EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/PS/ 90-007 (CO)
15 March 1990

Project: NAPS
Domain: DATABASE
Category: PAPER
Status: FINAL

OBJECT ORIENTED PROGRAMMING WITH A RELATIONAL DATABASE

Jan Cuperus, Wolfgang Heinze, Claude-Henri Sicard

Abstract

The particle accelerators of the CERN proton synchrotron complex consist of thousands of pieces of very diverse equipment which must be controlled in a uniform way via software interface modules residing in some 50 micro-computers. These modules belong to classes which are described in ORACLE tables and updated through a form. A full heritage mechanism is implemented. A second form permits instanciation of members of the class and attribution of values to the instance variables. At any time, the data tables for a given processor can be generated. These tables, together with a fixed frame and a library of methods, form complete modules for accessing the equipment. Documentation of a class and its instanciations can be generated on request.

To be presented at the 8th ORACLE Users Conference, Madrid, April 1990.

Geneva, Switzerland

OBJECT ORIENTED PROGRAMMING WITH A RELATIONAL DATABASE

Jan Cupérus, Wolfgang Heinze, Claude-Henri Sicard
CERN, European Organisation for Nuclear Research
1211 Genève 23, Switzerland

Summary

The particle accelerators of the CERN proton synchrotron complex consist of thousands of pieces of very diverse equipment which must be controlled in a uniform way via software interface modules residing in some 50 micro-computers. These modules belong to classes which are described in ORACLE tables and updated through a form. A full heritage mechanism is implemented. A second form permits instanciation of members of the class and attribution of values to the instance variables. At any time, the data tables for a given processor can be generated. These tables, together with a fixed frame and a library of methods, form complete modules for accessing the equipment. Documentation of a class and its instanciations can be generated on request.

Introduction

The CERN proton synchrotron complex consists of ten interconnected particle accelerators built over a period of some 30 years. It contains very diverse equipment which somehow must be controlled so that the accelerators can work together. To make this possible, there are several software and hardware layers. Control data go down to the equipment and information about the accelerator goes up to the operator:

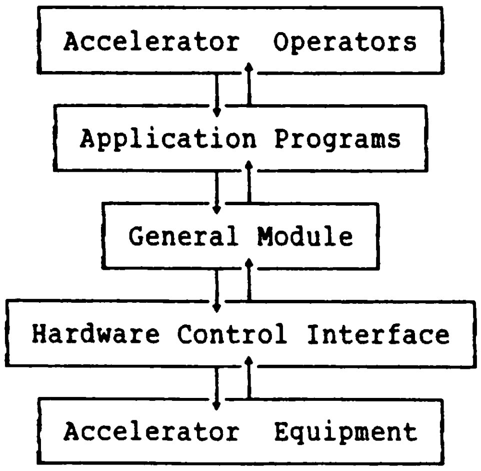


Fig.1: Software and hardware layers for accelerator control.

Application programs (AP) can autonomously control parts of the accelerator and display or log information but usually they present a high level interactive interface to the operator.

The hardware control interface is bus-oriented (VME, CAMAC etc ...) and translates computer words into contacts and levels which control the equipment and the other way round for status and measurement information.

In order to hide the irrelevant details of the equipment interface from the application programs, the General Module (GM) is built between the two. We started writing equipment interface modules 12 years ago by editing frames with a text processor and initialising variables from a file. This worked but it was difficult to modify the module afterwards and there was no proper documentation of the finished module. A few years ago we decided to build the modules with the help of a relational database. The remainder of this paper will be about the GM and how it is constructed from information in the database.

