The State Manager: A Tool to Control Large Data-Acquisition Systems

Ange Defendini, Robert Jones, Jean-Pol Matheys, Pierre Vande Vyvre, Alessandro Vascotto
CERN, Geneva, Switzerland

Abstract

The State Manager system (SM) is a set of tools, developed at CERN, for the control of large data-acquisition systems. A dedicated object-based language is used to describe the various components of the data-acquisition system. Each component is declared in terms of finite state machines and sequences of parametrized actions to be performed for operations such as the start and end of a run. The description, written by the user, is translated into Ada to produce a run-control program capable of controlling processes in a distributed environment. A Motif-based graphical interface to the control program displays the current state of all the components and can be used to control the overall data-acquisition system. The SM has been used by several experiments both at CERN and other organizations. We present here the architecture of the SM, some design choices, and the experience acquired from its use.

I. INTRODUCTION

Today's large data-acquisition systems are composed of an increasingly large set of programs which prove difficult to control. Furthermore, the different programs are not independent but co-operate and need to be synchronized: for example, they must be started and stopped in a given order. Finally, a system composed of many different programs is difficult to operate if one has to interact with each of these programs.

The SM [1,2] is a neat and flexible solution to this problem. It is a tool for building distributed run-control systems by means of a dedicated object-based language.

The system to be controlled is decomposed into a set of objects. Objects correspond to a part of the system: a program or a subsystem. Each object must then be described as a state machine, its main attribute being its current state. The state can take any value in a list of values declared by the user in his SM program. An object can interact with other objects by sending commands to them. The command triggers the execution of an action, which is terminated when the object reaches a new state.

Each activity of the data-acquisition system to be controlled should be handled by a single process. These processes are called associated processes because they are associated with an SM object. The SM communicates with them via messages handled by the OSP package [3]. The SM sends the commands triggering the execution of actions, and the associated processes reply when they assume a new state. These messages constitute the interface between the SM and the associated processes. The same interface is used by an overall control program to send commands to the SM itself as shown in Figure 1.

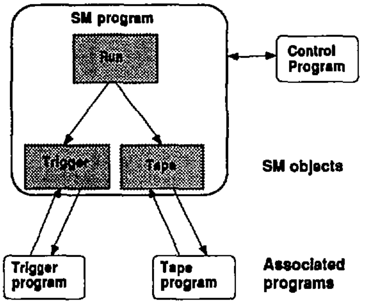

Figure 1. The SM program and the external world.

The communication package deals with distributed environments and thus allows commands to be sent to processes running on remote machines.

The objects are divided into two categories:

- The associated objects are associated with a program dealing with a device or an activity.

- The objects of the second category correspond to abstract entities that form part of the description of the system. They are internal to the SM.

The SM program written by the user is translated by the SM translator into Ada [4]. This Ada code is then compiled and linked to produce an executable image. The execution of this image will activate the run control and establish the communication with the associated processes.

II. THE SM LANGUAGE

A. Object declarations

The SM language contains declarations and instructions. The declarations are used to define the name of an object, its states, and actions. An example of a state machine for an object 'RUN' is given in Figure 2.

The corresponding SM declarations are:

```
object : RUN
    state : DORMANT
        action : START
    state : ACTIVE
        action : PAUSE
        action : STOP
    state : PAUSED
        action : RESUME
```

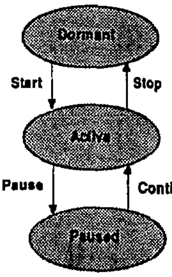

Figure 2. Example of a state machine.

The execution of an action is different, depending on whether the object is associated or not. For an associated object, the execution of an action consists of sending a command to its associated process. When the process has terminated the execution of this action, it will return to its new state, which is mimicked by the associated object. For a normal object (not associated), the execution of the action consists of the execution of a sequence of SM instructions.

B. The instructions

Four basic instructions are contained in the language.

The DO instruction is used to send a command asynchronously to an object. The sender carries on with its own execution after the command has been sent. The command is put into a queue if the receiver is not ready to execute. A queue of pending commands is maintained for each object in the system. The next command is delivered to an object when this object is ready to accept it, i.e. when the object is in a stable state and is not executing any action.

The sequence of instructions corresponding to an action is terminated by the instruction 'TERMINATE_ACTION /STATE=state name'. This instruction can be placed anywhere in the code, thus stopping the execution of the code and putting the object in a stable state specified in the instruction.

