Tools for the Automation of Large Distributed Control Systems

C. Gaspar and B. Franek

***Abstract*—The new LHC experiments at CERN will have very large numbers of channels to operate. In order to be able to configure and monitor such large systems, a high degree of parallelism is necessary. The control system is built as a hierarchy of sub-systems distributed over several computers. A toolkit—SMI++, combining two approaches: finite state machines and rule-based programming, allows for the description of the various sub-systems as decentralized deciding entities, reacting in real-time to changes in the system, thus providing for the automation of standard procedures and for the automatic recovery from error conditions in a hierarchical fashion. In this paper we will describe the principles and features of SMI++ as well as its integration with an industrial SCADA tool for use by the LHC experiments and we will try to show that such tools, can provide a very convenient mechanism for the automation of large scale, high complexity, applications.**

***Index Terms*—Automation, control systems, distributed systems, finite state machines, rule-based systems.**

I. Introduction

LARGE Control Systems, like the ones needed by the LHC experiments have some specific requirements.

- Distribution and Parallelism—Due to the large number of devices and input/output channels, the acquisition and monitoring of the data has to be done in parallel and distributed over several machines.
- Hierarchical Control—The data gathered by the different machines has to be summarized in order to present a simplified but coherent view to the users Decentralized Decision Making—Each sub-system should be capable of taking local decisions since a centralized decision engine would be a bottleneck.
- Partitioning—Due to the large number of different teams involved and the various operation modes of the system the capability of operating parts of the system independently and concurrently is very important.
- Full Automation—Standard operation modes and error recovery procedures should be, as much as possible, fully automated in order to prevent human mistakes and to speed up standard procedures.
- Intuitive User Interfaces—Since the operators will not be control system experts it is important that the user interfaces provide a uniform and coherent view of the system and are easy to use.

Manuscript received June 16, 2005; revised September 20, 2005.

C. Gaspar is with the CERN, CH 1211 Geneva 23, Switzerland (e-mail: Clara.Gaspar@cern.ch).

B. Franek is with the Rutherford Appleton Laboratory, Didcot OX11 0QX, U.K. (e-mail: B.Franek@rl.ac.uk).

Digital Object Identifier 10.1109/TNS.2006.874470

In order to solve these problems effectively the four LHC experiments at CERN have combined efforts by creating a common control project—the Joint Controls Project (JCOP) [1], to define and implement common solutions for their control and monitoring systems.

In the context of JCOP a common architecture has been devised and a common control Framework [2] has been developed.

This Framework is based on a SCADA (Supervisory Control And Data Acquisition) system called PVSSII [3]. PVSSII, although providing most of the needed features, does not provide for hierarchical control and abstract behavior modeling. So another tool: SMI++ [4] has been integrated with PVSSII and can thus be used as a component of the Framework.

II. Architecture

From the software point of view, JCOP adopted a hierarchical, tree-like, structure to represent the structure of sub-detectors, sub-systems and hardware components. This hierarchy should allow a high degree of independence between components, for concurrent use during integration, test or calibration phases, but it should also allow integrated control, both automated and user-driven, during physics data-taking.

This tree is composed of two types of nodes: "Device Units" (Devs) which are capable of "driving" the equipment to which they correspond and "Control Units" (CUs) which correspond to sub-systems and can monitor and control the sub-tree below them. Fig. 1 shows the hierarchical architecture defined by JCOP.

The architecture defined by JCOP is the basis for the development of the common Framework. Each LHC experiment can than adopt this architecture and use the Framework tools wherever they find it suitable.

III. Framework

The JCOP Framework provides for the integration of the various components (devices) in a coherent and uniform manner. JCOP defines the Framework as:

"An integrated set of guidelines and software tools used by detector developers to realize their specific control system application. The Framework will include, as far as possible all templates, standard elements and functions required to achieve a homogeneous control system and to reduce the development effort as much as possible for the developers".

The architectural design of the software Framework is an important issue. The Framework has to be flexible and allow for the simple integration of components developed separately by different teams and it has to be scalable to allow a very large numbers of channels.