Equipment Classes

Power supplies for magnets, timing-pulse generators, vacuum pumps, and beam position detectors are obviously quite different devices and we would not expect them to react in the same way. On the other hand, power supplies of various types and manufacturers may present differences which are quite irrelevant to their task of setting a voltage or a current. Devices which we want to look similar from the higher levels are grouped in a CLASS. Common features shared by several classes are expressed in a SUPERCLASS and included in the class through an heritage mechanism. Starting from the general GM class, we can construct a whole hierarchy of classes:

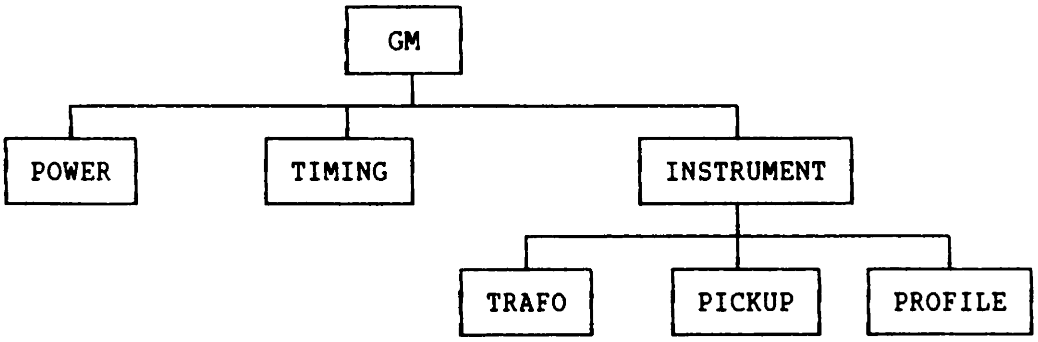


Fig.2 : An example of a class hierarchy

We have some 5000 pieces of equipment, grouped in about 100 classes. Some classes, such as POWER and TIMING have hundreds of members while some very specialised measurement aparatus are unique in their class.

Calling Sequence for the GM

The calling sequence is object oriented but, for the moment, implemented in normal C language:

```
ERRNO = GM(CLASS, MEMBER, SELECTOR, SVALUE)
```

MEMBER is a number identifying the piece of equipment in the class, SELECTOR is a symbol identifying the action to be taken and SVALUE is a pointer to a data structure which corresponds to the selector. SVALUE is written to the GM when the selector is a command (e.g. WRCV for write-control-value) and read from the GM when the selector is a request for information (e.g. STAQ for status-acquisition). ERRNO is 0 when the communication with the GM and the hardware control interface (usually over the computer network) was OK and returns an error number when not. It does not indicate whether the equipment is working correctly: for this you must read the status or check the equipment settings.

Structure of the GM

The GM consists of 4 parts:

(1) A frame which is an identical piece of code for all the classes. It checks the calling arguments and implements the selector mechanism.

(2) A data table, specific for the class, which contains the variables used by the class. Variables can be class-variables, which are defined once for the whole class, and instance-variables which have a different value for each member of the class.

(3) A selector table with records containing the calling sequence for the method which implements the action requested by the selector.

(4) A method library which contains 'standard' methods, used by many classes and some methods which are specific to one or a few classes. Re-use of methods is frequent because they can be parametricized.

The data table and the selector tables are derived from ORACLE tables.

Variable definition and the inheritance mechanism

The class and instance variables are defined in table VARDEF :

```
VARDEF= (CLASS +      * name of the class *
         VARNAME +    * name of the variable *
         CI +         * C or I, for Class- or Instance-variable *
         VARACCESS +  * RO or RW for read-only or read-write *
         VARTYPE +    * R,I,R(12),I(5),S(16): real,int,string (array) *
         VARDESCRIP   * short description of function of variable *    )
```

Table VARDEF lists only the variables defined for a particular class but says nothing about inherited variables. Table GMCLASSES :

```
GMCLASSES= ( CLASS + SUPERCLASS + ...  )
```

