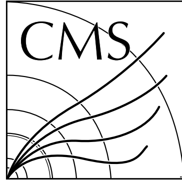
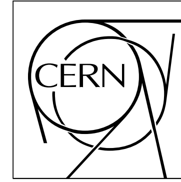


The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



13 February 2002

Evaluation of Oracle9i C++ Call Interface

Z. Xie, V. Innocente

CERN, Geneva, Switzerland

Abstract

This is a report on the evaluation of Oracle9i C++ Call Interface (OCCI) by CMS. We describe the evaluation of the usability of current version of OCCI by CMS from the software design point of view and the software quality of OCCI. This evaluation work is part of the evaluation of the Oracle9i database as an event store. We identify a missing middle-layer between OCCI and CMS Object-Oriented software. Two prototypes have been developed to demonstrate some of our design ideas of such a middle-layer.

1 Introduction

In July 2001, CMS decided to evaluate the Oracle9i database [1] as a candidate for its baseline for persistent data storage. This decision is mainly motivated by the worry of the market performance of the Objectivity/DB which is the current persistency baseline of CMS.

The Oracle database seems to be a good candidate for the CMS event store since it has a well-established software company behind and claims peta-byte data storage capability. What particularly interested us are the so-called object features of the Oracle9i database[2]. Oracle9i supports objects by mapping objects into rows in “object tables” where attributes of an object correspond to the columns of the table. A unique Object Identifier (OID) is assigned to each row of the object table. The navigation between objects is achieved by using the OIDs. Other object features include the support of data-type inheritance and the support of collection data-types as varrays.

These features put Oracle9i into the category of databases referred to as “object-relational”. But one has to be aware that although the Oracle9i database has some object-oriented features, it still belongs to the relational world. The “impedance mismatch” [3] becomes a problem when one chooses to traverse and access objects via their relationships as in the object world and to store data in tables as in the relational world. One of the main problems for CMS to use the Oracle database as an event store is how to glue seamlessly together the CMS object-oriented software with a fundamentally relational database. An ideal object-oriented persistence interface for a relational database should allow users to access and to navigate persistent objects transparently without being aware of the underlying tables.

Oracle9i provides a C++ Call Interface (OCCI) [4] as the communication layer between the C++ applications and the back-end database. We expected it to act as the layer between our object oriented software and the back-end Oracle database which allows transparent object access and navigation.

In the following sections we describe the evaluation of the usability of current version of OCCI by CMS from the software design point of view and the quality of OCCI as a software. Section 2 and 3 describe our evaluation of the usability of OCCI and its accompanying tool as the interface between the Oracle database and the CMS software. In Section 4 our prototypes of a middle-layer between OCCI and the applications are described. Section 5 and 6 are our evaluation of the software quality of OCCI. Section 7 gives the conclusion of this evaluation work.

It should be noted that for our evaluation we have used Oracle 9i version 9.0.1.0, 9.0.1.1 and 9.0.1.2 on Solaris. We also used a backport of the 9.2 development version to 9.0.1.2 that fixed some (but not all) memory leaks we experienced.

2 OCCI

Oracle C++ Call Interface (OCCI) is an application program interface (API) that provides C++ applications access to data in an Oracle database. It is a rather new product, first released in June 2001 with Oracle9i. OCCI supports both the associative and navigational style of data access. In associative access, data is manipulated by executing SQL statements and PL/SQL procedures. OCCI supports also the navigational access in which applications use references (REFs), which are essentially the OIDs of the target objects, to navigate through related objects. Navigational access does not involve executing SQL statements except to fetch the references of an initial set of objects. In our test, data is accessed using only the navigational approach.

OCCI manages also the client-side object cache which is allocated in the program’s process space. The object cache maintains the association between the object copy in the object cache and the corresponding database object. Upon *commit*, changes made to the object copy in the object cache are automatically propagated back to the database. The object cache maintains a pin count for each persistent object in the object cache. The pin count functions as a reference count for the object. The object becomes eligible for garbage collection when the pin count of the object becomes zero.

When using OCCI to manipulate objects, one should initialise the OCCI programming environment in object mode and then use the environment handle to establish a connection to the database server. There is one object cache allocated for each OCCI environment.

3 OTT and the problems of using OTT

When a C++ application retrieves instances of object types from the database, it needs to have a client-side representation of the objects. OCCI provides an abstract class *PObject* to be the base of the concrete representation of

the objects. The concrete representations of the objects are generated by the utility Object Type Translator(OTT).

