Doc Number X3J16/91-0128
WG21/N0061

Meta - Object - Protocol: A Runtime Type Information System for C++

Frank Buschmann
Konrad Kiefer
Johannes Nierwetberg

SIEMENS AG
Corporate Research and Development
ZFE IS SOF 3
Otto - Hahn - Ring 6
8000 München 83
Germany

E-Mail: mop@ztivax.siemens.com
kk@ztivax.siemens.com

ABSTRACT

In the area of C++ programming a sophisticated mechanism for runtime type identification and access to runtime type information is required to support many applications and libraries. The present paper describes an adequate runtime type information system for the C++ programming language: the **Meta-Object-Protocol** (MOP). An integration strategy is presented to demonstrate how type information can be generated and how the MOP can be included and used within C++ applications. To use the MOP, classes must be extended with a static data member which contains the type information about the class and with additional functionality to access this data member from outside the class and to identify its type. The MOP itself is implemented in a library class called **Meta**, which is used to access the generated type information. The class Meta includes member functions to access type information about classes, data- and function members of classes and their types. The MOP follows the object-oriented approach and supports typesafe downcast, development of libraries and toolkits and integration of different libraries.

1 Introduction

Various attempts have been made in C++ to implement a method of type identification, access to type information for objects and classes and a mechanism to access additional, application specific type information /3/, /4/, /5/, /6/. There are several reasons for the need of such methods and mechanisms:

- Support for typesafe downcast
- Support for integration of different libraries
- Support for library and toolkits

In this paper we examine the problem of type identification and accessing type information in C++. We introduce a library class **Meta** which supports access to different type information at runtime. The functionality of the class Meta is called the **Meta-Object-Protocol** (MOP). An integration concept is presented to demonstrate how the type information is generated and used within C++ applications. MOP supports the requirements resulted from the different problems enumerated above. It provides access to information about classes, data- and function members of classes and their types. Additionally extensions of classes which are necessary for type identification and accessing type information are described.

The development of MOP was initiated by a Siemens internal working group, consisting of representatives from corporate research and development and from those operating divisions of the company which are dealing with (object-oriented) software production, e.g. for the automation, telecommunication and power generation domain.

The MOP is a proposal for a possible runtime type information system that should be included into the C++ programming language. However, MOP consists of more functionality than it is useful to integrate into the language. The description of MOP is therefore focused on those parts of the MOP methods which are - in our opinion - useful for a standardized runtime type information system. The additional functionality is just described briefly to introduce the whole concept of MOP and to demonstrate the functionality required by large scale industrial software developers.

2 The Concept of MOP

At present, type identification and access to type information are not integrated into the C++ programming language. Since many C++ applications need access to type information, a Siemens in-house project is launched to provide a type information system with a full set of member functions required by the company's operating divisions. It is called the **Meta-Object-Protocol** (MOP). This runtime type information system is realized via extensions to classes which allow type identification and access to type information. In addition a library class **Meta** is implemented which includes the member functions of MOP.

In this chapter we introduce the concept of MOP and underline our interest in standardizing type information for C++. Features of a language extension that might be helpful for type information systems like the MOP are listed at the end of this chapter.

The concept of MOP follows the object-oriented approach. The functionality of MOP is divided into three layers. The three layers consist of member functions with access to:

Layer 1:	type information about classes
Layer 2:	type information about data- and function members of classes and their types
Layer 3:	information about relative addresses of data members and functions which support storing and reading of objects. The goal of this layer is to provide functionality which supports the implementation of persistency.

The introduction of the concept is focused on the MOP layer 1, but we give a brief overview over the current status of the whole concept. Up to now implemented and tested is just MOP layer 1.

2.1 Extensions to classes

Every class must have access to information about its type and it must be possible to access this type information from outside the class. To achieve this functionality in the framework of our concept the following extensions must be added to each class:

```
private:
    static Meta classInf;
public:
    virtual const Meta* getMeta() const;
    static const Meta* getInfo() const;
```

The data member **classInf** contains the type information about the class. It is a static variable which does not enlarge the size of instances. This variable must be initialized properly before starting the main routine of C++ applications.