gives the immediate superclass of the class but this superclass can have a superclass of its own, and so on. The superclass hierarchy of the class can be obtained with a tree-oriented search. The complete set of variables for a given class (denoted by variable :XCLASS) is then obtained with :

```
SELECT   CLASS,VARNAME,CI,VARACCESS,VARTYPE,VARDESCRIP
FROM     VARDEF, COUNT
WHERE    (CLASS, COUNT) IN (
            SELECT  CLASS, LEVEL                          ┐
            FROM    GMCLASSES                             │(1)
            CONNECT BY PRIOR SUPERCLASS = CLASS           │
            START   WITH CLASS = :XCLASS) AND             ┘
         (VARNAME, COUNT) IN (                                 ┐
            SELECT  V.VARNAME, MIN(COUNT)                      │(2)
            FROM    VARDEF V, COUNT, GMCLASSES G               │
            WHERE   G.CLASSNAME = V.CLASSNAME AND              │
                    (V.CLASSNAME, COUNT) IN (                  │
                       SELECT  CLASS, LEVEL               ┐    │
                       FROM    GMCLASSES                  │(1) │
                       CONNECT BY PRIOR SUPERCLASS = CLASS │    │
                       START   WITH CLASS = :XCLASS)      ┘    │
            GROUP   BY V.VARNAME) AND                          ┘
         NOT VARTYPE = 'X';
```

Table COUNT = {COUNT} contains the integers from 1 to 10;
(1) selects the set of the class and superclasses of the class;
(2) selects the name and lowest level in the superclass hierarchy for each variable defined in one of the classes of the set;

Finally, we select the attributes for these variables which are not suppressed by having VARTYPE = 'X' in the lowest definition. This whole construct allows us to inherit variables but also to add, change or suppress variable definitions in the lower level classes.

The expressive power of SQL for solving such kinds of problems seems to be a bit wanting and an auxiliary table COUNT was used to get around the limitations. Also, the execution speed of logically equivalent formulations can vary wildly but the variant used here proved to be quite fast.

Note that we have only defined the variables but we have not given them values. Depending on VARACCESS, variables can be (1) Read-only: in fact constants containing the fixed data of the equipment such as addresses and scaling factors and (2) Read-write: operational values such as the voltage of a power supply. The RO data are contained in an ORACLE table INSTVAL and downloaded into the GM data table when the module is generated. The RW data are set by the operators or loaded from archive files to bring the machine in a well defined state. In the future, we intend to replace these archive files by ORACLE tables. RW data can also contain results of acquisitions from the equipment, such as status, measured voltage, or results of beam measurements.

The selector definition table

The SELECTOR is a pointer in a table which tells the GM which METHOD to use and with what arguments in order to get the desired action. Each class can inherit selectors from its superclasses and define its own. The selectors are defined in ORACLE table SELECTORDEF :

```
SELECTORDEF= {CLASS +     * name of the class *
              SELECTOR +  * name of the selector *
              RW +        * RO, WO, RW for read/write action on SVALUE *
              METHOD +    * the name of the procedure to be called *
              PARLIST +   * method arguments, separated by commas *
              SELDESCRIP  * short description of selector action *     }
```

The arguments are literals or names of defined variables. The SVALUE argument is always implicitly transmitted to the method. The attributes of all selectors for a class, including the inherited ones, are obtained with a SQL statement similar to the one for variables in previous section.

The Methods

METHODS do most of the actual work. They are listed in table METHODS :

```
METHODS= {METHOD +      * the name of the method *
          SVALTYPE +    * the type of SVALUE *
          AUTHOR +      * the name of the author of the method code *
          LASTUPDATE +  * when method was last updated *
          METDESCRIP +  * short description of function of the method *
          SOURCE +      * file name for the source code *
          LIBRARY       * name of the library file with object code *  }
```

