SMI++ Object Oriented Framework for Designing and Implementing Distributed Control Systems

B. Franek[2] and C. Gaspar[1]
[1] CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
[2] Rutherford Appleton Laboratory, Chilton, Didcot, OX11 0QX, Great Britain

Abstract

In the SMI++ framework, the real world is viewed as a collection of objects behaving as finite state machines. These objects can represent real entities, such as hardware devices or software tasks, or they can represent abstract subsystems. A special language (SML) is provided for the object description. The SML description is then interpreted by a Logic Engine (coded in C++) to drive the Control System. SMI++ objects can run in a variety of platforms all communication being handled transparently by an underlying communication system - DIM . This framework has been used by the DELPHI experiment at CERN [1] for the experiment control. A significantly upgraded version is now being used by Babar experiment at SLAC [2].

I. INTRODUCTION

SMI++ is based on the original State Manager concept [3] which was developed by the DELPHI experiment in collaboration with the DD/OC group of CERN.

In this concept, the real-world system to be controlled and the control system to be designed is described in terms of objects behaving as Finite State Machines. Objects can represent concrete entities, for example an hardware device or abstract entities like a logical sub-system. The objects representing concrete entities interact with the hardware they model and control through associated processes or proxies.

The main attribute of an SMI object is its state. Commands sent to an object trigger actions that can bring about a change in its state. Objects can also spontaneously respond to state changes of other objects. The objects are typically organised in hierarchical structures called domains. A fully automated control system can be achieved by a top-level domain controlling all underlying domains.

The object model of the real-world and control system is described using State Manager Language (SML). This language allows detailed specification of the objects such as their states, actions and associated conditions. The SML code is parsed and translated into a database that is then used by a generic 'Logic Engine'. Each logic engine drives an SMI Domain.

The logic engine has been designed using an Object Oriented design tool (Rational Rose/C++) [4] and coded in C++ language. It uses the translated SML representation to instantiate the required objects and then responds to external events to drive the object model of the control system. It is often advantageous to build the application as a collection of cooperating Logic Engines running on different computer platforms. The communication between SMI Domains is embedded in the SMI system. All issues related to distribution and heterogeneity of platforms are transparently handled by the underlying communication system - DIM [5].

SMI++ and DIM also offer a set of graphical tools to implement, configure, test and monitor the control system.

A previous version of this concept (SMI) has been used by the DELPHI experiment at CERN (since 1990) to control subsystems with different constrains such as safety critical or real-time. The full experiment is controlled using this mechanism up to complete automation. It is a complex system consisting of 1000 objects organized in 50 SMI domains and distributed over 40 machines [6]. SMI++ is also being used by the BaBar experiment at SLAC for their design of run control.

II. SMI++

SMI++ is a framework for designing and implementing distributed control systems. It provides:

- A method based on a special language to describe the controlled world and to code the control logic.
- A set of tools to implement and test the control system.

The SMI++ method combines two concepts: Objects and Finite State Machines (FSM). The real world to be controlled, being typically a set of hardware devices and software tasks, is decomposed into well defined entities (objects). The objects behave as Final State Machines. The control system is equally conceived as a set of abstract (or logical) objects behaving as FSM's. These objects contain the control logic and can send commands to other objects

In order to reduce complexity of large systems, logically related objects are grouped into SMI domains. In each domain, the objects are organised in a hierarchical structure and form a subsystem control. Usually only one object (top level object) in each domain is accessed by other domains. The final control system is then constructed as a hierarchy of SMI domains.

Proxy processes provide the abstractions of the hardware components in the SMI world and implement the actual actions on the hardware.

User Interfaces allow the control and visualization of the system.

This framework allows an easy reconfiguration of the system: changes in the hardware can be easily integrated by modifying or replacing proxies and modifications in the control logic by

changing the SML code. The decoupling between the actual actions on the hardware (done by the Proxy Processes) and the control logic (residing in the SMI objects) makes the evolution of a system from its first test phase up to the final complexity a very smooth process. The basic concepts of the framework are graphically outlined in Figure 1.

