The Very Large Telescope real time database An Object Oriented Approach based over Rtap

G.Chiozzi
European Southern Observatory (ESO)
D-85748 Garching bei München, Germany
Email: gchiozzi@eso.org

T.J.Accardo
Computerized Processes Unlimited, Inc.
Metairie, LA, USA
Email: tja@cpu.com

Abstract

The Very Large Telescope (VLT) of ESO, under construction in the Atacama desert in Chile, consists of an array of four telescopes of 8m diameter.
The control software is inherently distributed, since it is based on LANs interconnecting several control workstations (WS) and specialized microprocessor boards (Local Control Unit, LCU). The workstation provide user interface and high level functions, while the LCUs handle real-time control.
A basic layer of common software provides, on both workstations and LCUs, services for interprocess communication, synchronization among CPUs, On-Line Real Time Database, errors and alarms handling, events logging and booking of equipment for security and protection.
On the workstation side, this software is based on the services given by Rtap, while on the LCUs specialized software implements all the necessary functionality with the same software interface. The On-Line Real Time Database is heavily distributed. Every unit in the system (WS or LCU) has its own hierarchical local database, where all the system parameters are stored and where real time data are collected.
To help the database design, development and maintenance during the lifetime of the VLT project, the Rtap concept of "point configuration files" has been extended introducing Object Oriented concepts. Using these extensions, a set of related points can be grouped in a class. This allows extensive reuse of common database structures defining instances of the classes or using inheritance to create new specialized classes.
A pre-processor analyses the files that describe the database and generates "point config files" in the standard Rtap format. Interactive tools are under development to provide the same functionality of the Rtap Database interactive tools, plus a full comprehension of the class and inheritance scheme.

1 Introduction

The first part of this paper will give a global presentation of the VLT project and of the general architecture of the control software under development. The On-Line Real Time Database is then introduced as a fundamental component of this architecture. After a brief discussion on the design issues that are at the basis of the choice of an Object Oriented approach, the chosen model is described in details. The text contains a number of examples from the VLT architecture, used to show how the described concepts can be applied.

2 The VLT project

The Very Large Telescope consists of an array of four identical main telescopes, each having a mirror of 8m diameter. When the four telescopes are used together, they behave as a single instrument with an equivalent

size of 16m. This shall be the largest size available at the beginning of the next century on ground based telescopes. The large size of the array provides also, compared to a single 16m mirror, a very high angular resolution, that corresponds to the possibility of resolving finer details[2].
The VLT site is in the northern Chile, in the Atacama desert, on top of the Cerro Paranal at an altitude of 2700m. The first telescope will be installed in 1996 and the others will follow at time intervals of one year. The whole VLT program, including instrumentation and interferometry will be concluded some years later, giving a very long time frame for the installation of the VLT and of its control software in particular.

3 The VLT Control Software

The Control Software for the VLT is responsible for the control of the main and auxiliary telescopes, the interferometer and the instruments. This correspond to a distributed environment of 120-150 coordinating workstations and microprocessor based VME systems (Local Control Units, LCUs). The LCUs are distributed in suitable locations to perform the real time control of the mechanical, electrical and optical components of the VLT (Fig.1). Each unit is connected to one or more specialized network. In order to work together, the different computing units must be synchronized. Depending on the level of accuracy required, time is obtained hardware-wise from a timing bus, or software-wise through the LAN, from a common time reference system.

The architecture of the Control Software reflects the physical hardware distribution of the telescopes and of the different instruments. Such a distributed architecture also increases hardware and software reliability of the whole system[9].