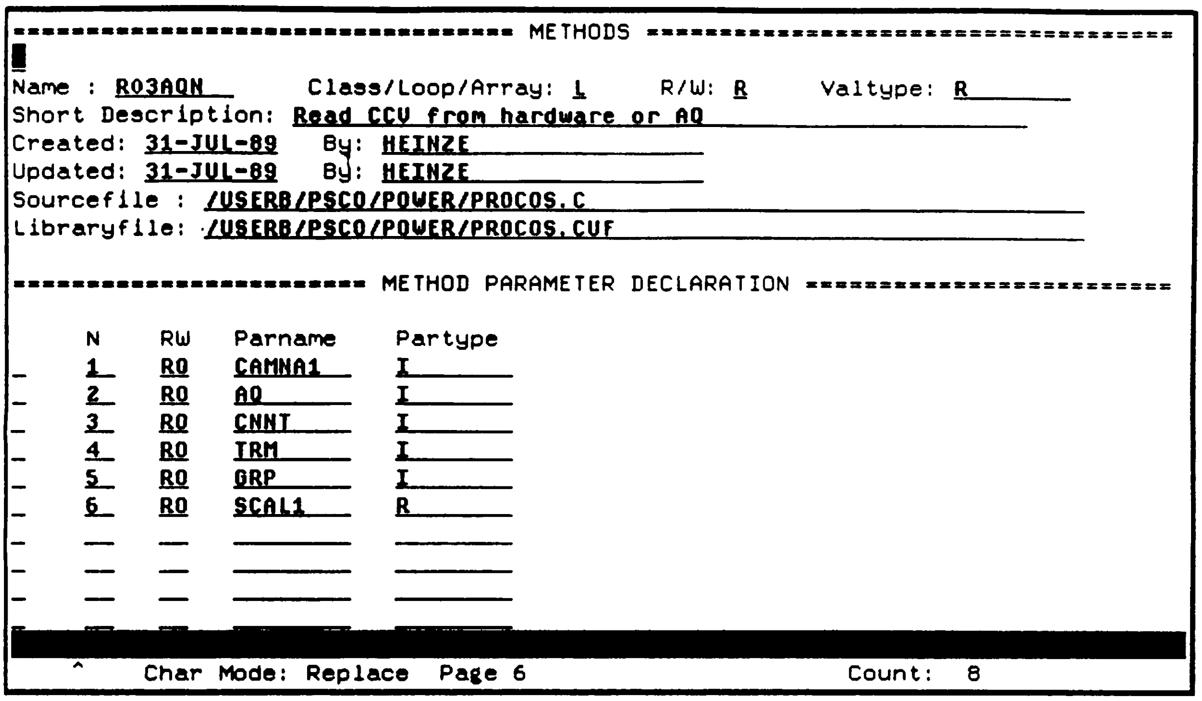
Associated dummy arguments are defined in table METHODPARAMS :

```
METHODPARAMS= {METHOD +    * the name of the method *
               SEQNO +     * the sequence number of the argument *
               PARNAME +   * the name of the dummy argument *
               ACCESS +    * RO, WO, RW for read/write access *
               PARDESCRIP  * short description of argument function *  }
```

For the moment, the source code for the methods is on separate files. A logical development would be to include the source code text in the database which is now possible with the available text handling database extensions.

Defining the class with GMNEW

GMNEW is a form for creating or modifying a class. There are several blocks for specifying general information, class variables, instance variables, selector branching with actual arguments, methods with dummy arguments. Fig. 3 shows one of the blocks.



```
================================= METHODS ==================================

Name : R03AQN       Class/Loop/Array: L     R/W: R     Valtype: R
Short Description: Read CCV from hardware or AQ
Created: 31-JUL-89    By: HEINZE
Updated: 31-JUL-89    By: HEINZE
Sourcefile : /USERB/PSCO/POWER/PROCOS.C
Libraryfile: /USERB/PSCO/POWER/PROCOS.CUF

====================== METHOD PARAMETER DECLARATION =======================

        N    RW    Parname     Partype
_       1    RO    CAMNA1      I
_       2    RO    AQ          I
_       3    RO    CNNT        I
_       4    RO    TRM         I
_       5    RO    GRP         I
_       6    RO    SCAL1       R
_       __   __    ________    ________
_       __   __    ________    ________
_       __   __    ________    ________

   ^     Char Mode: Replace  Page 6                     Count:  8
```

Fig.3: method definition block and associated argument declaration block in master-slave relationship.

