A component oriented framework for the control-command environment of the Virgo experiment

Christian ARNAULT [1], VIRGO Collaboration [2]

LAL Bat 200, Orsay France, `arnault@lal.in2p3.fr`

The Virgo experiment is a challenge both at the physics side, for the data production and analysis and for the online control of activities. This poster session presents a set of tools, organized as a framework which constitutes a significant part of the online control command environment.
This environment relies on a component oriented packaging and an object-oriented design that provides a consistent way of producing and using the control-command components so that quality concepts involved in their development put emphasis on reliability, ease of long-term maintenance and interoperability.
The Virgo environment and particularly the online context is presented with in-depth discussion on the basics components provided in this framework.

Key words: control; communication; graphics; specification languages;

1 Introduction

The purpose of the Virgo interferometer is to detect arrival of gravitational waves on earth from astronomical sources. This is achieved by measuring the motion of *free* masses caused by the travelling deformation of space-time. The foreseen displacement is so tiny on human scale that it should be compared to measuring hair diameter over the distance from earth to the nearest star.

The apparatus is based on distance measurements performed with interfering light rays travelling in two vacuum enclosures 3 km long, forming a Michelson interferometer, with arms containing a Fabry-Perot cavity. It is situated in Cascina, near Pisa in Italy.

The architecture of the control and read out systems has to account for the large distances which may separate two components. To preserve flexibility in the interferometer setting up, the control systems are organized as much as possible into independent units in charge of the adjustment of a well defined component.

Each control process is a server built in the framework of the client-server model. It is a real time program which has direct access to the hardware required by its functionality (typically ADC's, servoloop,...). The user interfaces (clients of the control processes) configure the server and/or monitor the data provided by the server. Several such clients can be simultaneously used to monitor its informations, but only one client at a time (the master client) has the privilege to configure a server. Possible conflicts are locally solved using standard rules and information provided by the online database.

A framework of software components has been built in order to provide uniform services all accross this environment. An object oriented design is used to ensure independence of components and flexibility against future evolutions. Some of the most important components of this framework are detailed in the next sections.

[1] CNRS Software engineer
[2] `http://galileo.pi.infn.it:80/virgo/`

2 Cm : the core of the distributed environment

Cm [1] handles all the communications between processes on the computer network by providing platform independent communications tools. Cm applications are referenced by a logical *name*, that must be unique within one Cm domain. Cm domains are managed by one special application - the NameServer - in charge of the physical addressing scheme (allowing several independant such domains to coexist).

Cm transparently manages the asynchronous operations through a high level framing protocol and callback based mechanisms. The CmMessage objects encapsulate every formatting operations needed to construct structured messages.

3 Gb : the reactive graphical object manager

Gb [1] provides a specification language for *scenes* made of structured graphical representations of reactive objects, an engine able to interpret it and an Xt *widget* (named GbButton) able to drive the graphical engine. Logical Virgo components (eg. *states*, *controls* or *resources*) or physical ones (eg. *devices* or *processes*) can be represented and operated using Gb.

The Gb language permits to assemble graphical primitives into a hierarchical structure, define templates of graphical objects, defining attributes of such templates and establish dynamic links between objects. The graphical engine manipulates *events* that may be either user selection of a graphical object or *symbolic events* sent by an object. Objects can be made *sensitive* to both kinds of events and may react either by sending a symbolic event or by calling a function of the application.

4 Db : the online database

Db [1] is the common manager and provider for configuration data of any of the online activities. Data items consist of *named* C *objects* and can be of any legal C type (with a very few exceptions such as *enumerated types*).

Objects are managed in a central Db server (that handles also access control on objects). Therefore objects are accessed by their name transparently over the network and selections of set-of-objects can be done with various criteria (name pattern, conditions on value, ...). Objects are stored into textual files (using a human readable format) that may even permit a non-network access to objects. An history of the contents evolution for every object is maintained by the system and recoveries of object values are possible on the basis of absolute or relative time and date specifications, or using symbolic tags.

5 Fsm : the finite state machine engine

Fsm [1] manages sets of distributed applications behaving as a *network* of finite state machines. The Fsm toolkit provides a specification language (refered to as *the Fsm language*) with which classes and machines can be defined. Requests to operate transitions can also be stated in that language.

Machines are generically defined in *classes* where possible *states* and *transitions* for any instance of a class are declared. Transitions define permitted *paths* between the possible states of a machine. They may be submitted to *rules* that must be obeyed by the supporting machine before the destination state can be reached. These rules are formed with *expressions* specifying constraints on states of other machines or *sets of* machines. After rules verification, transitions trigger *sequences* of operations expressed as *requests* to set machines or *sets of* machines to states (resulting in chaining transitions) and user actions (typically compiled user functions).

6 `Scl` : the sequencing control language

`Scl` [1] provides a specification language for defining and operating real time objects (objects which reactions cannot be strictly predicted in time) and a multi-threaded *engine*. `Scl` objects are specified by their properties (*attributes*, *states*, *behaviour*) and are instanciated in different `Cm` applications using an internal transparent protocol for exporting their states.

`Scl` extends the notion of *event* generally found in sequential control systems by giving objects a real-time *sensitivity* to modifications of the internal state of other objects, and by considering not only their *discrete* states, but also the evolution of their *continuously valued* attributes. Real-time or logical constraints can be applied on such sensitivity in order to control the objects behaviour.

Both simulations and real control sequences can be specified and operated using the multi-threaded `Gb` engine.

7 Conclusion

The component based approach used to design this framework really improved the development phases, by allowing a high flexibility in individual design while keeping a powerful interoperability. The framework is now continuously extending towards either new generic components (eg. error logging, shared memory management, scope emulation, etc..) or specific support for Virgo devices (such as for vacuum control).

References

[1] `http://www.lal.in2p3.fr/virgo/developments.html`