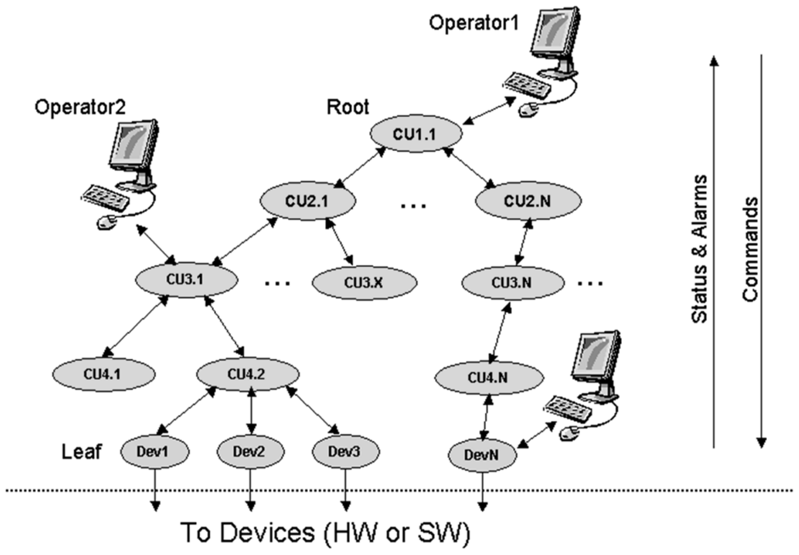


Fig. 1. JCOP software architecture.

Some of the components of this Framework include the following.

- Guidelines imposing rules necessary to build components that can be easily integrated (naming conventions, user interface look and feel, etc.)
- Drivers for different types of hardware, such as fieldbuses, and PLCs.
- Ready-made components for commonly used devices configurable for particular applications, such as high voltage power supplies, temperature sensors, etc.
- Tools to build control hierarchies of device and control units as described in the architecture chapter.

While "Device Units" are typically implemented using PVSSII features directly, "Control Units" are abstract objects and are better implemented using a modeling tool. For this purpose SMI++ was integrated into this Framework.

IV. SMI++

SMI++ is a toolkit for designing and implementing distributed control systems. SMI++'s methodology combines three concepts: object orientation, Finite State Machines (FSM) and rule-based reasoning.

Classes and objects allow the decomposition of a complex system into smaller manageable entities. Each entity, or object, is described as a finite state machine, allowing the modeling of its behavior in terms of simple states and actions. Each object is always in a well-determined state. For each state a list of rules can be declared which when fulfilled will trigger, asynchronously, a change of state or the execution of an action.

SMI++ objects can be "concrete" or "abstract". Concrete objects interface to real-world devices, like a power supply or a temperature sensor, through an associated process: a proxy. Abstract objects represent logical entities, like a Run or the Detector Control System, and are implemented within SMI++.

Logically related objects, abstract and concrete, can be grouped inside SMI++ "domains" representing a given sub-system. Fig. 2 shows the interconnections between the components of the SMI++ toolkit. In this figure two SMI++ domains are depicted, one of them containing several connections to hardware devices, through proxies. A user interface can attach to the different domains in order to view the states of their objects or to send commands to them.

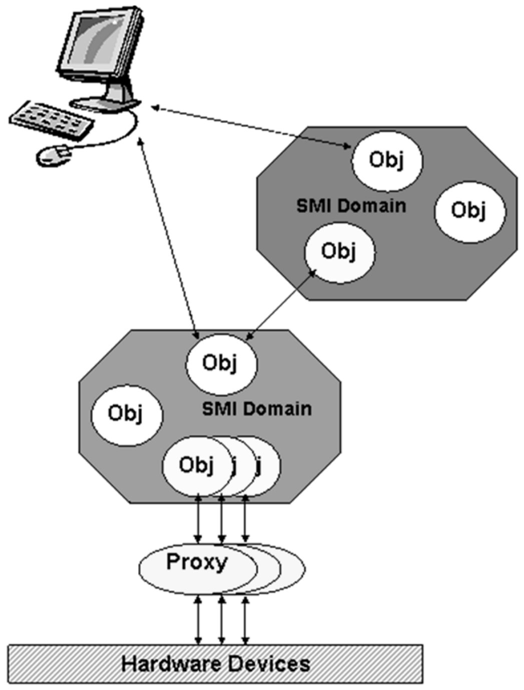
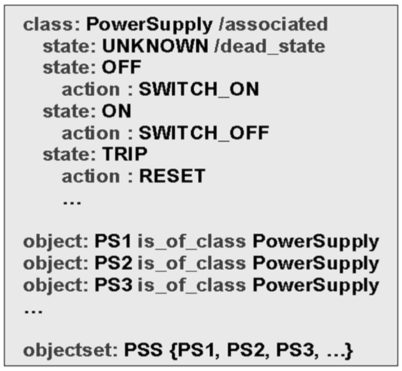