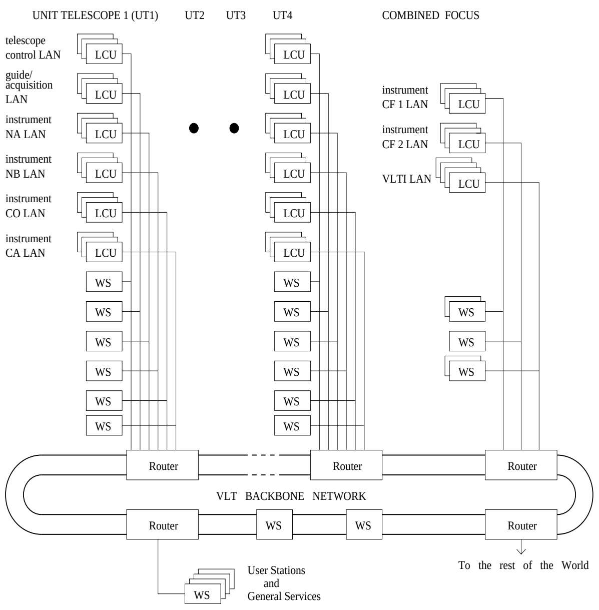

Fig.1:VLT architecture overview

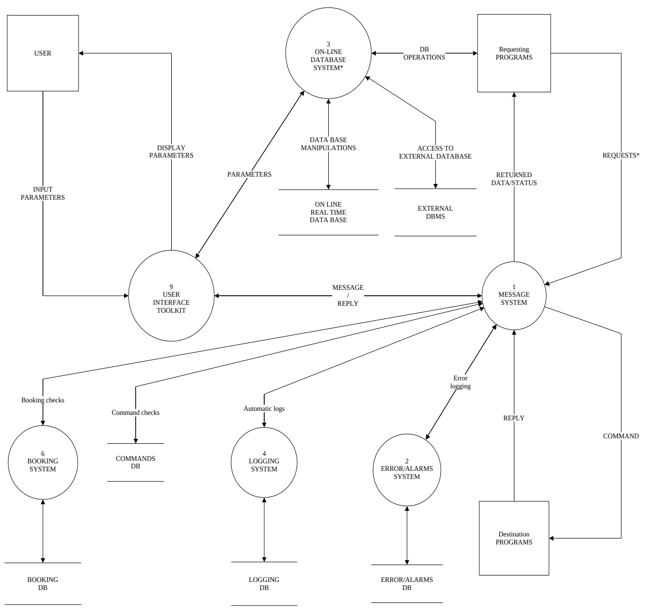

Fig.2:VLT central control software structure

The basic software used for the VLT control system, called Central Control Software (CCS), is a set of modules designed to provide services common to many applications (Fig.2). Most of these common services are available both on workstation and LCU platforms and provide on both the same application programming interface.

The commercial product Rtap (Real Time Application Platform), from Hewlett-Packard, is at the heart of CCS on the workstation side. On top of it, a layer of software implements the ESO-specific requirements.

The main modules of CCS are:

- **message system** - Provides interprocess communication services.
- **on-line database** - Provides real-time data store/access capabilities on a distributed architecture.
- **logging system** - Provides services for logging and retrieval of historical information.
- **error/alarm system** - Provides logging and notification of alarm and error conditions.
- **booking system** - Provides services for booking and protection of shared resources.
- **time system** - Provides the Time Reference System and synchronization services.
- **user interface toolkit** - Provides a common and uniform user interface platform.

4 The On-Line Database

The VLT architecture relies on an On-Line Database that provides real-time data capabilities on a distributed architecture (Fig.3).
As a consequence, a relevant part of the design of the VLT is the design of the sections of this database distributed on the different Workstations and LCUs.

Each local database has a hierarchical structure mapping the physical objects that constitute the telescope system (instruments, telescope control components....) and describing how they are logically contained one inside the other (how one component is “part-of” another one).

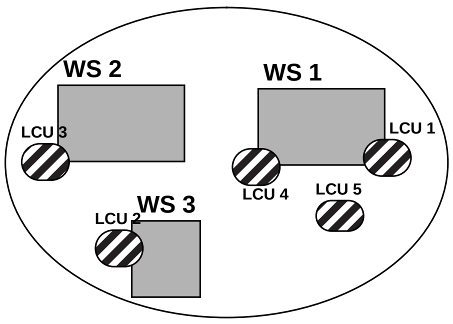

Fig.3:VLT On-Line Database structure

Each local database has a partial image only of the units of the system that the particular WS or LCU has to use. These components can be true property of the Workstation or LCU itself, or can be just images locally maintained for performance reasons, so that it is not necessary to ask for data through the network every time. The whole database is the merge of all the local sections.
This means that the global database comprehending all the Workstations and LCUs will have often different images for the same physical objects (usually a complete one for the LCU that handles it and a partial one for the WS controlling that LCU).
At the same time there will be a lot of different places in the database describing objects with the same or with a very similar internal structure and behaviour. For example, the VLT system will have a lot of motors, encoders or moving axis that have the same general characteristics and can be represented in a logical tree; here common logical parts can be grouped together and the designer can describe only the differences that characterize the peculiarities of each object.
Using an Object Oriented terminology[1], inside the database there are a lot of different instances of the same class (such as motors and encoders) or of similar classes (different kinds of motors), and different images of the same object: a “model” describing the full properties of the object (usually on the LCU’s database) and some partial “views” (usually on a control Workstation), containing only what is necessary to control the device from outside.