The IF instruction tests the state of one or many objects and combines the results in a logical expression. All the commands present in the object queue must be executed before testing the object state. The IF instruction synchronizes the object executing the IF statement with the objects whose state is tested in the instruction. The language also specifies a special state, the 'dead state', which is assumed by an associated object when the program associated to it is not running. This special state allows testing in the SM code whether the associated program is running or not.

The WHEN instruction triggers the execution of an action spontaneously when a logical expression based on the states of objects becomes true. This instruction is used to react asynchronously to a state change in the system.

The example below uses the four basic SM instructions:

```
object : RUN
    state : DORMANT
        action : START
            do MOUNT TAPE
            do START TAPE
            do ENABLE TRIGGER
            if (TAPE in_state WRITING) and
                (TRIGGER in_state ENABLED) then
                terminate_action/state=ACTIVE
            else
                terminate_action/state=FAILURE
            endif
    state : ACTIVE
    when TAPE in_state END_OF_TAPE do STOP
        action : STOP
            do DISABLE TRIGGER
            do STOP TAPE
            do DISMOUNT TAPE
            terminate_action /state=DORMANT
    state : FAILURE
        action : RESET
            do RESET TAPE
            do RESET TRIGGER
            terminate_action /state=DORMANT
            ...
```

C. The SM domain and the visible objects

The object name has to be unique in one SM program because the object must be addressable unambiguously. However, this may be a limitation in big systems composed of the repetition of similar subsystems. It may also be easier to divide a big system into smaller SM programs. This is what the SM domains are for (Fig. 3). The SM domain is a logical domain that consists of one, and only one, SM program and its associated programs. The SM domain limits the visibility of an object. The name of the object must be unique in one domain, but the same object name can be used in different domains.

An object belongs to one domain only, but it may be rendered 'visible' to outside domains. One SM program can therefore be controlled from another SM program. Figure 3 shows an example of a top-level SM controlling two other SMs in different domains.

The way to invoke an object of another domain is to specify explicitly its domain as shown in the example of Figure 3: the SM program of the domain MAIN contains references to the objects 'TPC::RUN' and 'HCAL::RUN'.

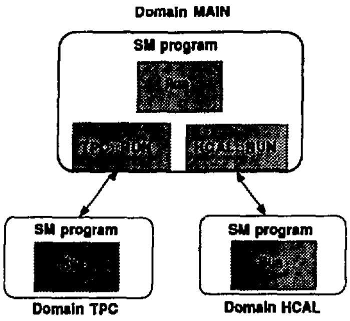

Figure 3. Example of use of the SM domain.

D. The classes and the access objects

With the declaration statements seen up to now, the user has to declare each object in the SM program. However, it is quite common to have systems where many objects are identical, but not their names. The notion of class has been introduced to cope with this case. A class of objects is similar to a data type in standard programming languages. The declaration of a class must define the class name, its states, and its actions. Once a class has been defined, it is sufficient to invoke the class name in the declaration of the objects that belong to that class. The following example shows the declaration of a class 'TAPE', and of two objects 'TAPE1' and 'TAPE2' of the class 'TAPE':

```
class : TAPE /associated
   state : AVAILABLE
      action : MOUNT
   state : MOUNTED
      action : DISMOUNT
object : TAPE1 is of class TAPE
object : TAPE2 is of class TAPE
```

This feature improves the readability and the 'maintainability' of the code, and reduces the size of the declarative part of the program.

A special type of object has been introduced to handle an object belonging to a class: the access object, which is like a pointer to any object of a given class. The 'access' statement specifies which object of the class is being accessed. The basic SM instructions can use the access object to refer to one object of a class indirectly:

```
object : CURRENT_TAPE is access to class TAPE
      state : NOT_USED
         action : SELECT_TAPE1
            access TAPE1
            ...
   do MOUNT CURRENT_TAPE^

   if (CURRENT_TAPE^ in_state MOUNTED) then
   ...
```

E. Command parameters

Some associated programs may need parameterized commands. Parameters are specified in the SM code as simple strings or as logical names translated at run-time. The parameters are appended to the string of the command before it is sent to the associated process. An example is given below.

```
do MOUNT ("/LABEL=" VOLUME) CURRENT_TAPE^
```

III. THE STATE MANAGER AND THE EXTERNAL WORLD

A. The control program

The SM program itself can be controlled at run-time by a 'control program', which can send a command to the SM with a call to a subroutine. The control program can also examine the current state of the system. A library of routines is available for the communication between the control program and the SM.

A general purpose control program has been built using the Motif graphical user-interface toolkit. Each object is shown on the screen as an icon. The user interacts with an object by clicking on its icon to reveal a popup menu. The user can send a command to the object, see the object's past states and actions, and view the queue of actions to be performed by an object. The display shows one domain at a time. This domain can be selected by the user.