Fig. 2. SMI++ run-time environment.

SMI++ domains and their objects are described using a simple language: SML- the State Management Language.

A. SML Language

Concrete objects are simply declared in SML, i.e., only their list of states and possible actions for each state are defined. The actual implementation is provided externally by a process



```
class: PowerSupply /associated
   state: UNKNOWN /dead_state
   state: OFF
      action : SWITCH_ON
   state: ON
      action : SWITCH_OFF
   state: TRIP
      action : RESET
      ...

object: PS1 is_of_class PowerSupply
object: PS2 is_of_class PowerSupply
object: PS3 is_of_class PowerSupply
...

objectset: PSS {PS1, PS2, PS3, ...}
```

Fig. 3. SML device declaration.

written in "C", or in "C++" or by a PVSSII script, if the device is implemented within PVSSII. Abstract objects are completely implemented in SML. Objects, concrete or abstract, can be grouped into "objectsets" to ease the manipulation of large number of objects. Fig. 3 shows an example declaration of several devices. In this example three power supplies are declared as being of class "PowerSupply". The three power supplies are then included in an "objectset" (called PSS) so that they can be manipulated as a group, when necessary.

In the class definition the qualifier "/associated" is used to mean that the actual implementation is done by an associated proxy. The "/dead_state" qualifier instructs any derived object to go to the specified state, in this case "UNKNOWN", if for any reason the respective proxy is not running or not reachable.

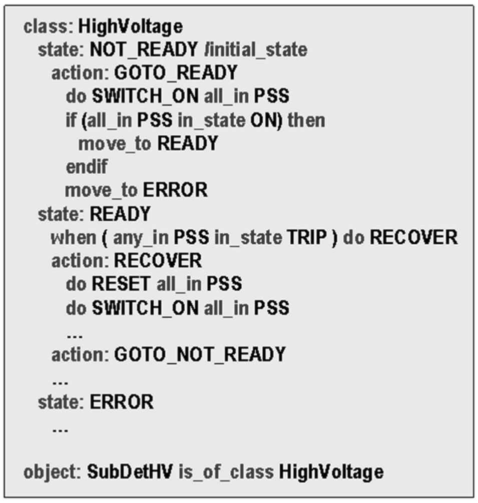
Abstract object description will contain not only the possible states and actions, but also the code that implements each action and the rules for asynchronous transitions.

The SML instruction set is reduced to a small number of simple and intuitive instructions.

The most important instructions are:

- "do" instruction: send a command to an object or to a set of objects;
- "if" instruction: test the state of an object or a group of objects;
- "move_to" instruction: end the action by moving to a new state.

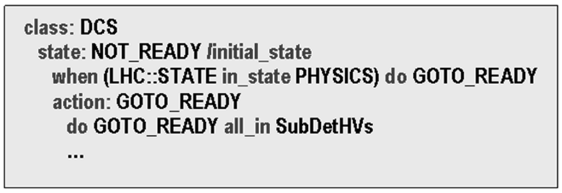
SMI++ objects are normally in a stable, well-defined, state, but they can temporarily be transiting from a state to another. An object is "transiting" between the time it received a command (via a "do" instruction) and the time it reached a new state (via a "move_to" instruction). While an object is transiting any "do" instructions it receives are queued and only processed after it reaches the new state. Also any "if" instructions involving the state of a transiting object will block until the object reaches a stable state, this new state is then used in the evaluation of the "if" condition. "Do" instructions, on the other hand are non-blocking. I.e. several consecutive "do" instructions will be sent in parallel to the various objects involved. The conjunction of the behavior of these two instructions allows sending "do" commands to several objects in parallel and waiting until they all respond using a single "if" instruction.



```
class: HighVoltage
  state: NOT_READY /initial_state
    action: GOTO_READY
      do SWITCH_ON all_in PSS
      if (all_in PSS in_state ON) then
        move_to READY
      endif
      move_to ERROR
  state: READY
    when ( any_in PSS in_state TRIP ) do RECOVER
    action: RECOVER
      do RESET all_in PSS
      do SWITCH_ON all_in PSS
      ...
    action: GOTO_NOT_READY
    ...
  state: ERROR
    ...

object: SubDetHV is_of_class HighVoltage
```

