-compile or maintain a separate set of binaries.)
2) New releases are frequent, and it is easier to modify annotations of declarations in include files than to write entirely new stub functions for each release. (New releases of Geant4 occur up to twice per year.)

C. Related Work

Previous well-known marshaling systems include rpcgen [9], Corba IDL [10], and Java serialization [11] as part of the Java RMI (Remote Method Invocation) facility. 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 [12]. With the rise of XML, there are now also many packages to marshal data into XML. Foremost among these is XML-RPC [13], a variation of RPC using XML for the marshaled representation. Other XML marshaling packages include gSOAP [14], JAXB, Castor XML and many others. In related work, Grogono and Sakkinen have proposed annotations of C++ to distinguish deep copying, shallow copying and gradations between the two extremes [15].

II. Overview of Marshalgen

Marshalgen has an annotation-based strategy, which allows the original application source code to be used unchanged. 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.

For each application class, (`LinkedList` in the example), the user must annotate the include file (add Marshalgen comments). Figure 2 illustrates some annotations for `LinkedList.h`. Some of the possible Marshalgen annotations are listed in Table I.

Marshalgen is invoked by calling `marshalgen LinkedList.h`. This generates C++ code and an include file, MarshaledLinkedList.h, for a class MarshaledLinkedList. The result is code for a new class, `MarshaledLinkedList`, along with a constructor, `MarshaledLinkedList()`. An instance of `MarshaledLinkedList` contains

```
#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 by end user)

```
#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. 2. `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 recursively call itself to marshal a recursive data structure

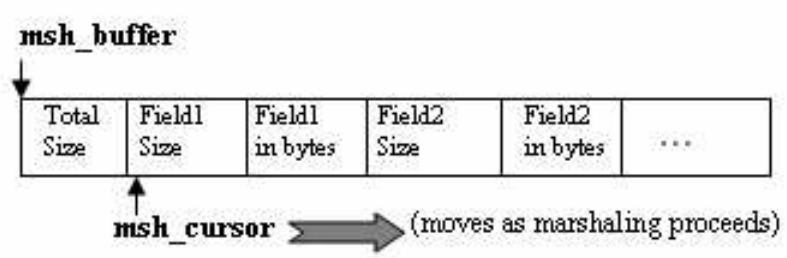

Fig. 3. Internal Architecture of the Marshaled Buffer

a marshaled buffer of `LinkedList`. The buffer can be unmarshaled by calling a member function `MarshaledLinkedList::unmarshal()`. That code is then compiled and linked with the application. Figure 1 shows typical usage of the marshaling methods provided by MarshaledLinkedList. Finally, figure 3 illustrates the data layout of the marshaled buffer produced at runtime.

For more details on the internals of Marshalgen, see [3].

III. Issues and Solutions

A. Polymorphism

In object-oriented programming, an identifier (variable) may have compile-time type B, but run-time type D. Additionally, a class may be a template class, parametrized by a type T. Yet, marshaling routines must be prepared to handle any instance of the template. The marshaling package must be able to correctly distinguish and dynamically dispatch to the proper marshaling routine of each type.

1) Genericity (Templates): A class template may be instantiated with different types, and each type may require a separate marshaling policy.

Consider the example on the left of Figure 4, the type of the data member `T data` may be instantiated to `Bar1` or `Bar2`. Since `Bar1` or `Bar2` may have different data members and require different marshaling policies, the user should be able to specify the circumstances under which marshaling routine for a given class should be used. The annotations for that mechanism are on right of Figure 4.