The functions **getMeta()** and **getInfo()** simply return the address of classInf. The virtual function getMeta() is essential for dynamic type checking. In object-oriented software-systems a pointer to a class may point to an instance of a subclass of this class. The current dynamic type of this pointer cannot be determined by the compiler. But correct dynamic type information about the type is needed in some applications. The function getMeta() fills this gap.

Example: Accessing type information of classes

If a common superclass Object (like in NIH class library) for all classes of an application exists, the type declaration Object* does not specify the type of the object to which the pointer points, exact type information about the present type of that object is required.

```
Meta* metaPtr;                  // Declaration of a pointer to type information
Object* objPtr;                 // Declaration of a pointer to an object;
metaPtr = objPtr->getMeta();    // metaPtr points now to type information about the dynamic
                                // type of objPtr.
```

With the help of the static function getInfo() information about the type of a class is extracted without having an instance of the class.

The implementation of getMeta() and getInfo() must also be added to each class, as well as the initialization of the static data member classInf.

If runtime type information becomes part of C++, functions to access type information at runtime like getInfo() and getMeta() must be supported by the C++ language and its compilers.

2.2 MOP layer 1

The MOP itself is implemented in a class called **Meta**. The class Meta includes member functions to access type information about classes stored in the data member classInf. A class like the class Meta must be included into the standard C++ library if type information becomes part of the language. We propose three member functions of Meta for MOP layer 1. The parameter **errflag** denotes irregular behaviour of the functions, e.g. the information was not properly initialized.

const char name (int &errflag) const*

The function **name(...)** returns the name of the class, if necessary with scope hierarchy.

long size(int &errflag) const

The function **size(...)** returns the size of the class in byte.

const Meta baseClassIter(int &errflag, const BaseProps* &bProps, int start = 0) const*

The function **baseClassIter(...)** iterates through the direct base classes of the given class. "start" sets the iteration start at the beginning of the (internal) list of base classes. The properties of a base class are given in the parameter bProps.

```
struct BaseProps {
    int virtuality;     // not virtual == 0; virtual ==1;
    int visibility;     // private == 0; public == 1;
};
```

Passing the information about base classes in the type BaseProps avoids additional functions. But it also means that the struct BaseProps must be defined in a library, too. The function BaseClassIter returns information needed for typesafe casting and for subclass inquiries, but functions for pointercast itself are not realized. Nevertheless there are arguments to support the casts of pointers and references directly /4/, /7/ and to extend the functionality of Meta in this direction.

The realisation of the class Meta is quite simple. It keeps the type information in its private data section. To store the list of base classes and the base properties, simple array lists are used. The full declaration of Meta for MOP layer 1 is as follows:

```
struct BaseProps {
    int virtuality;                 // not virtual == 0, virtual == 1
    int visibility;                 // private == 0, public == 1
    BaseProps(){};                  // constructors of BaseProps
    BaseProps(int virt, int vis);
};

class Meta {
    public:
      // Meta - Object - Protocol for MOP layer 1
      const char* name (int &errflag) const;
      long size(int &errflag) const;
      const Meta* baseClassIter(int &errflag, const BaseProps* &bProps, int start = 0) const;

      // constructor for classes without base classes
      Meta(char* namePar, int sizePar );

      // constructor for classes with base classes
      Meta(char* namePar, int sizePar, MetaArrayList baseList, BasePropsArrayList propsList);

      // default constructor
      Meta(){};

    private:
      // internal data section for the Meta - Object - Protocol
      const char* nameVar;
      int sizeVar;
      MetaArrayList baseListVar;
      BasePropsArrayList basePropsListVar;

    // type information for the class Meta itself (see 2.1.1)
    private:
      static Meta classInf;
    public:
      virtual const Meta* getMeta() const;
      static const Meta* getInfo() const;
};
```

The class Meta has been implemented and tested.

2.3 MOP layer 2

The layer 2 of MOP introduces type information about data- and function members of classes and their types. To ease handling of the different kinds of type information we propose a set of different Meta classes - one for each kind of type information - instead of having one class Meta for all kinds. The kinds of type information that are presently identified are: type information about classes (see chapter 2.2), data members, function members, typedef declarations, enum declarations, pointers, references, arrays and simple types. To organize these classes a base class Meta is introduced that only comprises the virtual MOP. The different classes are derived from Meta and implement those functions of the MOP that are applicable to the type information of the type they represent.