To use OTT, one has to define the data model in the form of a SQL script and send the script both to the Oracle database to define the schema and to OTT. OTT then generates automatically C++ header (*.h*) and implementation (*.cpp*) files where all user defined data types are translated into C++ classes which are inherited from *PObject*. An example of the OTT generated header file is shown in Appendix A. An example application using OTT generated classes can be found in Appendix B.

The problems for us to use OTT are two-fold:

1. Data model drives object model

   As described above, OTT requires the data type definition, which is essentially one's data model design, as input. OTT assumes that the object model (if there is any object model at all) is based on the data model. But for our use case, we have an Object-Oriented software framework building upon our object model which models both data and behaviour. There's no reason for us to rebuild a data model from scratch and a object model based on it. Instead, we think that the data model should be somehow deduced or built from the existing object model. It's pretty clear that OTT has Data-Oriented application in C++ in mind, thus not suitable for the Object-Oriented way of software development of CMS.

2. Force changes in the existing software

   When one uses OTT, all user defined persistent classes have to inherit from the OTT generated classes which use *PObject* as the base class. This approach forces change in the existing software. For example, a tracker hit object *PSimHit*, instead of being a stand-alone class, should be changed to inherit from *PSimHit_C* where *PSimHit_C* is the OTT generated class. This change might involve few lines of code, but it would bound our software to a particular persistence mechanism, in this case Oracle. This is contrary to the current CMS policy of not binding physics and user code to a particular persistence mechanism.

From software design point of view, OCCI/OTT approach implies change in the modelling of our software and restricts the flexibility and reusability of the applications. We anticipate major difficulties in integrating such an approach in an Object-Oriented software framework like COBRA[5].

4 Prototyping a middle-layer between OCCI and applications

As described in the previous section, OTT is not suitable for our use. So two questions arise immediately:

1. How will the client-side representation of the objects be implemented without changing existing classes and without using OTT?

2. How can one build or deduce the data model from the existing object model?

4.1 First prototype

Immediate and apparent answers to the above questions are:

1. Without OTT, the persistent object classes have to be implemented by the user by hand. A persistent object class should be a *PObject* and a CMS object at the same time. Practically this means the persistent object is multiple inherited from *PObject* and the CMS transient class. For example, the persistent class of the *PSimHit* object is *PSimHit_C* which has both *PObject* and *PSimHit* as base classes. And *PSimHit_C* is user-implemented.

   By using multiple inheritance, the existing CMS class is not touched while the behaviour of the transient object is conserved.

2. The straightforward way to extract data information from the existing software is to look into the header(*.h*) files. For instance, by looking into the header file *PSimHit.h* one knows that the object *PSimHit* has eight data members and two of them are objects of type *Local3DPoint*. Then the data structure of the *Local3DPoint* datatype can be found in header file *Local3DPoint.h*.

These are the basic ideas behind our first prototype. Furthermore, we also want to see if Oracle related software is potentially easy to be integrated into existing CMS software. There are mainly two problems. As shown in first lines of the code listed in Appendix B, when building the application directly on OCCI, all the persistent objects should be registered to a hash map *oracle::occi::Map* for one Oracle session. This requires the application to know and to register explicitly in advance all the possible persistent objects. It's impossible to integrate such kind of applications into a generic software framework like COBRA. Besides, OCCI functions and calls with heavy SQL flavour are not hidden at all from the user.

Our solutions to these problems are the following. To avoid registering all objects in one go and from one place, each persistent object has an registration agent, e.g. *PSimHit_Regist*. Like *PSimHit_C*, it's completely independent. A helper class *PObjRegister* registers a persistent object only when required. To encapsulate common OCCI or SQL style functions, three other classes have been developed. *PManager* handles general services as open/close Oracle connection, commit, etc and it collaborates with *PObjRegister* for object type registration. *PReader* is an iterator-like template class which handles retrieving data from Oracle. Similarly, *PWriter* handles saving data into Oracle.

In summary, it is very difficult to integrate applications built directly on OCCI as such into a generic software framework like COBRA. To make it possible, some kind of middle-layer software has to be developed. On one hand, persistent object related classes should be independent so that anyone from any level of the CMS software can use them and/or create new ones; on the other hand, common Oracle services and SQL commands should be encapsulated into common software.

An independent module "Oracle/RDBPopulator" has been implemented and called from the main application inside COBRA. The modified COBRA application populates detector hits into both Objectivity database (as it does before) and the Oracle database. No other CMS code is modified. From user application level, i.e. from ORCA (the CMS reconstruction software)[6], users are not aware of the change.