`IsSameClass<T,Bar1>` is a user-defined C/C++ expression specifying the circumstance under which the user

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
//MSH: ptr_as_array	For pointers to an array with element type from first four cases

TABLE I

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

```
//MSH_BEGIN
template <T> class Foo {
public:
  T data;
};
class Bar1 { ... }
class Bar2 { ... }
//MSH_END
Foo<Bar1> f1;
Foo<Bar2> f2;
```

```
//MSH_BEGIN
template <T> class Foo {
public:
  T data;      /* MSH: predefined
   [elementType:
    (IsSameClass<T,Bar1>) => Bar1
    | true => Bar2]
  */
};
class Bar1 { ... }
class Bar2 { ... }
//MSH_END
Foo<Bar1> f1;
Foo<Bar2> f2;
```

Fig. 4. Annotation for marshaling templates (code on left is before annotation, code on right is after annotation)

wants `T data` to be marshaled as an object of type `Bar1`. Note that the expression inside `(...)` can be any boolean C/C++ expression. The annotation in this example tells Marshalgen to marshal the data member `T data` as an object of type `Bar1` if `IsSameClass<T,Bar1>` is true, otherwise marshal it as an object of type `Bar2`.

In general, the syntax of the option is as follows:

$$[elementType : phrase_1|phrase_2|...]$$

where the syntax of each phrase is:

$$phrase \rightarrow (C\_boolean\_expression)\text{“}=>\text{”}\ type$$

The phrases will be evaluated from left to right, if the *C_boolean_expression* of a phrase is true, the evaluation terminates immediately, and the data member will be marshaled according the *type* of that phrase.

2) Polymorphic access (multiple derived classes): There is another case in which the run-time type of an object is not known at the time of generating marshaling code. This is the case of union of classes. An object may be declared in the source code to have a type `Base`, but at run-time the object may be a subtype of `Base`.

Consider the example on the left in Figure 5. In this example, the object that `Base* ptr` points to could be of type either `Base` or `Derived1`, or `Derived2`. In order to dispatch to the marshaling routine for the proper subclass, the user can use annotations similar to the template case (see the right of Figure 5).

On the right in Figure 5, the expression inside `(...)` can be any boolean C/C++ expression. In this particular example, `(dynamic_cast<Derived1*>ptr!=NULL)` is a C++ expression equivalent to the `(ptr instanceof Derived1)` expression in Java. It tests whether the object pointed to by `ptr` is an instance of class `Derived1`. The annotation in this example tells Marshalgen to marshal the object pointed by `ptr` as an object of class `Derived1` if `dynamic_cast<Derived1*>ptr!=NULL`, as an object of class `Derived2` if `dynamic_cast<Derived2*>ptr!=NULL`, or as an object of class `Base` otherwise.

Remark 1: Since a goal of Marshalgen is to marshal without modifying the original class hierarchy, it is not an option to add marshaling and unmarshaling methods to the original classes. Even the visitor pattern [16, pp. 331] cannot be invoked, since that requires the addition of an `acceptVisitor` method.

Remark 2: An alternative design is for Marshalgen

```
//MSH_BEGIN
class Base { ... }
class Derived1 : public Base { ... }
class Derived2 : public Base { ... }
class Foo {
public:
   Base* ptr;
}
//MSH_END
```

```
//MSH_BEGIN
class Base { ... }
class Derived1 : public Base { ... }
class Derived2 : public Base { ... }
class Foo {
public:
   Base* ptr; /* MSH: predefined_ptr
    [elementType:
       (dynamic_cast<Derived1*>ptr!=NULL) => Derived1*
     | (dynamic_cast<Derived2*>ptr!=NULL) => Derived2*
     | true => Base* ]
   */
}
//MSH_END
```

Fig. 5. Annotation for marshaling of union of classes (code on left is before annotation, code on right is after annotation)

to implement MarshaledDerived1 and MarshaledDerived2 as derived classes of MarshaledBase. Hence, given an identifier, `mobj`, with compile-time type `MarshaledBase`, one could then call `mobj->marshal()`, and the call would be dispatched automatically to `MarshaledDerived1:marshal()`, `MarshaledDerived2:marshal()`, or `MarshaledBase:marshal()`. The methods `MarshaledDerived1:marshal()`, etc., would then call `Base:marshal()` as part of their implementation. As a result, there is no need for the case-by-case code in the annotation as in Figure 5.