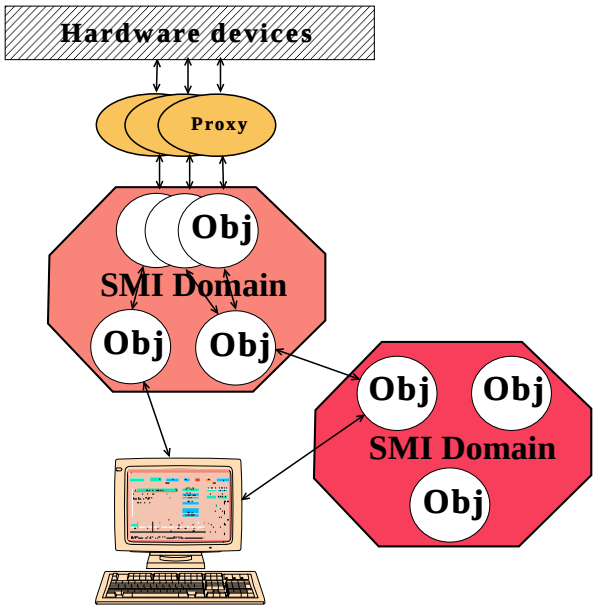

Fig. 1 Basic concepts of SMI++

A. State Manager Language SML

The object model of the control system is described using State Manager Language (SML). This language allows detailed specification of the objects such as their states, actions and associated conditions. The main characteristics of this language are :

- Finite State Logic

Objects are described as finite state machines. The only attribute of an object is its state. Commands sent to an object trigger actions that can bring about a change in its state.

- Sequencing

An action on an abstract object is specified by a sequence of instructions, mainly consisting of commands sent to other objects and logical tests on states of other objects. Actions on objects representing hardware components are sent off as messages to the Proxy Processes.

- Asynchronous

Several actions may proceed in parallel: a command sent by object-A to object-B does not suspend the instruction sequence of object-A. Only a test by object-A on the state of object-B suspends the instruction sequence of object-A until object-B reaches a stable state.

- AI-like rules

Each object can specify logical conditions based on states of other objects. These when satisfied will trigger an action on the local object. This provides the mechanism for an object to respond to unsolicited state changes of its environment.

```
!- Example of SML code

object : RUN_CONTROL
  state : READY
    action : START_RUN
      do MOUNT TAPE
      if TAPE not in_state MOUNTED
        do MOUNT_ERROR ERROR_OBJ
        terminate_action/state=ERROR
      endif
      do START READOUT_CONTROLER
      if READOUT_CONTROLER in_state RUNNING
        terminate_action/state=
                          RUN_IN_PROGRESS
      ...
  state : RUN_IN_PROGRESS
    when TAPE in_state FILE_FULL
      do PAUSE_RUN
    when READOUT_CONTROLER in_state ERROR
      do ABORT_RUN
    action : ABORT_RUN
      ...

object : READOUT_CONTROLER/PROXY
  state : READY
    action : START
    ...
  state : RUNNING
    action : PAUSE
    action : ABORT
    ...
```

B. Tools

The SMI framework provides a set of tools to generate and implement the control system, as shown in Figure 2.

1) SML translator

The description and behaviour of the objects coded in SML is parsed by the SML translator. It also translates the code into a Domain Description intermediate file which is then used at run-time by the State Manager (see below) to drive the model of the Domain.

2) Proxy generator

From the SML code it generates skeletons for all the proxies in the domain. The user can then plug-in specific code to actually

drive the hardware and link with an SMI++ run time library to produce the proxy.

3) GUI

A generic User interface is also provided. It allows the control and monitoring of the objects in a specific domain. The states of the objects can be visualized and commands can be sent to each object. The interface is configurable; the objects can be selectively displayed, moved around the display and different colors can be selected for different states of the objects.

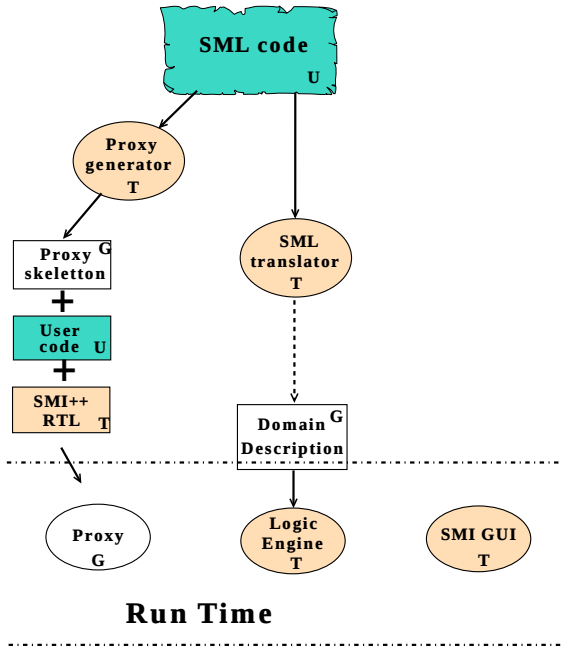

Fig. 2 SMI++ Tools

4) State Manager process

This is the key tool of the SMI++ framework. At run-time, it organizes and synchronizes activities performed by the independent hardware components assigned to the domain and possibly objects in other domains. It does this by using the translated SML code for the domain. It responds to external events and 'drives' the control system by following the coded control logic and sending the necessary commands to proxies and objects in other domains. It was designed using an Object Oriented design tool (Rational Rose/C++) and coded in C++. It's main C++ classes are shown in Figure 3. They are grouped into two class categories :

- SML Classes

These classes represent all the elements defined in the language such as - states, actions, instructions etc. They are all contained within the *SMIObject* class (representing SMI objects). At the startup of the process, they are instantiated from the translated SML code.