An example of user application building on these middle-layer classes can be found in Appendix C. As we can see from the example, comparing to the applications building on the OTT generated classes, the chunk of code handling the object registration disappeared. Instead, the registration task is handed to the independent classes like *PSimHit_Regist*, *Local3DPoint_Regist* etc. So user can register objects from any level of the software framework. This also makes software packaging easier. Besides, heavy SQL flavoured calls like *setSQL* is now hidden from the user. One just passes an instance of a transient object to its corresponding persistent class to make it eligible for database operations. The application is much simpler to understand and to write.

A more detailed description of the implementation of this prototype can be found in the Reference [7].

4.2 Second prototype

Though the first prototype basically solved our problems, it requires quite a lot of manual work by users. First of all, one has to go through all the related header files to figure out a data model. This can be frustrating especially when all the header files are scattered in different directories. Then user has to implement the persistent classes, like *PSimHit_C*, by hand even though major part of these code are repeated each time.

Not completely satisfied by the first prototype, we ask ourself again the previous two questions, and we come up with the following new answers:

1. The concrete persistent object class should be implemented by the compiler at the compilation time. Similar and/or repeated implementations should be generalised and be put in a class library.
2. By using a C++ parser, one can extract data information from existing header (*.h*) files.

In this prototype, there are two distinct parts. On one hand, there's a C++ parser that takes an object header (*.h*) file, say *PSimHit.h*, as input then analyses and traverses the resulting Abstract Syntax Tree (AST) to generate all the related data type definitions. User can send these data type definitions to the Oracle database to define the schema.

On the other hand, the middle-layer is enriched by three template classes. *PObjHandle* is a template class that encapsulates most of the implementations of a persistent object (e.g. *PSimHit_C*) user had to write by hand with the first prototype. By using this template class, we delegate most of the task of implementing a persistent object to the compiler. But user still has to do something: OCCI wants to know what to stream and of course the template

class *PObjHandle* does not know this. To make life easier for users, our middle-layer provides two other template classes *StreamIn* and *StreamOut*. They are generic functor objects that behave like functions and accept any number and any type of arguments. Now, what remains for user to do is very simple: for each persistent object, one has to implement a class with three member functions: a constructor that copies the transient object, a member function that tells the generic streamer what to stream in and a similar member function that tells the generic streamer what to stream out. An example of the user implemented class is listed in Appendix D.

The code in the application appears similar to that with the first prototype except one has to use *PObjHandle* *<PSimHit_C>* instead of *PSimHit_C* directly.

A more detailed description of this prototype can be found in reference [8].

4.3 Prospects

With respect to the first prototype, much less manual work is needed by user in the second prototype. Although in the second prototype the code generation is more automatic, we are still not completely satisfied. Once one is able to extract all the type information of the class hierarchy from the C++ parser, there should be no need for the user to instruct the generic streamer with these information again. An ideal generic streamer should be able to look for the required information in a data dictionary and no intervention from user should be required. What stored in the data dictionary is the data information which can be extracted from a C++ parser, a compiler or even a decompiler.

How to implement this kind of intelligent generic streamer strongly depends on how the data dictionary is constructed and what information it contains. Thus the design and the implementation of an intelligent generic streamer is beyond the scope of this note. We are aware that, at this very moment, the RTAG1(Persistency Framework) of the LHC Computing Grid Project (LCG)[9] has been asked to construct a component breakdown for a persistency framework of which a data dictionary is clearly a part. We plan to follow closely its work and the activity that will follow that should bring to the production of a common LHC persistency framework. We are definitively interested in the possibility to reuse its data dictionary, and eventually other components, to build the required middle-layer above ORACLE OCCI.

5 Software quality of OCCI

During the test of OCCI using our model only, we have encountered several bugs. One is the memory leak in the object cache. Another problem deals with reading an object from the database when this object has another object as its member. These bugs are fixed by Oracle developers eventually. But one has to follow the Oracle software development cycle which might take one month to get back a bug fix.

When using OCCI to write applications, one also has to pay attention to some details like to *unpin* object by hand to avoid memory leak, to deliberately add a scope to avoid program crash. We don't think it should be the application programmer's responsibility to do such kind of work.

Therefore, we don't judge the current version of OCCI (9.0.1.2) as a mature bug-free software.

6 First attempt to benchmark the OCCI performance

A speed performance test of OCCI has been setup and we obtained some preliminary results. But these numbers are not stable since we don't have an isolated and controlled test environment. We did the test using the IT/DB machine where client and server are running on the same machine and there are also other applications running on that machine.