We do not support a direct access to type information about data- and function members and their types. Access to type information about data- and function members is possible only through the type information about classes, to type information about types of data- and function members only through type information about these members. With this concept type information about members is bound to the type information about the corresponding class, information about types of members to the type information about these members.

The functionality of this layer has still been a subject of discussion. In the following we give a brief overview over MOP layer 2:

Additional member functions accessing type information about classes:

const Meta classListIter(int &errflag, int start = 0) const;*

The function **classListIter(...)** iterates through all visible classes. "start" sets the iteration start at the beginning of the (internal) list of classes.

const Meta findClass(int &errflag, char* name) const;*

The function **findClass(...)** looks for type information about a class specified by its name.

int nbMembers(int &errflag) const;

The function **nbMembers(...)** returns the number of data- and function members of the class.

Member functions accessing type information about members, their types and friends:

The functions **name(...)** and **size(...)** are implemented in MOP layer 1, but they are applicable to members as well.

const Meta dataMemberIter(int &errflag, const DataProps* &dProps, int start = 0) const;*

The function **dataMemberIter(...)** iterates through the data members of the given class. "start" sets the iteration start at the beginning of the (internal) list data members for this class. The properties of a data member are given in the parameter dProps.

const Meta fctMemberIter(int &errflag, const FctProps* &fProps, int start = 0) const;*

fctMemberIter(...) iterates through the function members of the given class. "start" sets the iteration start at the beginning of the (internal) list of function members for this class. The properties of a function member are given in the parameter fProps.

int nbParams(int &errflag) const;

nbParams(...) returns the number of parameters of a function member. -1 indicates that the list contains the ellipse.

const Meta paramIter(int &errflag, const ParamProps* &pProps, int start = 0) const;*

paramIter(...) iterates through the parameters of a function. "start" sets the iteration start at the beginning of the (internal) list of parameters of the function. The properties of a parameter are given in the parameter pProps

const Meta friendIter(int &errflag, int start = 0) const;*

friendIter(...) iterates through all friends of a class. "start" sets the iteration start at the beginning of the (internal) list of friends.

int nbIndices(int &errflag) const;

nbIndices(...) returns the number of indices of an array.

int nbElems(int &errflag) const;

nbElems(...) returns the number of elements of an enum type.

const Meta enumIter(int &errflag, int start = 0) const;*

enumIter(...) iterates through all elements of an enum type. "start" sets the iteration start at the beginning of the (internal) list of the enum elements.

const Meta typeCompIter(int &errflag, int start = 0) const;*

typeCompIter(...) returns the address of the type information about type components of member-, typedef-, and function type-declarations. It iterates through all components of a type declaration. "start" sets the iteration start at the first component of a type declaration.

const char typeDecl(int &errflag) const;*

typeDecl(...) returns the type declaration of class-, struct-, union-, typedef-, enum- and data member declarations as well as the return type of function members and global functions. This method allows type identification and type comparison.

KindType kindOf(int &errflag) const;

kindOf(...) identifies the kind of the current type information, e.g. "classKind" for type information about classes.

2.4 Type information as a language extension

If type information at runtime becomes part of C++, the building of type information systems like MOP would be much easier. According to current experience with MOP we found three major items a standard type information system should guarantee.

- Standardized type information and functions to access this information for all types in C++, even for types declared in typedefs. The aspect of support for type information for all types is an open issue in our concept.

- Appropriate initialization of the type information by the compiler and the linker.

- A mechanism to add user defined type information to the standard information. The need for extensions of generated type information is recognized. It is planned to develop such a mechanism in an own step after realisation of the "basic" functionality of MOP.

We are very interested in an extension of C++ in this direction and we will cooperate with the according ANSI X3J16 working groups.