5 Design, development and maintenance requirements

To help the database design, development and maintenance phases during the lifetime of the VLT project, it is necessary to have tools able to automate and enforce the implementation of the concepts introduced in the previous section.

5.1 Database design

During the design phase of the database, it is necessary to have access to a common repository of templates for the description of the different modules than can be reused by various parts of the VLT, introducing some peculiar characteristics where necessary. If a concept of class, based on object oriented principles, is available, the design is much easier, since:

- a wide library of classes describing objects commonly used can provide simple basic components (such as I/O signals, encoders, motors) or higher level parts (such as axis, CCD devices...);
- a component of the VLT completely described by a class can be described with just a reference to the class, with the setting for initial values where necessary;
- an object that is different just for some aspects with respect to an instance of the class is designed making a reference to the class and describing what it has more and/or different: all the rest is inherited from the base class;
- a new object, suitable for reuse, becomes a class in itself and is readily available to all other potential users;
- the description of a class is not a flat entity, but it can contain, as internal members, instances of other classes, for example an axis contains an encoder, a control loop and a motor.

Having these concepts implemented, it will help a lot the designer to create similar database structures, uniform for the whole VLT project, and in reusing the code sharing knowledge.

5.2 Database development

Once a proper design has been done, the development of the true database must be a straightforward process. To get this result, it is necessary a simple description language for the database structure that:

- handles the definition of instances from classes
- is able to define a new class by difference with a parent class
- can describe multiple views of the same instance in a single place
- takes proper defaults for everything not specified

5.3 Database maintenance

During the maintenance phase there is often the need to modify the characteristics or the behaviour of an entity that has been used in many different places.
It is therefore necessary a support to modify the general description (i.e. the class description) in a way that the modification is automatically applied to all the instances.
At the same time, the modification has to be applied automatically to all the sub-classes.
This means that the simple copy-and-paste technique, where there is a template and new classes or instances are obtained copying and modifying it, cannot be used. This technique is at the same time tedious and error prone:

- it is easy to introduce errors while copying
- modifications cannot be propagated automatically
- it is easy to forget to propagate some modification or to introduce errors while doing this by hand.

6 Rtap Database structure

The On-Line database on the Workstations is the hierarchical real-time database of Rtap[9]. The LCU counterpart has been developed in order to be compatible with the Rtap implementation[4] for what concerns the general structure and the syntax used to describe data points, i.e. the syntax of the so called “point configuration files”.
The description of a database is a directory tree on the Unix or VxWorks filesystem: every directory represents a database point and contains a configuration file describing the point characteristics and subdirectories for the description of sub-points.
The concept of template is implemented having standard trees of point description files that can be copied in a specific position and modified by hand.
Obviously this approach cannot give any automatic support for inheritance and suffers for all the problems mentioned in the previous paragraph about copy-and-paste technique.

7 Class support extension

To fulfil the requirements analyzed in the previous sections, we have extended the semantic and syntax of Rtap point configuration files introducing the concept of class, with inheritance and overloading[2].
Real points in the database are built as instances of classes and a single file (called a “branch configuration file”) describes a full database branch.
This means that a branch config file contains the description of all the objects that are logically contained (i.e. subject to a “part-of” relation) inside a single entity.
Each branch config files contains:

- general branch definitions
- local classes definitions
- points declarations

A pre-processor reads “branch configuration files” as input and generates the corresponding database in the form of Rtap standard “point configuration files”, resolving all the dependencies between classes. The full database of a Workstation or of an LCU is built processing one or more branch config files.
The pre-processor implements all the standard features of the C pre-processor, such as conditional compilation and #define or #include directives[6].

7.1 Classes and inheritance

From a logical point of view, a class is a data type able to describe data structures (and their behaviour, with the support for methods), just like an integer attribute is able to describe an elementary integer data entity.
The class definition can thus be considered as a definition of a new structured data type and this will be used as any other available data type. This means that a class instance can be used just in the same way as a native type while defining attributes inside new classes or points. It is then possible to build hierarchies of points logically contained one inside the other.