Fig. 4. SML object implementation.



```
class: DCS
  state: NOT_READY /initial_state
    when (LHC::STATE in_state PHYSICS) do GOTO_READY
    action: GOTO_READY
      do GOTO_READY all_in SubDetHVs
      ...
```

Fig. 5. SML asynchronous object automation.

For each state, before defining the possible list of actions, the user can define a list of rules, by means of "when" instructions. "when" instruction: while the object is in a given state whenever the condition gets fulfilled, execute an action ("do" action) or change state ("move_to" state).

- "When" instructions are the means of SMI objects to react to changes of other objects, i.e., to changes in their environment. Several rules can be listed consecutively, they are examined in order, if one of them triggers it will be immediately executed possibly provoking a change of state and therefore stopping the execution of the list of rules.

In the example of Fig. 4, a "HighVoltage" abstract object is modeled. It uses the group of power supplies (the PSS objectset) declared in Fig. 3 above, in order to send commands and to react to state changes of all the power supplies using simple instructions.

Objects can be dynamically included or excluded into/from a set by any object in the system by using two other instructions: "insert object in set" and "remove object from set". As a result the code example in Fig. 4 will act/react on all power supplies included in the set at the time of execution.

As mentioned before, objects can react to state changes of any other object in the system, via a set of "when" rules. This mechanism can be used to completely automate standard operations, without user intervention. In the example of Fig. 5 an object of class DCS is made to get all sub-detector's DCS systems ready whenever the object "STATE" in the "LHC" domain goes to state "PHYSICS". Allowing for the complete automation of the experiment based on the state of the LHC accelerator.

B. Tools

SMI++ offers a set of run-time tools to implement and deploy the control system. Some of the tools are the following.

- Smirtl—a run time library to be used by proxies in order to communicate with their respective SMI domain.
- smiSM—A logic engine which at startup reads the translated SML code and instantiates the objects of a given domain. There will be one smiSM process per SMI domain.
- Smiuirtl—A run-time library that can be used by client programs, in particular user-interfaces, in order to view the state of any object in the system and to send commands to them.

All three tools handle all necessary communications transparently, due to the use of an underlying communication package, called DIM [4]. Any of the components, SMI domains, proxies and user interfaces, can dynamically move from one machine to another on-the-fly (i.e., they can be stopped on one machine and restarted on another one), all communications are automatically reestablished and all necessary states recalculated. SMI++ tools are available on several platforms, including various UNIX flavors (in particular Linux) and Windows. The complete control system can be deployed across several (hundreds) heterogeneous machines.

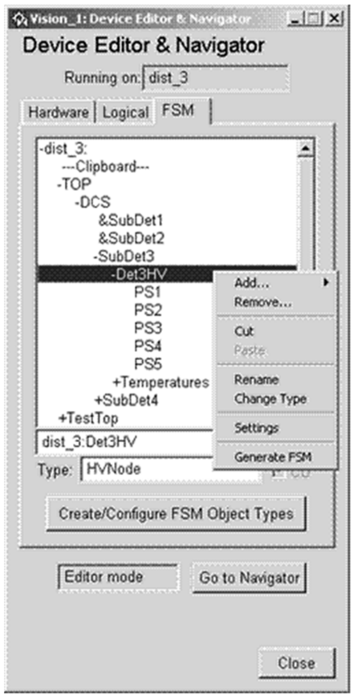


Fig. 6. Framework panel for creating hierarchies.