Many aspects of the display can be customized by the user; objects icons can be hidden, moved and replaced by user-defined ones. The user's personal configuration can be saved and restored automatically when the display program is restarted.

B. The associated programs

The associated programs running under the control of the SM must conform to a well-defined interface. They must be command-driven and send back their state when it has been modified. A library of routines is available for the communication between the associated processes and the SM.

The simplest structure of an associated program is as follows:

```
C  Initialization call
   call SMI_INIT
C  Associate the program with an object
   call SMI_ASSOCIATE (object_name)
   do while (program active)
C       Receive next command to execute
        call SMI_GET_COMMANDW (command)
C       Decode the command
C       Return new state after execution
        call SMI_TERMINATE_COMMAND (state)
   end do
```

C. The multiple-state associated objects

Some associated programs may be difficult to describe in terms of a state machine with a unique current state. It is possible to divide them into sub-objects, each of which have their own state. A sub-object can be a device that the program has to deal with, or a level of alarm, and so on. The result of this division into sub-objects is that the object itself has many concurrent states

Extra SM instructions, not described above, are required to fully define object's state machines. A tool exists to help the user generate associated programs state machines.

IV. IMPLEMENTATION

The translation of the SM language into Ada proceeds directly by the semantics of its main instructions.

Each SM object executes its actions in parallel with the other objects. Each object is translated into an Ada task which is an independent thread of code. The current state of an object becomes a variable with a value equal to one of the value of an enumerated type. Actions are translated into Ada rendez-vous, where two tasks are synchronized.

However, the SM action is asynchronous whereas the Ada rendez-vous is synchronous. Therefore, for each object, a second Ada task is introduced to handle the queue of pending actions, to disable its access and to produce the asynchronism needed for the SM action. The two Ada tasks corresponding to an object are shown in Figure 4.

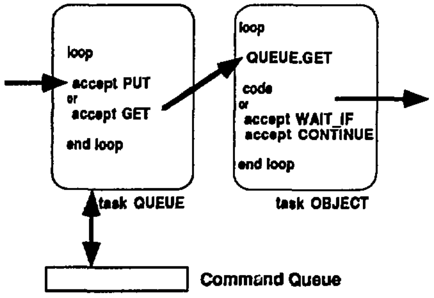

Figure 4. The Ada code corresponding to an object.

As described before, the IF instruction requires some synchronization between the object executing the IF statement and the objects whose states are evaluated in the condition. Two rendez-vous of the task 'Object' block and unblock the execution of an object at the beginning and the end of the IF statement.

For example consider the following SM instructions.

```
object : RUN
...
        do MOUNT TAPE
        if (TAPE in_state MOUNTED) then
```

The DO instruction is translated into a rendez-vous with the QUEUE task to add an action to the queue of the destination object. The IF instruction causes the TAPE object to block, its state to be evaluated by Run, and then to unblock.

```
QUEUE(TAPE).PUT(MOUNT)
QUEUE(TAPE).PUT(WAIT_IF)
if (OBJECT(TAPE).STATE=MOUNTED) then
endif;
OBJECT(TAPE).CONTINUE
```

In addition, a dedicated Ada task consists of all the WHEN instructions contained in the program. This task is scheduled each time an object assumes a new state. The task evaluates all the WHEN conditions and adds the appropriate actions to the queues.

V. CONCLUSION

The SM is a new approach to the problem of run control. It has proved to be both flexible and reliable, during its use at CERN in collaborations such as DELPHI, OBELIX, and Omega.

By coding associated programs according to simple principles, SM provides an object-based approach to DAQ design that benefits the control and maintenance of the system.

VI. REFERENCES

[1] J. Barlow et al., "Run Control In Model: The State Manager", in *Proc. 6th Conf. on Real-Time Computer Applications in Nuclear, Particle, and Plasma Physics*, Williamsburg, Virginia, 1989 [IEEE Trans. Nucl. Sci. NS-36 (1989) p. 1549]

[2] A. Deffendini, B. Franek, P. Vande Vyvre and A. Vascotto, The state manager user manual, CERN-ECP internal note, Geneva. This document can be obtained, upon request, from the authors, c/o CERN.

[3] R. Jones, OSP user's guide, CERN-ECP internal note, Geneva. This document can be obtained, upon request, from the author, c/o CERN.

[4] Ada Joint Program Office, United States Department of Defence, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, Washinton D.C.: *Government Printing Office*, 1983.