Filling in the data with GMFILL

With GMNEW, the class is defined but there are as yet no members. Members are created by filling a table EQUIPMENT :

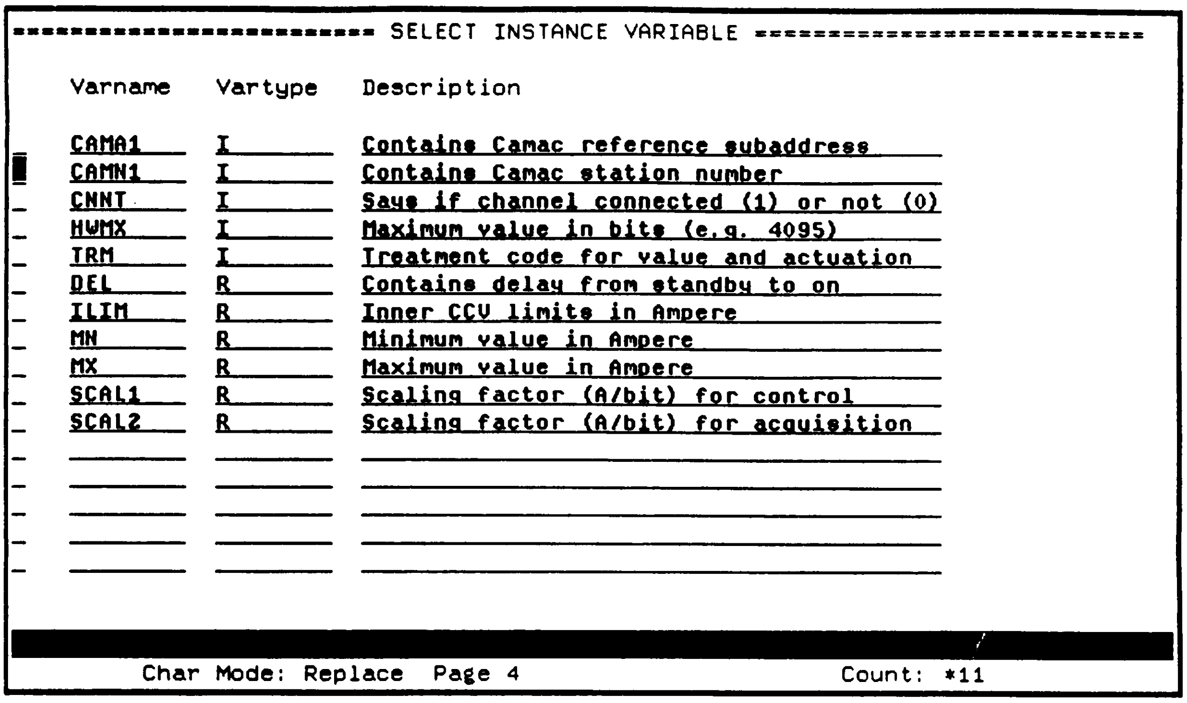
```
EQUIPMENT= {CLASS +       * the name of the class *
           MEMBER +       * member identification number in the class *
           EQNAME +       * member name *
           COMPNAME +     * the computer where the GM will run *
           EQDESCRIP +    * short description of the member *
           ...            * other information about the member *         }
```

When the members and the variables are defined, we can attribute values to the RO variables with form GMFILL. The first block of this form asks for the CLASS, the first and last member you want to update, and your initials. Pushing NEXT-BLOCK will now cause any missing RO variables to be created in table INSTVAL :

```
INSTVAL= { CLASS + MEMBER + VARNAME + VALUE + INITIALS + LASTUPDATE }
```

SQL*FORMS does not allow you to do this creation in a straightforward way, so we update a dummy table and implement the creation in a PRE-UPDATE trigger.

You have now the choice of updating all variables for a selected member or a selected variable for the range of members, whichever is more practical. Value is entered as a character string. Arrays are entered as numbers, separated by commas. Fig.4 shows two pages of the form.