3 Open Issues

Some issues in the MOP approach for type identification require further discussion

- Support of exception handling

  Exception handling uses implicitly the type identification mechanism. Since a catch clause can catch a type which is one of the base classes of the thrown object, it is necessary that the exception handling mechanism has access to information about inheritance hierarchies. If the C++ exception handling mechanism should use the MOP, there will be a need for an integration concept to incorporate the MOP into the exception handling mechanism. Since it is allowed to throw away any type - not just classes - the MOP must be extended to support type inquiry for all types directly.

- Extension of the generated type information with application specific information

  It is sometimes necessary to extend the compiler or tool generated type information with application specific information. For example such information is required for a flexible object-oriented inter-process-communication. On the one hand the concept for extension of type information must be very flexible - it has to support both the addition of as well as the inquiry on every kind of information required by applications -, on the other hand it must be easy to use.

- Initialization of type information about global types

  The implementation of MOP layer 2 requires to store type information about other types than classes. Type information for classes is initialized statically through invoking the constructors of the class Meta directly. A problem is the initialization of type information about user defined global types which are used within classes, e.g. type information about types declared with the typedef construct of C++. The reason for this problem is that the Siemens approach to access type information is to extend classes, but types can not be extended.

  A possible solution to this problem is to introduce a member function for all classes called **initMeta()**. This member function initializes the type information for all types used within the class. Since the method is directly invoked at the beginning of an application's main routine, the correct initialization of global types is guaranteed. The disadvantage, however, is that the initMeta functions may do some redundant work. Consider the following example: While two classes are using the same global type, code for initialization of this type is implemented in the initMeta member functions of both classes. The initialization mechanism of the class Meta has to avoid that type information for this type is declared twice. Such a mechanism however decreases the performance of the initalization of type information.

- "Extern C" classes and classes without virtual functions

  Classes without virtual functions or with "extern C" declarations pose another problem. If the virtual function getMeta() is added to these classes (or structs) their storage layout will change. Currently C++-structs are handled as classes and C-structs as types.

- Interlinkage of information

  For MOP layer 2 we propose a class Harwich of Meta classes instead of a single class which implements the whole functionality of MOP. However, the problem is to interlink the different kinds of type information and to build up the internal structure of MOP. There are some possible solutions to this problem, but the functionality of MOP must be completed before it is possible to select an appropriate layout for the internal structure of MOP.

4 Integration of the Concept

There are two possible solutions for integrating the MOP concept to provide type identification and access to runtime type information for applications.

- Integration without language and compiler support (current state)
- Integration with language and compiler support (proposed by Siemens)

4.1 Integration of MOP without language and compiler support

Without any support of MOP through the C++ programming language and its compilers there is a need for a tool called **MOP-generator** to support the runtime type information system. Tasks of the MOP-generator to make available type identification and access to type information at runtime are:

- Extension of classes: According to the MOP concept described in chapter 2.1 a private static data member **classInf** and two member functions **getInfo()** and **getMeta()** must be implemented in each class.
- Generation of type information: The MOP-generator must add new code to the source files of applications which initializes the type information before starting the programs main routine
- Inclusion of the class Meta: The class Meta must be included into every source file of an application to have access to the member functions of MOP.

The MOP-generator has to be included into the translation process and is to invoke right after the C++ preprocessor.

The MOP-generator needs the .i(xx) files of the application. The MOP-generator includes a lexical analysis created with the UNIX™ standard program LEX and a parser for the C++ programming language created with the Berkeley YACC (using the C++ grammar published by James A. Roskind). Based on the lexical analysis and the parser the .i(xx) files are scanned and the actions enumerated above are processed.

4.2 Integration of MOP with support of language and compiler

By supporting of MOP (or parts of MOP) through the C++ programming language, the C++ compilers automatically generate and initialize type information for those parts of MOP that are supported by the C++ programming language. To make available the remaining parts of MOP a MOP-generator is necessary.

5 Previous Work

Many class libraries and toolkits provide their own concept of runtime type identification. Relevant for our work are the Dossier approach /5/ and the approach of Hewlett-Packard /3/, /4/, which are summarized in the following chapters. Other interesting ideas on runtime type information can be found in /7/.