> For example, every tracking axis will contain an encoder, a control loop and a drive (that are library-provided classes): this structure must be described while defining the class for the tracking axis and the declaration of a point of this new type must automatically generate the sub-points, just in the same way it generates the normal attributes.

Each new class has to be derived from another one, from which it inherits all the properties, in terms of attributes and behaviour. A part from the inherited attributes, inside a class definition it is possible to:

- redefine attributes to assign new initial values
- add new attributes
- overload structured attributes

- define and overload methods

The class `BASE_CLASS` appears as an ancestor for every other class and implements the internal behaviour used to support the class concept inside the Rtap DB.
The following syntax is used for class definition:

```
CLASS ParentClass ClassName
BEGIN
    ... class description ...
END
```

where all the attributes definitions (basic types or class types) are placed in the class description section.

> The TRACKING_AXIS's class definition of the previous example will have the following format:
>
> ```
> CLASS MOVING_AXIS TRACKING_AXIS
> BEGIN
>         ATTRIBUTE ENCODER      encoder
>         ATTRIBUTE CONTROL_LOOP loop
>         ATTRIBUTE DRIVE        drive
> END
> ```
>
> Where the base class is `MOVING_AXIS`.

7.2 Point definition

The points are the true objects that have to be put in the database to describe the system.
Usually they have to be instances of a class. The main target of this approach is in fact to have an extensible library of classes able to fulfil all the needs of the model: new requirements must be satisfied designing new classes or changing the characteristics of already existing ones. This is the key to have a uniform environment within all the parts of the database and to maintain it without manual intervention.
In the simplest case, it is necessary only to declare a point as an instance of the desired class. The pre-processor will take care of retrieving the class definition, creating the point with all the attributes inherited from the parent class and its ancestors and assigning the initial values to the attributes:

```
POINT ClassName point_name
```

where ClassName is the name for the class of the point and point_name is the name given to the instance.

> Consider the common example of a point used to get a temperature value from an input channel. This task can be accomplished using a point of class TEMPERATURE_INPUT. This is a specialized sub-class of a more generic ANALOG_INPUT.
> We can simply declare it in the form:
>
> ```
>         POINT TEMPERATURE_INPUT myTemperatureValue
> ```
>
> The point added to the DB will have all the characteristics inherited by the TEMPERATURE_INPUT class and its ancestors; for example:
> - actual temperature value
> - scaling and conversion information
> - out-of-range indication
> - quality indication
>
> and so on.

In some cases it is necessary to assign special initial values to some of the inherited attributes, to configure the initial status of the point.
This can be done using an extended point declaration format:

```
POINT ClassName PointName
BEGIN
   ... point description ...
END
```

Inside the BEGIN ... END area, it is possible to:

- redefine attributes to assign new initial values
- add new attributes that will be placed only in this specific point instance, without the need of defining a new class
- overload structured attributes

For example, the TEMPERATURE_INPUT class inherits from its parent ANALOG_INPUT the ability of generating alarms when the monitored value becomes higher than a specified limit (defaulted to 100 arbitrary units). In our specific instance the limit has to change to 50 arbitrary units, because the device where it is installed is more sensitive to heating effects. This can be done using the extended syntax:

```
POINT TEMPERATURE_INPUT myTemperatureValue
BEGIN
        ATTRIBUTE  int maxValue 50
END
```

7.3 Overloading of structured attributes

Inside a class definition we can put as attributes instances of a class. For example every instance of TRACKING_AXIS will have a motor.
By the way, quite often we are not able to really specify in the general class definition what specific type of sub-component will be instantiated. For example, different types of TRACKING_AXIS (i.e. ALTITUDE_TA, AZIMUTH_TA or ADAPTER_ROTATOR) will all have motors, but of different kinds.
This means that we must have the possibility to specify a generic class type for an attribute of the class, and to give more details about it at instancing time or while defining sub-classes.
This procedure is what we call “overloading of structured (or class-type) attributes”:

Every attribute instance of a class can be re-specified (overloaded) with another class-type, provided that it is a sub-class of the original one.

This is particularly useful when the general knowledge is enough to drive the device from outside (for example with commands from a workstation, but the internal implementation has to take into account hardware constraints that depend on the specific instance.