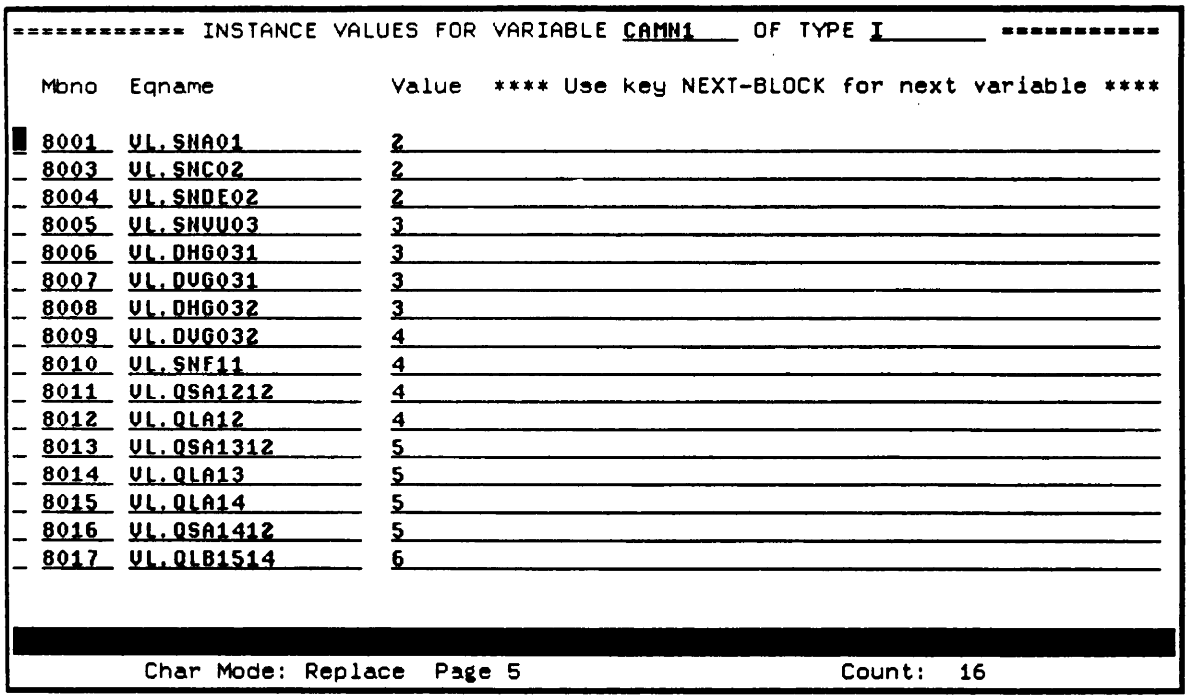


```
======================== SELECT INSTANCE VARIABLE ==========================

    Varname     Vartype     Description

_   CAMA1       I           Contains Camac reference subaddress
█   CAMN1       I           Contains Camac station number
_   CNNT        I           Says if channel connected (1) or not (0)
_   HWMX        I           Maximum value in bits (e.g. 4095)
_   TRM         I           Treatment code for value and actuation
_   DEL         R           Contains delay from standby to on
_   ILIM        R           Inner CCV limits in Ampere
_   MN          R           Minimum value in Ampere
_   MX          R           Maximum value in Ampere
_   SCAL1       R           Scaling factor (A/bit) for control
_   SCAL2       R           Scaling factor (A/bit) for acquisition
_
_
_
_
_

          Char Mode: Replace   Page 4                    Count: *11
```



```
============ INSTANCE VALUES FOR VARIABLE CAMN1   OF TYPE I        ===========

  Mbno   Eqname            Value   **** Use key NEXT-BLOCK for next variable ****

█ 8001   VL.SNA01          2
_ 8003   VL.SNC02          2
_ 8004   VL.SNDE02         2
_ 8005   VL.SNVU03         3
_ 8006   VL.DHG031         3
_ 8007   VL.DVG031         3
_ 8008   VL.DHG032         3
_ 8009   VL.DVG032         4
_ 8010   VL.SNF11          4
_ 8011   VL.QSA1212        4
_ 8012   VL.QLA12          4
_ 8013   VL.QSA1312        5
_ 8014   VL.QLA13          5
_ 8015   VL.QLA14          5
_ 8016   VL.QSA1412        5
_ 8017   VL.QLB1514        6

          Char Mode: Replace   Page 5                    Count:  16
```

Fig.4: Two blocks of form GMFILL. The upper one allows you to select a RO instance variable and the next block allows you to fill in the values for this variable for a range of equipment.