7 Conclusions

For the time being, we consider that OCCI is not completely bug free and stable enough for us to make any benchmark tests. Nevertheless, we are ready in benchmarking OCCI using the test programs we have developed. From the software engineering point of view, OCCI is too thin to be a stand-alone layer between the Oracle database and CMS object-oriented software framework. Efforts are needed to build a middle-layer between OCCI and CMS software. Our simple prototype shows some of our design ideas about the middle-layer.

From the evaluation work we gained more experiences in software design and generic programming which are very useful in any software development. Now we have better understanding of the object-relational database and

we have a clearer view of what an interface to an object-relation database would/should look like. This is useful experience in design any persistency framework that glues Object-Oriented applications with an object-relational database back-end. First of all, we think the object model should drive the data model in designing such an interface. This interface should be as generic as possible. Common services should be centralised and be encapsulated in generic and reusable classes while concrete object level classes should be decoupled and delocalized to make packaging of the applications possible.

We are looking forward to the result of the LCG common project on the persistency framework. We hope to be able to achieve a seamless integration between the product of such a project and OCCI, along the line described in this note.

8 Acknowledgement

We wish to thank D. Duellmann and D. Geppert for their invaluable help in understanding OCCI, and the Oracle9i database system in general, and for reporting our problems to Oracle Corporation. We are grateful to IT/DB group for allowing us to do all the tests using their Oracle9i database installation.

References

[1] Lenore McGee Luscher *et al.*, Oracle Corporation, **Oracle9i Database Concepts**

[2] Bill Gietz *et al.*, Oracle Corporation, **Oracle9i Application Developer's Guide: Object-Relational Features**

[3] Scott W. Ambler, Ronin International, **Mapping Objects to relational databases**

[4] D. Raphaely *et al.*, Oracle Corporation, **Oracle C++ Call Interface: Programmer's Guide**

[5] on-line manual of COBRA, http://cobra.web.cern.ch/cobra/

[6] on-line manual of ORCA, http://cmsdoc.cern.ch/orca/

[7] Z.Xie, **Oracle9i evaluation: Project Advancement Report (Part 1)**, http://cmsdoc.cern.ch/cmsoo/oracle9i/progress1.html.

[8] Z.Xie, talk at the HEP database workshop, Jan 29-31, 2002, http://hep-proj-database.web.cern.ch/hep-proj-database/workshop-Jan2002/Zhen-workshopOCCI.ppt

[9] LCG homepage, http://lhcgrid.web.cern.ch/LHCgrid/LW2002/

Appendix A: An example OTT generated C++ header file

```
class Local3DPoint_C : public oracle::occi::PObject {

protected:

   oracle::occi::Number theX;
   oracle::occi::Number theY;
   oracle::occi::Number theZ;
public:

   void *operator new(size_t size, void *ctxOCCI_);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   Local3DPoint_C();

   Local3DPoint_C(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};
```

Appendix B: An example of application using OCCI/OTT

```
void psimhit_map(oracle::occi::Environment* envOCCI_)
{
  oracle::occi::Map *mapOCCI_ = envOCCI_->getMap();
  mapOCCI_->put("USER.LOCAL3DPOINT_O", Local3DPoint_C::readSQL,
                Local3DPoint_C::writeSQL);
  mapOCCI_->put("USER.PSIMHIT_O", PSimHit_C::readSQL, PSimHit_C::writeSQL);
  mapOCCI_->put("USER.OVPSIMHIT_O", ovPSimHit_C::readSQL,
                ovPSimHit_C::writeSQL);
  mapOCCI_->put("USER.OVREFPSIMHIT_O", ovrefPSimHit_C::readSQL,
                ovrefPSimHit_C::writeSQL);
}

void write(Connection *conn)
{
   int nevt=100000;
   int nhits=10;

   for(int j=0;j<nevt;j++){
      OCCI_STD_NAMESPACE::vector<PSimHit*> hitcoll;
      for(int i=0;i<nhits;i++){
 Local3DPoint* ip=new Local3DPoint(1,1,1);
 Local3DPoint* op=new Local3DPoint(-1,-1,-1);
 PSimHit* hit=new(conn,"PSIMHIT_TAB") PSimHit(ip,op,1,1,1,1,1,i);
 hit->unpin();
 hitcoll.push_back(hit);
      }
      conn->commit();
      hitcoll.clear();
   }
}

void read(Connection *conn)
{
  Statement *stmt = conn->createStatement();

  stmt->setSQL("Select ref(t) from PSIMHIT_TAB t");
  ResultSet *resultSet = stmt->executeQuery();
  while (resultSet->next() == ResultSet::DATA_AVAILABLE) {
    Ref<PSimHit>hit_ref = resultSet->getRef(1);
    PSimHit          *PSimHit_ptr;
    cout << "Hit id=" << int(hit_ref->trackId()) << endl;
  }

  stmt->closeResultSet(resultSet);
  conn->terminateStatement(stmt);
}


int main(int argc, char** argv)
{
   if(argc<2) {
      cout << "arguments:  w:write, r:read" << endl;
      exit 1;
   }

  Environment *env = Environment::createEnvironment(Environment::OBJECT);
```

```
  psimhit_map(env);
  Connection *conn = env->createConnection("user","passwd","");

  try{
     switch(argv[1][0]) {
      case 'w' :
 write(conn);  break;
      case 'r' :
 read(conn); break;
      default :
 cout << "wrong argument" << endl;  break;
     }
  } catch (oracle::occi::SQLException &e) {
     cerr << "SQL exception :" << e.getMessage() << endl;
  }
  env->terminateConnection(conn);
  Environment::terminateEnvironment(env);
  return 0;
}
```