In the `TRACKING_AXIS` example, we will have a hierarchy of classes defined in the library and inheriting from the base class. Every sub-class will overload the motor attribute with a proper specific motor type. Consider the sample hierarchy for `MOVING_AXIS` and `MOTOR` given in Fig.4:

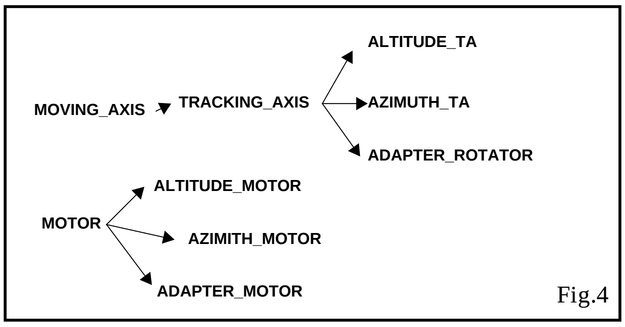

Fig.4

If the TRACKING_AXIS class is defined in the following way:

```
CLASS MOVING_AXIS TRACKING_AXIS
BEGIN
     .....
     ATTRIBUTE  MOTOR  motor
     .....
END
```

The sub-class will overload the motor attribute. For the example the AltitudeTA class will be:

```
CLASS TRACKING_AXIS ALTITUDE_TA
BEGIN
     ATTRIBUTE  ALTITUDE_MOTOR  motor
END
```

While defining a sub-class or an instance of a class that contains class-type attributes, it is also possible to modify some characteristics (typically the value) of an attribute inside the class-type attribute itself. When there are more levels of nesting between sub-points, the attribute to be modified can be some levels down in the hierarchy of the instances.

7.4 Static and automatic attributes

Usually all the attributes defined in a class will contain values to represent the current status of a specific instance of the class. These values can be constant or can vary with time, but in any case they are used to specify the properties of that single specific instance.
Each instance of the class must have its own personal copy of all these attributes (called “automatic”) and by consequence the point put in the database will contain all these automatic attributes as inherited by all the ancestor classes.
It is easy to see that this is not the only case: some attributes describe characteristics common to all the instances of a class (called “static”). In this case it does not make sense to have a private copy of the attribute in each instance.

If we consider a configuration in which we use 10 exactly identical motors, it becomes evident that the maximum rotation is a constant value common to all these 10 instances. The better solution in this case is to have a special sub-class of motor and to have a way of defining and accessing the maxRotation attribute as “static” for the whole class.

The pre-processor recognizes the following syntax to define “static” attributes:

```
STATIC_ATTRIBUTE <Type | ClassName> AttributeName
BEGIN
        .... attribute description ...
END
```

From the user point of view, there is no difference accessing static or automatic attributes, but the static ones are actually stored in a common place inside each local database and are actually shared by all the instances of the same class residing in that local database.

8 Scan system support

In the general VLT database architecture, the same physical object can be mapped both on the local database of an LCU and on the one of a Workstation. By the way, in most cases this last image is only a partial view of the LCU true object and not all the attributes will be mapped.
More over, the mapped attributes must actually contain the same values stored in the LCU image.
This means that values that can change with time must be kept aligned using the Scan System.
Since this necessity is usually known at database design time, the loader provides a way to define Scan Links.

In the ATTRIBUTE declaration it is possible to define scan links to attributes of other points using a simple syntax:

```
ATTRIBUTE Type AttributeName
BEGIN
      // To define scan by polling
      SCAN_POLL  PathOfScannedAttribute PollTime
      // To define scan by srbx
      SCAN_SRBX PathOfScannedAttribute deadband_type deadband
END
```

Consider for example an attribute that must contain a value polled from a given attribute of a point on an LCU every 2.0 seconds:

```
ATTRIBUTE float temperatureCurrentValue
BEGIN
     SCAN_POLL "@LTE23:<alias>temperature.value" 2.0
END
```

This, coupled with the ability to use pre-processor directives, makes easy to define in the same branch config file both the images of a point.