V. Framework Integration and Features

SMI++ has been interfaced to the PVSSII SCADA system, in order to provide a JCOP Framework component for Hierarchical Control.

PVSSII provides an API (Application Programming Interface) for external access. This API was used to create a PVSS process which provides the communication between the two packages: at run-time it transfers Proxy states from PVSS to the corresponding SMI Domain and Proxy commands back (using Smirtl) and it transfers Object states from the SMI Domains to PVSS (and Object commands back) for visualization and user control (using Smiuirtl). PVSSII also provides a very intuitive graphical editor which was used to develop both a configuration tool and an operation tool for the Hierarchical Control component.

A. Hierarchical Control

The Framework offers tools to implement a hierarchical control system. As described in the Architecture chapter above, JCOP defined a hierarchical control tree composed of two types of nodes: "Device Units" and "Control Units". These components match perfectly the SMI++ concepts: Device Units correspond to concrete objects implemented as proxies within PVSSII and Control Units correspond to SMI domains.

The integration of SMI++ with PVSSII provided for several new features:

A graphical user interface was created which allows the configuration of object types, declaration of states and actions, etc. and for the generation of SML code, actions and rules through the use of wizards. The hierarchical tree of components can also be configured graphically as shown in Fig. 6.

The PVSSII archiving mechanism can in addition be used to store state transitions and so be able to retrieve the time evolution and long-term statistics of object state changes.

The integration of the SMI++ paradigm with the JCOP Framework also provided for a clear definition of interfaces and task separation: the PVSSII implementation of device units in terms of scripts contains only basic actions, like "RESET" or "CONFIGURE", it has no intelligence concerning when or in which sequence they should be executed. The logic behavior in terms of sequencing or dependency on other objects in the system is described in SML and implemented by the SMI objects. The advantage is that if it is necessary to replace some hardware only the PVSSII part is affected. On the other hand if the logic behavior should change than only the SMI rules change.

B. Partitioning

As mentioned before, partitioning is the capability of monitoring and/or controlling a part of the system, a sub-system, independently and concurrently with the others in order to allow for tests, calibration, etc.

Each Control Unit knows how to partition "out" or "in" its children. Excluding a child from the hierarchy implies that its state is not taken into account any more by the parent in its decision process, that the parent will not send commands to it and that the owner operator releases ownership so that another operator can work with it.

It was felt that excluding completely a part of the tree was not flexible enough, so the following partitioning modes were defined and implemented in the Framework:

- **Included**—A component is included in the control hierarchy; it receives commands from and sends its state to its parent.
- **Excluded**—A component is excluded from the hierarchy, it does not receive commands and its state is not taken into account by its parent. This mode can be used when the

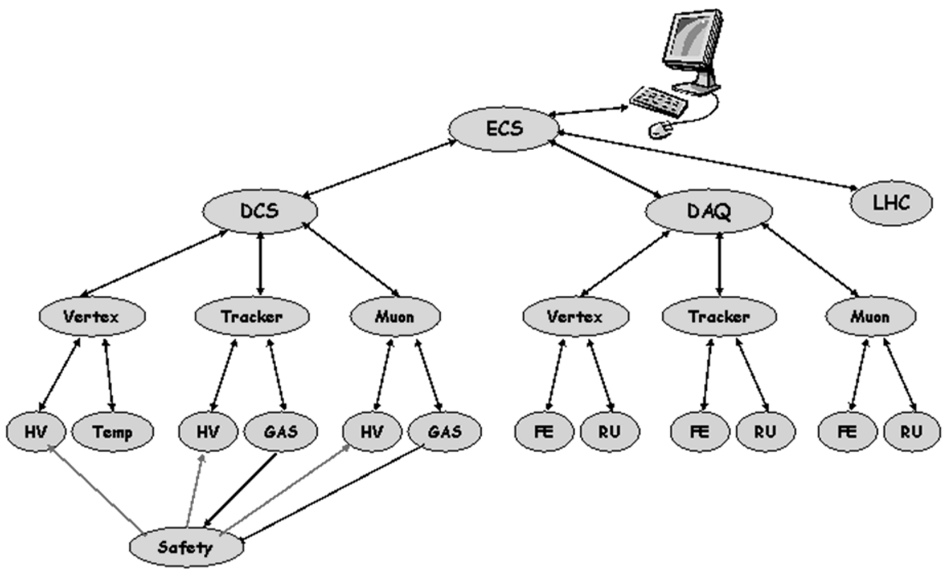


Fig. 7. Parallel control hierarchies.

component is either faulty or ready to work in stand-alone mode.