However, this alternative design had to be rejected. The difficulty in the alternative design is that one still has to assign the appropriate run-time type to the identifier `mobj`. For example, given an object `Base* obj` to be marshaled, one has to decide which of the following constructors to call: "`MarshalBase* mobj = new MarshalDerived1(obj)`", "`MarshalBase* mobj = new MarshalDerived2(obj)`", or "`MarshalBase* mobj = new MarshalBase(obj)`". As a consequence, the case-by-case code must be inserted before each call to the constructor for instantiating `mobj`. It is clearly preferable to embed the case-by-case code once only in the .h file as in Figure 5, rather than at each occurence of a call to the constructor as in the alternative design.

3) Inheritance (Subtype Polymorphism): A class may inherit data members from its ancestor classes. When marshaling an object, one usually wants to also marshal the data members it inherits from its ancestors as well.

In the example on left of Figure 6, whenever we marshal an object of type `Derived`, we may also want to marshal the data member `int i` that class `Derived` inherits from class `Base`. Marshalgen does not automatically marshal all the ancestor classes with the targeted class. The reason is that the class hierarchy may be very deep and there may even be multiple inheritances. This would cause Marshalgen to marshal unnecessary ancestor classes. As a result, the marshaling buffer would be unnecessarily large and the marshaling process would be inefficient.

For that reason, Marshalgen does not marshal data members of an ancestor class unless explicitly requested by the annotation `//MSH_superclass`. An example of that annotation is on right of Figure 6.

The data members of class `Base` to be marshaled are specified by separate annotations inside class `Base`.

B. Visibility of Data Members

A good object-oriented programming style would hide most of the data members from direct access from outside by declaring them `protected/private`. Since we are not allowed to add accessor/modifier methods to the original class, we may need a mechanism to access `protected/private` data members of the targeted class from outside.

a) General solution (does not work).: First, we tried to access the private data members of an object by casting the object to a "masked class". The masked class has the same data members as the targeted class, with all of them are declared public.

```
class OriginalClass{
private:
        int i;
        double f;
};
OriginalClass* org = new OriginalClass();
...
// the new defined class
class MaskedClass{
public:
        int i;
        double f;
}

MaskedClass* m = (MaskedClass*)org;
// allows us to access private data members
int a = m->i;
double d = m->f;
```

The issue with this approach is when we have the targeted class inherits data members from several ancestor classes, the

```
//MSH_BEGIN
class Base
{
public:
  int i;  //MSH: primitive
};
class Derived : Base
{
public:
  Bar b; //MSH: predefined
};
//MSH_END
```

```
//MSH_BEGIN
class Base
{
public:
  int i;  //MSH: primitive
};
class Derived : Base
{
public:
  Bar b; //MSH: predefined
  //MSH_superclass: Base
};
//MSH_END
```

Fig. 6. Annotation for marshaling in the case of inheritance (code on left is before annotation, code on right is after annotation)

layout order of data members from ancestor classes may vary among compilers (we observed the discrepancies between `gcc` and `SunCC`). It makes the construction of the masked class impossible in general.

b) Conservative solution.: Since our package is a source-to-source preprocessor, we can not use the masked class approach, which depends on the data member's layout of specific compilers. Instead, we assume that any well-designed class should have accessor/modifier methods for each significant data members (the data members worth being marshaled). The access to `private` data members can be done via the corresponding accessor/modifier methods.
This assumption, of course, is not true for all programs, but it is true for most programs we have encountered so far. This includes Geant4.

IV. An Example from Geant4

The above issues occur when we use Marshalgen Version 2 to marshal objects in parallelizing Geant4. In order to do parallel computation in a distributed environment, we need to be able to marshal and send over network the objects representing events (`G4HCofThisEvent` and related classes in Figure 7) and the objects representing simulated hits and particles (`G4VHitsCollection`, its derived class `G4THitsCollection` and other customized classes in Figure 8).

