THE PHELIX CONTROL SYSTEM BASED ON *CS*-FRAMEWORK 3.0

H. Brand, D. Beck, S. Götte, M. Kugler, GSI, 64291 Darmstadt, Germany

Abstract

The Petawatt High Energy Laser for Ion eXperiments [1,2], will offer the unique combination of a high current, high energy (GeV/u) heavy-ion beam with a powerful laser beam thus providing the opportunity to investigate a variety of fundamental science issues in the field of atomic physics, nuclear physics, and plasma physics. The PHELIX Control System (*PCS*) is based on the *CS* framework [3,4]. About 40 additional classes were developed for the *PCS* and about 250 objects are distributed on 13 PC's publishing about 10000 process variables. The *PCS* has been upgraded to *CS* version 3.0 recently. In *CS* 3.0 the entire communication layer has been changed to DIM [5] (Distributed Information Management), which is a light weight protocol for inter-process communication based on TCP/IP. The *PCS* was redesigned to make use and profit from the concept of named services. Clients may receive information from a service (observer pattern) or may send a command to a server (command pattern). By these means the implementation of the *PCS* behaviour with hierarchical state machines was eased.

CS FRAMEWORK 3.0

CS is a framework that can be used by many experiments. It is a multi-threaded, event driven, object oriented and distributed framework with SCADA functionality based on LabVIEW [6,7] from National Instruments and DIM. An experiment control system can be developed by combining the *CS* framework with experiment specific add-ons. *CS* is supported on MS-Windows, Linux and the real-time OS Pharlab (LabVIEW RT).

The *CS* framework provides an object oriented approach for standard LabVIEW by using template-VI's to create unique object references and threads and Functional Global Variables (uninitialized Shift-Registers) to maintain the object attribute values protected by semaphores. DIM is used for the local and network communication layer. The *CS* framework provides base classes for GUI's, State Machines or active processes that can react on events (see Fig. 1). As an example, objects of *BaseProcess* class can subscribe to DIM services and declare commands to ensure a common event data structure. Objects are instances of such classes and are created at runtime. Due to the common event mechanism, objects of *BaseProcess* or inherited classes can be dynamically combined to a dedicated distributed control system like LEGO® bricks.

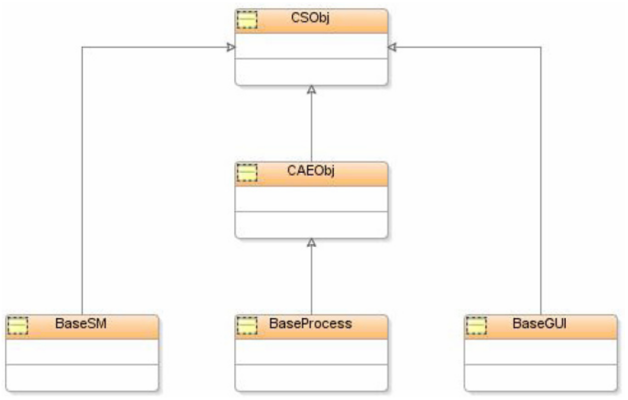

Figure 1: *CS* Framework Baseclass Hierarchy

FINITE STATE MACHINE

Many *PCS* classes inherit from *BaseSM* class which is the base class for finite state machines.

Each class that inherited from *BaseSM* has to overload an abstract method, which is called from the *BaseSM* thread, to implement its behaviour. The current state of an object is published as a DIM service.

The implementation of the overload method contains a case structure. Each case corresponds to a unique state and has the same sequence stucture.

1. Action on Entry: The first step implements all actions that must be executed on entry of state.
2. Do action: The next step comprises a while loop. It contains all actions that have to be executed while remaining in the state. This while loop will be stopped if a valid state change request is received. Valid next states can be defined by the class developer.
   A state change can be triggered in two ways.
   - Internal: If valid default next state is specified, the state machine will switch to it. If the default state is not specified, the object will remain in the same state and perform just another loop iteration.
   - External: The *BaseSM* class provides a method to request a conditional state change. This method has the following input parameters: requested state, condition, list of the next states that are allowed from the user point of view.
3. Action on Exit: Finally all actions have to be implemented that must be executed on exit of state.

PHELIX COMPONENTS

Fig. 2 shows the schematics of the fs-option of PHELIX which will deliver pulses of up to 500 J in 500 fs. The ns front-end will generate pulses of up to 10 ns and will allow a maximum energy of up to 1 kJ. The maximum energy is limited by the damage threshold of the FR-5 glass of the Faraday isolator. With the booster amplifier (not shown in Fig. 1), the energy may be increased to 4 kJ for 10-ns pulses.

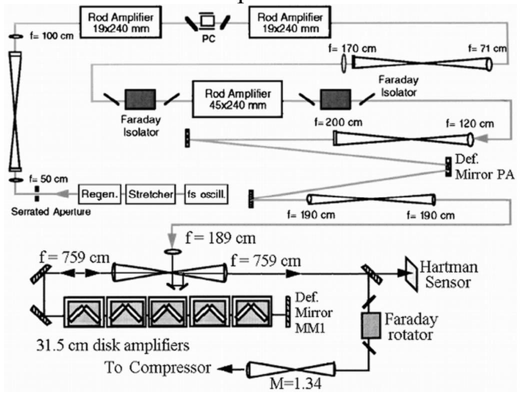

Figure 2: Schematics of PHELIX.