- **Manual**—A component is partially excluded from the hierarchy in that it does not receive commands but its state is still taken into account by its parent. This mode can be used to make sure the system will not send commands to a component while an expert is working on it. Since the component's state is still being taken into account, as soon as the component is fixed the operations will proceed.
- **Ignored**—A component can be ignored, meaning that its state is not taken into account by the parent but it still receives commands. This mode can be useful if a component is reporting the wrong state or if it is only partially faulty and the operator wants to proceed nevertheless.

The partitioning mechanism again fits perfectly with the SMI++ concept of object sets and was implemented using this feature: each child of a Control Unit can be part of two sets. A "Send Commands To" set and a "Take State into Account" set. The "Include" command will add it to both sets, the "Exclude" command will remove it from both sets, "Manual" and "Ignore" will add it only to the relevant set. Whenever a Control Unit wants to send commands to its children only the ones in the set will be affected, and the same for computing the state of the Control Unit. This functionality was encapsulated in a SMI++ Partitioning Object which is automatically inherited when a Control Unit is created.

C. Distribution

Both PVSSII and SMI++ allow for the implementation of large distributed and decentralized systems. There is no rule for the mapping of Control Units and Device Units into machines, i.e., there can be one or more of these units per machine depending on their complexity, or other factors such as development teams they "belong" to. The Framework allows users to describe their system and run it transparently across several computers. Since both tools can run on mixed environments comprising Linux and Windows machines, the user can also choose the best platform for each specific task.

D. Error Handling

Error handling is the capability of the control system to detect errors and to attempt recovery from them. It should also inform and guide the operators and to record/archive the information about problems for maintaining statistics and for further analysis offline.

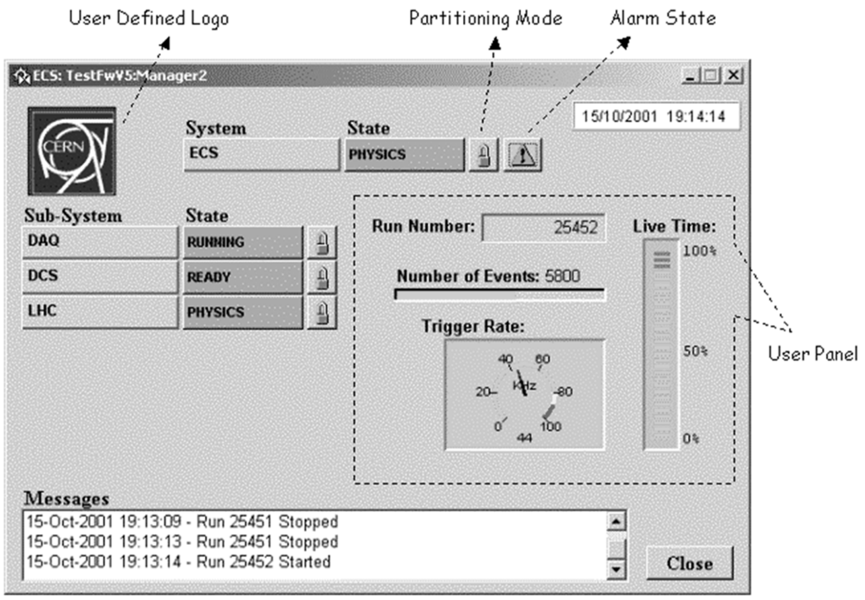
Since SMI++ is also a rule-based system, errors can be handled and recovered using the same mechanism used for "standard" system behavior. There is no basic difference between implementing rules like "when system configured start run" and "when system in error reset it". The recovery from known error conditions can be automated using the hierarchical control tools based on sub-system's states. In conjunction with the error recovery provided by SMI++ full use is made of the powerful alarm handling tools provided by PVSSII for allowing equipment to generate alarms (possibly using the same conditions that generate states), for archiving, filtering, summarizing and displaying alarms to users and to allow users to mask and/or acknowledge alarms.