Class variables have only one value for the whole class. They are entered in table INSTVAL with the fictive member number 0.

Generating the GM tables with MODULGEN

The program MODULGEN asks for the name of the target processor and then constructs the GM tables for all the classes and members implemented on that processor. The result is a set of tables in source code of the C language. A second program then compiles the tables, links them with the frame object code and the required method libraries and loads the resulting object on the target processor.

For constructing the tables from the information in the database, text strings must be converted into values and symbols must be converted into symbolic values with the help of appropriate symbol tables in the database. Inconsistencies must be clearly reported so that they can be easily corrected.

Generating the dispatcher tables

The modules are installed in remote microprocessors while the application programs run in workstations, and the whole is interconnected with Ethernet. In each workstation, there must be an equipment table and a dispatcher which routes calls to the appropriate computer. A dispatcher call is identical to a direct module call but there are extra features: it is possible to call a piece of equipment by name instead of by class + number and it is also possible to call an array of members (with a corresponding array of values) for the same class. The dispatcher cuts up the array in array calls to the appropriate computers and re-assembles the responses. The dispatcher tables are identical in each workstation and are acquired when the workstation is initialised or whenever an update is necessary. They are derived from table EQUIPMENT.

Generating the documentation with MODULDOC

Program MODULDOC asks for the classname and then generates the documentation in SGML format from information in the database. A header with general documentation is generated, followed by a plain text description of the module. At present, the database refers to a file where the description is kept but this description should be moved to the database itself. Then follows the description of the class tables (class and instance variables, selector branching and method attributes). Here, use is made of the description fields included in almost every table. After the class description the members are listed with the values of all RO instance variables.

Special features

Our accelerators work in cycles, lasting one to several seconds. In a cycle, some accelerators can accelerate electrons, positrons, protons, antiprotons or heavy ions and this for several possible users. Cycles of different kinds can follow each other in supercycles and, for each cycle, thousands of parameters must be set to different values. This is done by defining variables of type PPM (pulse-to-pulse-modulation) : each variable is in fact an array of 8 variables and a real-time program reads the tables in the GM between cycles and sets the appropriate values for that cycle. The operator can independently control each of 8 virtual accelerators independently of the others. For this an additional parameter indicating the virtual machine is included in the calling sequence.

In this paper, we have used the proper object-oriented terminology. For historical reasons, this differs in some points from the terminology we actually use. The form examples given here have been slightly adapted so as not to confuse the reader.

Object-oriented databases

Future developments of database management systems may go in the direction of semantic databases which impose integrity rules on top of the relational model, and object-oriented databases which impose structures on the data and define exactly what can be done with these structures. Roughly speaking, in semantic databases, what is not forbidden is allowed and, in object-oriented databases, what is not allowed is forbidden. Semantic databases are probably preferable as a versatile central depository of data, but structures such as described in this paper would be easier to implement if standardised object-oriented tools existed on top of the semantic database.

Conclusion

We combined the best features of object oriented design (safety through encapsulation, conceptual simplicity, uniform interface, heritage) with those of relational databases (all data accessible, easy updating, good documentation). The new system has been in use for two years and we are converting the remaining old style modules to the new system. Many man-years have been gained this way and the resulting system is more reliable and better documented.

Acknowledgments

We thank L.Casalegno for writing the code for the first version of the GM frame and F.Perriollat for constant encouragement and support during the execution of this project.

BIBLIOGRAPHY

[1] Casalegno et al, Building software modules for driving hardware controlling physical variables: an object-oriented approach, EPS conference on control systems for experimental physics, Villars-sur-Ollon, Switzerland, September 1987.

[2] J.Cuperus, The database for accelerator control in the CERN PS Complex, IEEE PARTICLE Accelerator Conference, Washington DC, March 1987.