PHELIX CONTROL SYSTEM ARCHITECTURE

The PHELIX Control System follows the architecture of the laser system and is composed hierarchically of many subsystems and modules as shown in Fig. 3. The initial laser pulse is produced in the ns- or fs-Frontend laser systems. It will then be amplified in the Preamplifier (PreAmp) and Mainamplifier (MainAmp). There are two Pulsed Power systems (PP PreAmp and PP MainAmp) that provide the pump energy. Each subsystem controls devices, arbitrary function generators, power supplies, motors etc., and diagnostics like oscilloscopes, cameras, power meters etc. Such subsystems are executed on their own local control nodes. Other subsystems like Timing or Data Acquisition are running on separate nodes since they have to control or readout several other subsystems.

PILS (PHELIX InterLock System), Beamline and Sequencer integrate all other subsystems to one global PHELIX Control system each focusing on different aspects.

The PHELIX control system uses the common three layer approach, device, application and operation layer. A central part of the *PCS* is modelled by using state charts.

Many PHELIX classes have inherited from all three active base classes, *BaseProcess*, *BaseSM* and *BaseGUI*. Most *PCS* objects have their own graphical user interface (GUI) for local operation. But the GUI event loop does not perform requested actions directly, but sends itself an event to be processed by its BaseProcess event loop thread asynchronously. By this means blocking of the GUI event loop is avoided and all user actions can also be dispatched programmatically from other objects, e.g. higher level application layer objects like state machines or sequencers. Their states e.g. *remote controlled* or *local operation* are implemented as state machines. GUI controls are disabled when an object becomes remote controlled and enabled again when it becomes released.

Objects typically subscribe to DIM services of other application and device layer instances. From this data they calculate their internal state or request state changes.

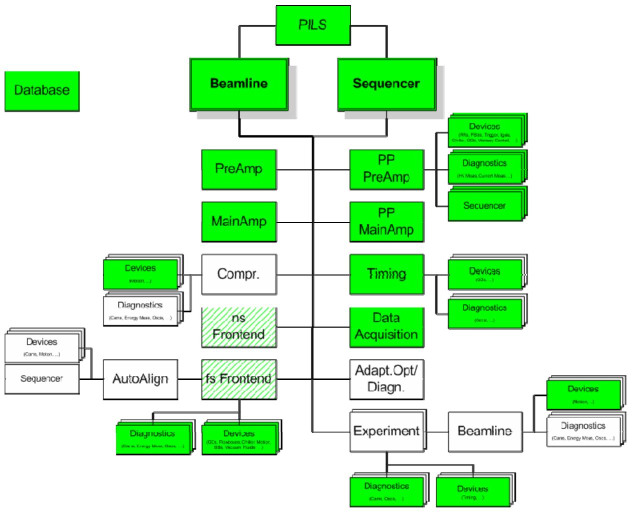

Figure 3: This figure illustrates the components of the *PCS* and their dependencies.

Device Layer

Each hardware device type is represented in the software by a class and each device by an object. *PCS* device objects are communicating with their devices directly via field bus (RS 232/485, GPIB, TCP/IP socket) or via OPC. They publish the current device state via DIM and wait for commands to be sent to the device. Device objects are passive that means they are not acting on their own.

Application Layer

Many *PCS* application layer objects are aggregations that control elementary devices or other application layer objects. Their behaviour is also defined by state machines as described before.

- Such an object requests the remote control of child objects. It creates them, if they do not already exist.
- The local GUI of a child object becomes disabled, so the parent object has exclusive control.
- An application layer object can be operated locally by using its GUI or controlled remotely by higher level application layer objects or from the operation layer.
- When the parent object becomes destroyed its child objects are set to local control and will be released by the parent object.

Such application layer objects have typically also inherited from *BaseProcess* and

Operation Layer

The PHELIX Sequencer is an example for an object in the operation layer. It provides a GUI that enables the PHELIX operator to prepare and deliver a PHELIX laser shot to an experiment. The sequencer requests the remote control of objects in the application layer. Their local GUIs become disabled to avoid concurrent modifications. The sequencer then monitors the status information and sends configuration commands or state change requests to objects in the application layer.

CONCLUSION

The PHELIX Control System has been successfully ported to *CS* Version 3.0 and was redesigned with respect to new features of the DIM communication layer, especially the Publisher – Subscriber pattern:

- All objects publish their status via DIM. Dependent objects subscribe to these services and react on value changes.
- The Publisher – Subscriber mechanism makes it easy to trigger transitions of finite state machines.
- A central part of the *PCS* has been designed and implemented by using hierarchical finite state machines.
- It provides stable and continuous operation with about 250 objects distributed on 13 PC's publishing about 10000 process variables.
- A first Laser-Shot with design energy of 500J was performed in March 2007

REFERENCES

[1] PHELIX Progress Report , GSI Report 2006, PLPY-PHELIX-15, p. 291, ISSN 0174-0814
[2] http://www.gsi.de/forschung/phelix/index_e.html
[3] D. Beck et al., Nucl. Instr. Meth. A 527 (2004) 567-579
[4] http://wiki.gsi.de/cgi-bin/view/CSframework/WebHome
[5] http://www.cern.ch/dim
[6] R. Jamal and H.Pichlik, “LabVIEW Applications and Solutions” (1999) Prentice Hall.
[7] http://www.ni.com/labview