E. Automation

By integrating SMI++, the Framework tools can provide for complete automation of a large control system. SMI++'s mechanism for automation of procedures and for automated error-recovery is quite suited for large systems: The recovery mechanism is the following.

- Bottom up—each object reacts in an event-driven, asynchronous, fashion to changes of its children.
- Distributed—each sub-system recovers its own errors and automates procedures for its sub-tree. For large physics experiments this is an advantage, since each team knows best how to handle their equipment.

This decentralized approach is inherently scalable, since there is no centralized expert system examining all faults in



15/10/2001 19:14:14

System	State
ECS	PHYSICS

Sub-System	State
DAQ	RUNNING
DCS	READY
LHC	PHYSICS

Run Number: 25452

Number of Events: 5800

Messages

```
15-Oct-2001 19:13:09 - Run 25451 Stopped
15-Oct-2001 19:13:13 - Run 25451 Stopped
15-Oct-2001 19:13:14 - Run 25452 Started
```

Fig. 8. Prototype run control interface.

the system, which could provoke a bottleneck. Furthermore it allows for parallel recovery, for example, if there is a general power cut, each sub-system can start recovering in parallel when the power comes back.

The Framework tools allow building a completely automated hierarchy based on the states of the devices composing the experiment and on the states of external elements like the LHC accelerator. In the example of Fig. 7, the top part of the figure represents a typical hierarchy for a physics experiment. But it is also possible to build parallel hierarchies, the bottom part of Fig. 7, shows an example of a “safety” hierarchy that would contain code like: “When any gas system in error cut all high voltages in the experiment”.

F. System Operation and Run Control

The Framework provides configurable operation panels. These panels are automatically generated and will have predefined areas showing the states of the hierarchical components, their partitioning modes, their alarm states, etc. and user defined areas that are specific to the task of that particular component. The user can navigate through the hierarchy by clicking on the different components.

The panel showing the component at the top of the hierarchy provides a high-level, summarized, view of the complete underlying system.

The main interface to a physics experiment is normally called the “Run Control”. A simple Run-control panel is shown as an example in Fig. 8.

The operation of the different sub-systems, or complete subdetectors when working in stand-alone mode, is based on the same tools and will provide similar interfaces.

VI. Conclusions

The SMI++ framework is a powerful tool which, while merging the concepts of object modeling, finite state machines and rule-based reasoning, allows the implementation of homogeneous, integrated and fully automated control systems. Its inherent capabilities of distribution and scalability over large sets of heterogeneous platforms makes it extremely well-suited to control very large applications.

The SMI++ framework has become a time tested, robust tool through its use by major particle physics experiments: the DELPHI experiment at CERN in the recent past and the BaBar experiment at SLAC, which is currently using it in production.

SMI++ has been integrated with the SCADA tool PVSSII in order to provide extra functionality, like graphical tools and standardized partitioning modes, and is now being used by the four LHC experiments at CERN for the design of either full or partial experiment control.

References

[1] A. Daneels and W. Salter, “The LHC experiments joint controls project, JCOP,” presented at the International Conference on Accelerator and Large Experimental Physics Control Systems Trieste, Italy, 1999.

[2] S. Schmeling, “Controls framework for LHC experiments,” presented at the 13th IEEE-NPSS Real Time Conference Montreal, QC, Canada, May. 18–23, 2003.

[3] *PVSS-II*, [Online]. Available: http://www.pvss.com.

[4] B. Franek and C. Gaspar, “SMI++—An object oriented Framework for designing distributed control systems,” *IEEE Trans. Nucl. Sci.*, vol. 45, no. 4, pp. 1946–1950, Aug. 1998.

[5] C. Gaspar and M. Dönszelmann, “DIM, a portable, light weight package for information publishing, data transfer and inter-process communication,” (in Ph. Charpentier) *Computer Physics Communications*, vol. 140, no. 1+2, pp. 102–109, Oct. 2001.