The Geant4 toolkit allows users to define their own types of hits or particles. As a result, most of the classes representing collections of hits or particles are designed as templates, allowing users to "plug-in" (instantiate) those templates with their customized types. The Marshalgen package has correctly handled the marshaling of templates in Geant4, dispatched according to the types the users instantiated.
Moreover, Geant4 is a toolkit designed to allow as much as generality for the users. As a result, the class hierarchy is relatively deep. As an example, the classes representing the hits are of 4-5 levels deep. Marshalgen has correctly handled the marshaling of classes both with many ancestor classes and multiple potential derived classes.

Acknowledgements

We gratefully acknowledge Victor Grinberg and David Lorenz for useful discussions and suggestions on issues of human-computer interaction, and object-oriented design, respectively. We also gratefully acknowledge Eugenio Korolev, Igor Malioutov and Eric Smith for their contributions to modifying and testing Marshalgen.

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," *Future Generation Computer Systems (FGCS)*, vol. 19, pp. 587–596, 2003, (also appeared in Proc. of 2^{nd} IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)).

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

[3] G. Cooperman, N. Ke, and H. Wu, "Marshalgen: A package for semi-automatic marshalling of objects," in *Proc. of The 2003 International Conference on Internet Computing (IC'03)*. CSREA Press, 2003, pp. 555–560.

[4] L. Cardelli and P. Wegner, "On understanding types, data abstractions, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.

[5] S. Agostinelli *et al.*, "Geant4: A simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, 2003, (over 100 authors, incl. G. Cooperman).

[6] Geant4 webpage, http://wwwinfo.cern.ch/asd/geant4/geant4.html.

[7] LinPack webpage, http://www.netlib.org/linpack/.

[8] 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.

[9] Sun Microsystems, Inc., "ONC+ developer's guide," Nov. 1995.

[10] Object Management Group, "The Common Object Request Broker: Architecture and specifi cation," Framingham, MA, USA, 1999, minor revision 2.3.1, OMG TC Document formal/99-10-07.

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

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

[13] XML-RPC, http://www.xmlrpc.com/.

[14] R. van Engelen and K. Gallivan, "The gSOAP toolkit for web services and peer-to-peer computing networks," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*. Berlin, Germany: IEEE Press, 2002, pp. 128–135.

[15] P. Grogono and M. Sakkinen, "Copying and comparing: Problems and solutions," *Lecture Notes in Computer Science*, vol. 1850, pp. 226+, 2000.