Consider the example of a device for which we have to monitor the temperature. The device will have its model on an LCU and an image on the control Workstation. The only attribute we need to monitor from the workstation is the value of the temperature, and this must be kept aligned via the Scan system.
To implement this behaviour we can define the class for the device in the following way:

```
CLASS BASE_CLASS HEATING_DEVICE
BEGIN
        #ifdef LCU
            ALIAS "heatingDevice"
            ..... attributes used only in the LCU
        #endif

        // "double definition" for the temperature attribute
        ATTRIBUTE TEMPERATURE_INPUT temperature
        BEGIN
        #ifdef LCU
             ... attributes properties in the LCU ...
        #else
             SCAN_POLL "@LTE23:<alias>heatingDevice.temperature" 0.5
        #endif
        END

END
```

9 Method Support

A complete definition for a class of objects comprehends both the variables used to describe the internal status of the object itself and the description of the behaviour and of the public interface of the object. The status is given by the current value of a set of variables stored in database's attributes. The interface is defined by the methods provided by the class.
A "Method Server" (running on the workstation or on the LCU where the object's database is placed) handles the requests for methods implemented by the object. This is currently under development.
From the point of view of the designer of a class, all the (new) methods defined are described in a Command Definition Table that contains a full description of the interface for the method, in terms, for example, of parameters (with type, units, range and default value, if they are optional or not), return values, help on line.
Every class inherits the definition for all the methods of its parent class and the implementation can be overloaded, but the interface definition cannot: every subclass can implement the method in a different way, but the interface must be the same for all the implementations.

The implementation of a method is done mapping its name to a function or a process that will be executed ev-

ery time the method is called for an instance of the class.
The mapping is done inside the `BEGIN...END` section of the class declaration.

For example, the declaration for class `RANGE_VALUE` will have the following form:

```
CLASS BASE_CLASS RANGE_VALUE
BEGIN
        // Attributes
        ATTRIBUTE bytes32 stringValue // string where the value is stored
        ... other attributes ...

        // Methods
        METHOD SET   SetRangeValue   FUNCTION
        METHOD CHECK CheckRangeValue FUNCTION
END
```

The SetRangeValue() and CheckRangeValue() functions will perform basic checks to verify if the given value is acceptable or not. In particular the SetRangeValue() function first calls CheckRangeValue() and if everything is ok, sets the new value.

10 A small example

The following small example, extracted from the Database Loader user Manual [2], shows a simplified implementation for a MOTOR database using the dbl syntax and the Rtap database structure generated for an instance of the class.

```
// ***********************************************************************
// * MotorExample.db - a simplified implementation of a MOTOR database.
// ***********************************************************************
// We define the MOTOR class and the support classes, for motor's sub-components

// CLASS MOTOR_STATUS: A set of attributes used to specify motor's status.
CLASS BASE_CLASS MOTOR_STATUS
BEGIN
        ATTRIBUTE int opMode        1
        ........
END

// CLASS SERVO_AMPLI: Every motor has a servo amplifier.
CLASS BASE_CLASS SERVO_AMPLI
BEGIN
        ATTRIBUTE int id               // Servo ampli. identification value
        ATTRIBUTE int boardStatus
        ........
END

// CLASS MOTION_CONTROLLER: Base class to define a motor's motion controller.
CLASS BASE_CLASS MOTION_CONTROLLER
BEGIN
        ATTRIBUTE int         id    // Controller ident. value (one of a list)
        ATTRIBUTE int         status
        ATTRIBUTE INTERRUPT   driveFault // Library defined class
        ......
END

// ========== THIS SECTION DESCRIBES THE MOTOR CLASSES ===============
// CLASS MOTOR: Basic abstract motor class.
// Every motor is characterized by to components:
//       - A motor status
//       - A servo amplifier
CLASS BASE_CLASS MOTOR
BEGIN
        ATTRIBUTE MOTOR_STATUS status
        ATTRIBUTE SERVO_AMPLI  amplifier
```

```
END

// CLASS MOTOR_MCON: special kind of motor: it has also a MOTION_CONTROLLER.
CLASS MOTOR MOTOR_MCON
BEGIN
        ATTRIBUTE MOTION_CONTROLLER controller
END

// =========== THIS SECTION DESCRIBES CHARACTERISTICS OF ================
// =========== ACTUAL MOTOR COMPONENTS (NOT ABSTRACT DESCRIPTIONS) ======
// We will have here as many classes as different types of servo
// amplifiers and motion controllers we have.

// CLASS VME4SA: A specific servo amplifier.
#define VME4SA_ID  1
CLASS SERVO_AMPLI VME4SA
BEGIN
        ATTRIBUTE int        id                VME4SA_ID
END

// CLASS MAC4: A specific motion controller.
#define MAC4_ID     1
CLASS MOTION_CONTROLLER MAC4
BEGIN
        ATTRIBUTE int        id                MAC4_ID
        ATTRIBUTE INTERRUPT  emergencyStop
        ATTRIBUTE INTERRUPT  motionEnd
END

// ========== A STANDARD MOTOR CONFIGURATION USED IN THE VLT ========
// CLASS STD_MOTOR_1: This motor has a VME4SA amplifier and a MAC4 controller.
CLASS MOTOR_MCON STD_MOTOR_1
BEGIN
        ATTRIBUTE VME4SA amplifier
        ATTRIBUTE MAC4   controller
END
```