Appendix C: An example of application using the first prototype

```
class hitwrite{
public:
  hitwrite(const PManager& mymanager,int nevt=20)
  {
    int nhits=10;
    for(int j=0;j<nevt;j++){
      vector< Ref< PSimHit_C> > hitcoll;
      for(int i=0;i<nhits;i++){
Local3DPoint ip(1,1,1); Local3DPoint op(-1.5,-1.6,-1.7);
PSimHit myhit(ip,op,1,1,1,1,1,i);
PSimHit_C* hit=save<PSimHit_C,PSimHit>(mymanager, "PSIMHIT_TAB",myhit);
hitcoll.push_back(hit);
      }

      ovPSimHit_C* hitcoll_c=save<ovPSimHit_C,ovPSimHit>(mymanager,
                             "OVPSIMHIT_TAB",hitcoll);
      hitcoll.clear();
      mymanager.commit();
    }
  }
};

class hitread{
public:
  hitread(const PManager& mymanager){
    Ref< ovPSimHit_C > hitcoll_ref;
    vector< Ref < PSimHit_C > >* v;
    PReader< ovPSimHit_C > red(mymanager,"OVPSIMHIT_TAB");
    while (red.next()){
      hitcoll_ref=red.getObj();
      v=&(hitcoll_ref->getv());
      int vsize = v->size();
      for (int i=0; i<vsize; i++) {
cout << "Hit id=" << (*v)[i]->trackId() << endl;
      }
    }

    Ref < PSimHit_C > hit;
    PReader< PSimHit_C > redhit(mymanager,"PSIMHIT_TAB");
    while (redhit.next()){
      hit=redhit.getObj();
      cout << "Hit id=" << hit->trackId()<< endl;
    }
  }
};

int main(int argc, char** argv)
{
  if(argc<2) {
    cout << "arguments:  w:write, r:read" << endl;
    exit(1);
  }

  PManager mymanager("user","passwd");
  PSimHit_Regist a(mymanager);
```

```
  Local3DPoint_Regist b(mymanager);
  ovPSimHit_Regist c(mymanager) ;
  mymanager.open();

  try{
    if (argv[1][0] == '-'){
      switch(argv[1][1])
{
case 'w' :
  {
    hitwrite t(mymanager);
    break;
  }
case 'r' :
  {
    hitread t(mymanager);
    break;
}
default :
  { cout << "wrong argument" << endl;  break;}
}
    }else{cout << "wrong argument" << endl;}
  } catch (oracle::occi::SQLException &e) {
    cerr << "SQL exception :" << e.getMessage() << endl;
    exit(-1);
  }

  mymanager.close();
  return 0;
}
```

Appendix D: An example of user implemented class using in the second prototype

```
#define private public
class Local3DPoint_C{
 public:

  Local3DPoint_C(Local3DPoint& t):proxy_(t){
  }

  void writeSQL(occi::AnyData& stream){
    StreamOut<float,float,float> st;
    st(stream,proxy_.x_,proxy_.y_,proxy_.z_);
  }

  void readSQL(occi:: AnyData& stream){
    StreamIn<float,float,float> st;
    st(stream,proxy_.x_,proxy_.y_,proxy_.z_);
  }

  operator const Local3DPoint & () const { return proxy_;}

 private:
  Local3DPoint & proxy_;
};
typedef PObjHandle<Local3DPoint_C> Local3DPointHandle;
#endif
```

The *#define private public* is required to get access to the data members of *Local3DPoint*.

The implementation of the *proxy* (pure reference, reference-counted, etc) will depend on how the user instances will be actually managed.