- Logic Engine Classes

Based on external events, these classes 'drive' the instantiations of the language classes.
CommHandler takes care of all the communication issues. It detects state changes in remote SMI objects and 'feeds' the state queue *(StateQ)*. It receives external actions coming from remote objects or from an operator and 'feeds' the relevant queue *(ExternalActionQ)*. It also communicates the state changes in local SMI objects to the outside world and sends commands from local SMI objects to remote objects.
Scheduler takes the information from the state and action queues and operates on the SMIObject instantiations in such a way that in effect each local object executes its own thread.

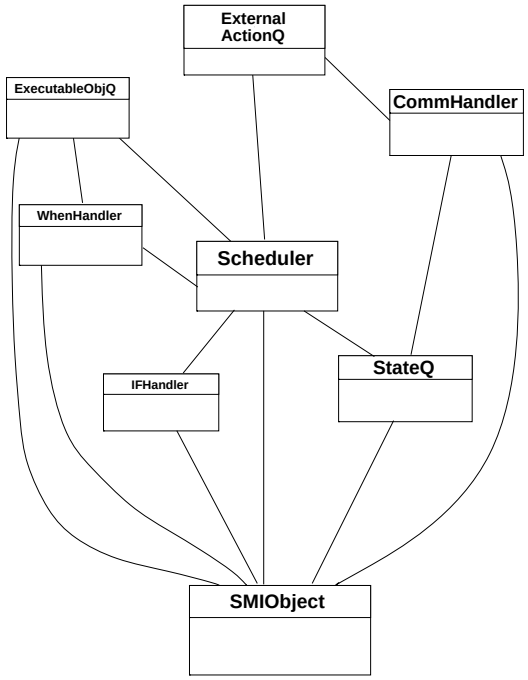

Fig. 3 State Manager's main classes

III. DISTRIBUTED ENVIRONMENTS

Current computer control systems have very often a highly distributed architecture consisting of workstations interconnected by a local area network.

SMI takes advantage of distribution, SMI Domains can run on a variety of computer platforms. The cooperation between SMI Domains including all exchanges between objects, are embedded in the SMI system. All issues related to distribution and heterogeneity of platforms are transparently handled by the underlying communication system -DIM (Distributed Information Management System) .

DIM's aim is to provide interoperability between

applications on different machines in heterogeneous distributed environments.

The DIM system was designed and implemented according to the following characteristics :

- Efficiency

The communication mechanism of DIM was chosen having in mind the asynchronous character of SMI objects and the speed in reacting to changes or error conditions in the system. The solution we thought the best is for clients to declare interest in a service provided by a server only once (at startup), and get updates at regular time intervals or when the conditions change. I.e. an asynchronous communication mechanism allowing for task parallelism and multiple destination updates.

- Transparency

At run time no matter where a process runs, it is able to communicate with any other process in the system independently of where the processes are located. Processes can move freely from one machine to another and all communications are automatically reestablished. (this feature also allows for machine load balancing).

- Reliability and Robustness

In an environment with many processes, processors and networks, it often happens that a process, a processor or a network link breakes down. The loss of one of these items should not perturbate the rest of the application. DIM provides an automatic recovery from crash situations or the migration of processes.

DIM uses a publish/subscribe mechanism. Any process can publish (Server) information and any interface (or any other process) can subscribe (Client) to this information. A unit of information is called a ”Service”. A Name Server keeps track of all the Servers and Services available in the system.

Servers ”publish” their Services by registering to the Name Server (Normally once at startup).

Clients ”subscribe” to Services by asking the Name Server which Server provides the Service and then contacting directly the Server. Client’s Services are then kept up-to-date in an event driven mode or at regular time intervals. Clients can also send commands to servers.

DIM is responsible for most of the communications inside the DELPHI Online System, it is used by SMI in order to transfer object states and commands, by the user interfaces in order to access SMI or any other necessary information and by many other processes for monitoring or processing activities. In the DELPHI environment it makes currently available around 30000 Services provided by 450 Servers. Dim is also being used by other experiments at CERN .