The interrelations between the classes defined in this example can be summarized in the following diagram (Fig.5), where the *"is a"* relation, (inheritance dependence between a class and a sub-class) and the *"part of"* relation (an instance of a class is a member of another class) are put in evidence.

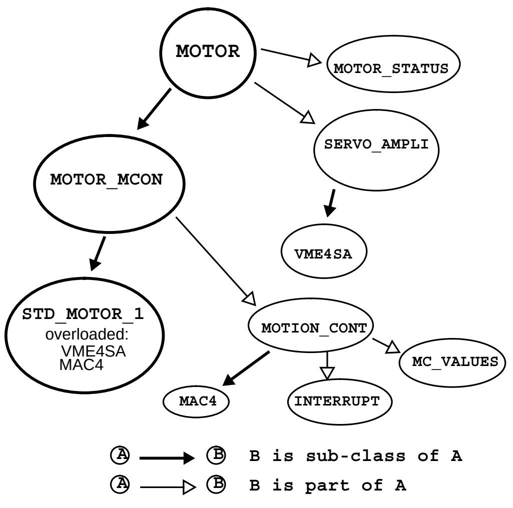

Fig.5: Class diagram

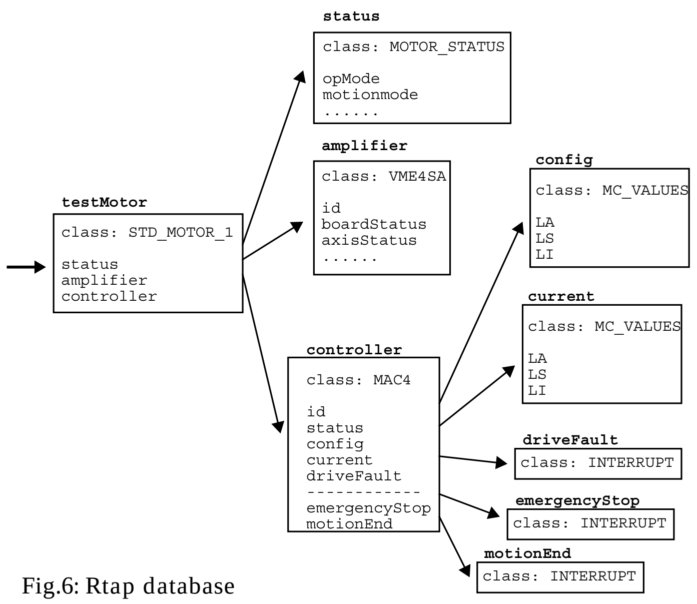

Fig.6: Rtap database

As a consequence, every time we define an instance of class STD_MOTOR_1 we get the Rtap database structure shown in the diagram (Fig.6). The Rtap database will contain also a point per each class (stored in a special "class configuration branch") used at run time for the handling of methods and static attributes[2].

11 Interactive database tools

The interactive database tools distributed within Rtap can be used with the database created by the pre-processor, but only to browse the database (modifications will easily put it in an inconsistent status). These tools are not available in the LCU environment and are not able to understand the concept of class, dealing with inheritance and static attributes.
For this reason, the development of interactive tools specialized for this purpose is planned for a next release of the software.
In particular it is foreseen the development of a Database Editor (providing functions similar to RtapDbConfig, but with support for classes) and of a Class Browser to manipulate the classes' definitions and the classes' hierarchies, similar in concepts to the class browsers available for all the object oriented programming languages.