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

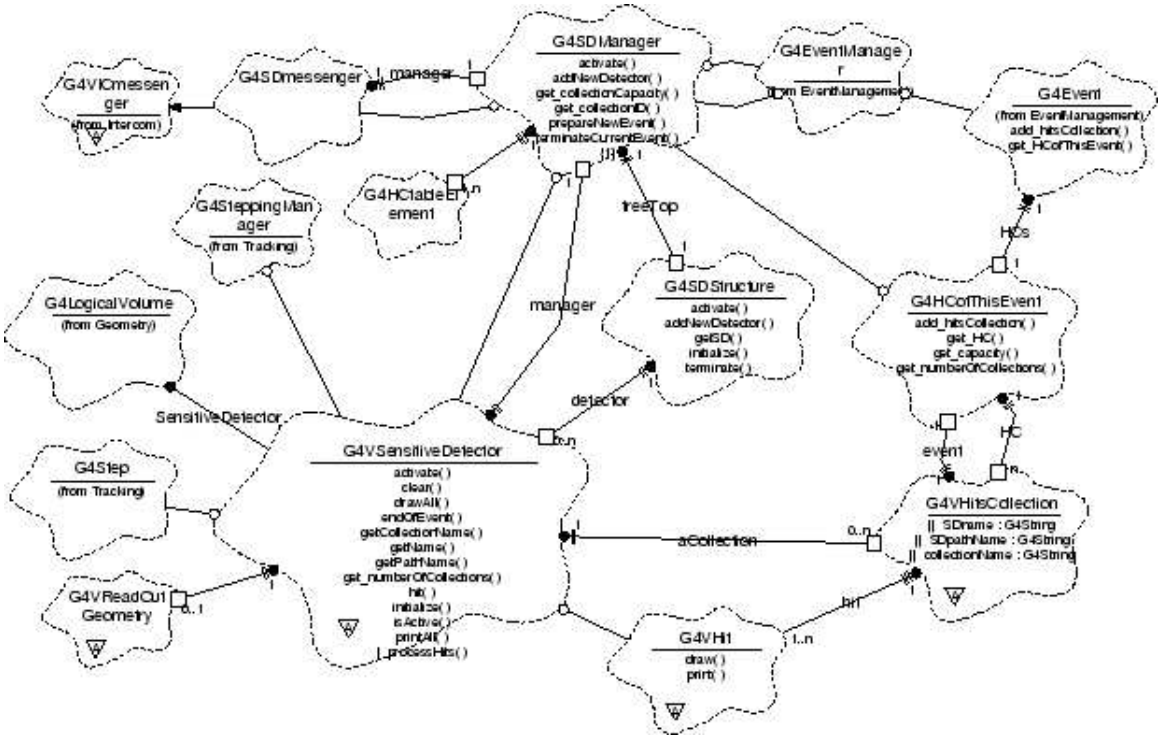

Fig. 7. Event and Hit Management classes

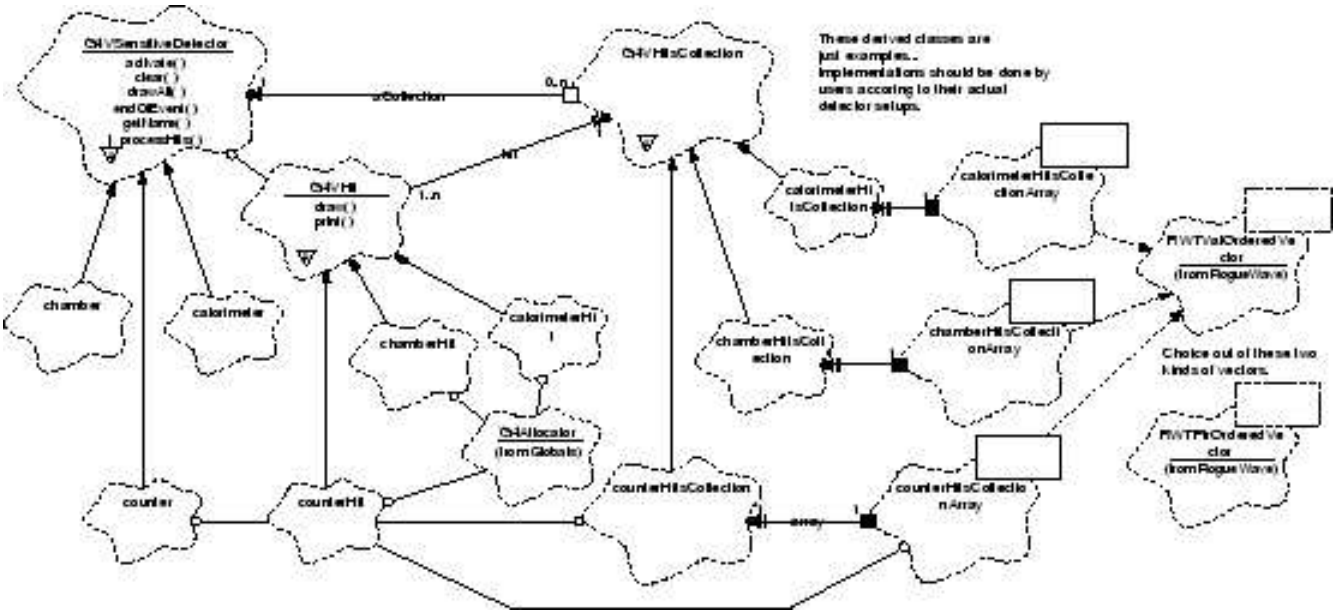

Fig. 8. Hit Collections classes

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