IV. SMI’S USE IN DELPHI

In DELPHI the full online system is controlled through this mechanism, the various areas of DELPHI have been mapped into SMI domains: sub-detector domains, DAS domain, SC domain, TRIGGER domain, etc. The full system comprises about 1000 SMI objects in 50 different domains and running on 40 machines.

A high level of automation of the experiment’s control system is very important in order to avoid human mistakes and to speed up standard procedures.

Using the SMI mechanism the creation of a top level domain - BIG BROTHER - containing the logic allowing the interconnection of the underlying domains (LEP, DAS, SC, etc.) was a relatively easy task.

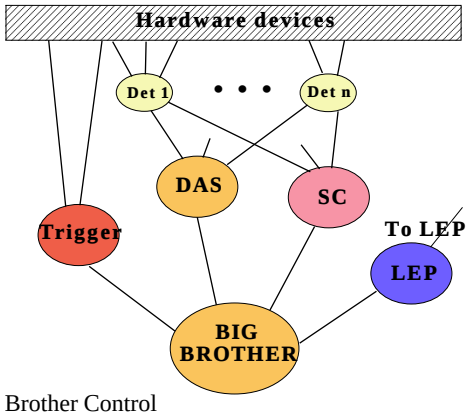

Fig. 4 Big Brother Control

Under normal running conditions BIG BROTHER pilots the system with minimal operator intervention as shown in Figure 4. In other test and setup periods the operator becomes the top-level object and using the user-interfaces he can send commands to any SMI domain.

V. IMPLEMENTATION AND AVAILABILITY

The first prototype of SMI++ was completed in June 1997. In July 1997 it has been extensively tested in DELPHI environment. During that time, the DELPHI experiment control was fully converted from the ’old’ version of SMI to the upgraded version SMI++.

SMI++ is available on any mixed environments comprising : VMS (VAX and ALPHA) and UNIX flavors (OSF, AIX, HPUX, SunOS, Solaris)

DIM is already available in the above platforms and on OS9,LINUX and LynxOS and is being ported to WindowsNT and VXWorks.

Other available tools are :

A DIM Display allowing the visualization of all the servers and clients in a certain DIM environment (including SMI and driver processes). Very useful for debugging applications.

A DIM to WWW gateway, allowing access to all DIM services (including SMI states). The WWW page can be written in HTML with specific DIM tags containing the service name. The DIM tags are translated when the page is loaded.

VI. CONCLUSIONS

SMI is a powerful tool for designing and implementing control systems, it merges the concepts of object modeling and finite state machines.

The SMI system provides a simple language to model the

application and a set of tools to compile, configure and run your applications on a variety of platforms.

The full control of the DELPHI experiment at CERN is implemented using this system, SMI proved capable of handling the control of different environments such as: data acquisition (including run control), slow controls, trigger, etc.

Due to the homogeneity in the control of the different parts of DELPHI it was possible to interconnect the different parts and completely automate the DELPHI operations. It also considerably reduced the efforts on maintenance and upgrade of the complete control system of DELPHI.

SMI++ implements extensions to the SMI concept and was re-designed for use by the BaBar experiment at SLAC. The main extensions visible to a user are related to more configuration capabilities at run-time, availability on a larger set of platforms (including heterogeneous distributed environments) and a higher support on graphical tools.

VII. ACKNOWLEDGMENTS

We would like to thank some of our colleagues at CERN for fruitful discussions. In particular to Ph. Charpentier, M. Jonker, P. Vande Vyvre and A. Vascotto.

VIII. REFERENCES

[1] DELPHI Collaboration, Aarnio, P. et al. (1991). The DELPHI Detector at LEP. In: *Nuclear Instruments and Methods in Physics Research A303*, pp. 233-276.

[2] BaBar Technical Design Report SLAC-R-95-457 March, 1995

[3] J. Barlow et al.(1989). Run Control in MODEL: The State Manager *IEEE trans.nucl.sci.36*, pp. 1549-1553.

[4] Rational Rose/C++, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, USA

[5] Gaspar, C. and Dönszelmann, M. (1993). DIM - A Distributed Information Management System for the DELPHI experiment at CERN. In: *Proceedings of the IEEE Eight Conference REAL TIME '93 on Computer Applications in Nuclear, Particle and Plasma Physics*. Vancouver, Canada.

[6] T. Adye at al. (1992). The DELPHI Experiment Control *Proceedings of the International Conference on Computing in High Energy Physics '92*. Annecy, France.