5.1 Summary of Dossier Approach

In the Dossier approach /5/ which was developed for the InterViews toolkit, a tool called **mkdossier** scans the source files of an application and generates a statically initialized Dossier structure for each class in the user program. This Dossier structure is accessed through a virtual **GetClassId** function which needs to be added to each class. The Dossier approach proposes the following language extensions to simplify accessing the Dossier structure of classes:

- A predefined static member called **dossier** would be added to each class.
- The syntax **typename::dossier** would be used to access the dossier of a type
- The syntax **object.dossier** would be used to access the dossier of an object
- The syntax **pointer->dossier** would be used to access the dossier of the class object being pointed to. If the class being pointed to is polymorphic then the dossier of the class of which the object is an instance would be returned.

5.2 Summary of Hewlett Packard approach

The Hewlett Packard solution /3/, /4/ to the type identification problem is to introduce new built-in functions and a new built-in type **typeid** or a library class **TypeId** (this has still been an open issue) to support type identification uniformly for all types.

The built-in functions are:

- **ptr_cast** determines whether a dynamic object is subtype of a given type and if so casts down the pointer to that type
- **ref_cast** determines if the actual object referred by a reference is subtype of a given type and if so converts the reference to a reference of that type.
- **subtype** determines whether an object is subtype of a given type

The typeid type (or the TypeId class) includes two operations:

- **stype** returns the typeid value for a static type
- **dtype** returns the typeid value for a dynamic type

In addition a library class **TypeInfo** is introduced to provide further information about types. The access to this information is through the typeid of a given type. The TypeInfo class also provides functionality to extend the compiler generated type information with application specific information.

5.3 Comparison

Both the Dossier and the Hewlett Packard approach are similar to our approach with respect to the following aspects:

- All approaches provide a uniform access to type information about classes at runtime
- The approaches allow various kinds of functionality to be accessed by using member functions

Beside these similarities there are some fundamental differences between the Siemens approach and both the Dossier and the Hewlett Packard approach.

- Access to type information

  Both the Dossier and the Hewlett Packard approach make available a unique value associated with a type. This value allows to determine if two types are the same and to access further type information. Our approach of accessing type information follows more an object-oriented concept. Each class implements a static data member containing the type information about the class. The access to the type information is possible with two member functions which must be added to each class (see chapter 2.1). With this approach each object can be asked via the two member functions to give the type information about its class. Beside this direct access to type information about classes there is an indirect access to type information about data members, function members and their types. Access to type information about data- and function members is possible only through the type information about classes, to type information about types of data- and function members only through type information about these members.

- Support of type information about other types than classes

  The Dossier approach only supports inquiry for classes, the Hewlett Packard approach for all types. We support type information about classes, data- and function members of classes and their types. This requires to cover the whole type concept of C++. The difference to the Hewlett Packard approach is that the different type information items are not independent from each other. In our approach type information about members of classes is bound to the type information about the class, type information about types of members is bound to the type information about these members. For example, it is not possible to ask for type information about an integer directly, but through type information about a data member of type integer.

- Extension of classes

  In the Siemens approach the whole type identification mechanism and the access to this information is embedded in the library class Meta. The following extensions of classes are necessary to access the type information at runtime:

  - A static data member **classInf** must be available for each class which contains the type information about the class. This data member allows a direct access to the type information about the class within its member functions
  - Two member functions **getMeta()** and **getInfo()** must be available for each class to access the type information about the class from outside and to identify types.

  These extensions are only concerning classes and the behaviour of objects.

References

/1/ Margaret A. Ellis, Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison Wesley, 1990

/2/ Bjarne Stroustrup, Possible directions of C++, USENIX C++ Conference, 1987

/3/ Dimitry Lenkov, Michey Mehta, Shankar Unni, Type Identification in C++, USENIX C++ Conference, 1991

/4/ Dimitry Lenkov, Type Identification in C++, ANSI document, Lund, 1991

/5/ John A. Interrante, Mark A. Linton, Runtime Access to Type Information in C++, USENIX C++ Conference, 1990

/6/ Keith E. Gorlen, An Object-Oriented Class Library for C++ Programs, Proceedings of the USENIX C++ Workshop, 1987

/7/ Bjarne Stroustrup, The C++ Programming Language, 2nd edition, 1991