12 The pre-processor

The pre-processor to generate Rtap "point config files" from "branch config files" has been in itself designed using an Object Oriented approach and has been developed using the Tcl script language[8] and it's object extension [incr Tcl][7].

The implementation of the pre-processor, called dbl (DataBase Loader), consists of two main activities:

1. parsing the branch config file
2. creating internal representations of attributes, points, classes, and structural information about the database

A third activity, of course, is to actually produce the "point config files", but this is a simple exercise once the internal representation is complete.

A detailed description of the branch config files' parsing is beyond the scope of this paper. Here we want only to mention that the most important activity during the parsing is deciding when to create and modify internal representations of attributes, points, classes, etc.

As a branch config file is read, internal data structures - objects - are created to represent what the branch config file is describing.
These object types include:

- attributes
- points
- classes
- an Rtap database structure

Attribute objects simply encapsulate information associated with an attribute in a point: name, value, type, DE type, CE definition, and read/write groups. These items may be both queried and modified in an attribute object.

Point objects encapsulate similar types of information, such as name, alias, class, and categories. In addition, point objects may have a list of associated attribute objects. Branches of points can be constructed, since a point can also contain a list of other points which are children of itself, as well as an indication of who its parent point is.

Class objects encapsulate the raw information which describes a class. This includes basically the same information as the point object, but both simple attributes and structured attributes (instances of another class) are supported. Instead of storing parent and children points, class hierarchy information is kept instead. The most important aspect of a class object, though, is its ability to create an instance of itself. That is, it can create a branch of point objects which represent an instance of that particular class. This branch of point objects may then be furthermore customized, e.g. to give attributes initial values or insert scan links. Finally, this branch of points may be inserted in a database object.

The database object is a representation of the point structure in an Rtap or LCU database. As instances of classes are created, they are inserted into the database object, which captures the structural information about that particular branch, and output to point config files. By maintaining the structure of the database internally, a certain amount of error checking can be done during the pre-processing phase instead of the load phase.

The object-oriented approach for implementing the pre-processor allowed fast prototyping as well as a great deal of flexibility for change. The object types used can be effectively used in other applications such as a class browser or builder. Furthermore, by decomposing the data in this way, one can leave the external definition of the database object in place, for instance, but change its internal implementation to use a live Rtap database instead of point config files.

13 Conclusions

Although specifically designed for the VLT project, the Object Oriented concepts implemented by the dbl preprocessor seem to provide a general scheme to improve maintainability and reusability of data structures based over Rtap, along with a better integration with the software that will be used to access the data. Up to now this has been tested only on a limited scale, but in the next months the VLT will enter its hottest development phase and the whole CCS concept will be heavily exercised, both internally at ESO and externally by contractors responsible for instrumentation software.
This will tell us if the goal has been reached.

14 Bibliography

[1] G.Booch - **Object-Oriented Analysis and Design, 2nd. ed.** - Benjamin/Cummings, CA, 1994

[2] G.Chiozzi - **CCS On Line Database Loader User Manual, Issue 1.1** - VLT-MAN-ESO-17210-0707, European Southern Observatory, Munich, 1995

[3] D.Enard - *The European Southern Observatory Very Large Telescope* - J.Optics, vol.22, n.2, pp.33-50, 1991

[4] B.Gilli - *Workstation Environment for VLT* - Proceedings of SPIE, vol.2199, pp.1026-1033, 1994

[5] B.Gustafsson - *VLT Local Control Unit Real Time Environment* - Proceedings of SPIE, vol.2199, pp.1014-1025, 1994

[6] B.W.Kernighan D.M.Ritchie - **The C Programming Language, second ed.** - Prentice Hall, NJ, 1988

[7] M.J.McLennan - *[incr tcl]* - *Object-Oriented Programming in Tcl* - Proceedings of Tcl/Tk Workshop, pp.31-38, 1993

[8] J.K.Ousterhout - **Tcl and the Tk Toolkit** - Addison Wesley, MA, 1994

[9] G.Raffi - *Control software for the ESO VLT* - Proceedings of Int. Conf. on Accelerator and Large Experimental Physics Control systems, Tsukyba, Japan, Nov. 1991

[10] **RTAP/Plus Integration** - Hewlett-Packard, Canada, 1993

* Rtap is a registered trademark of Hewlett-Packard