Run Control in Model: the State Manager

*J. Barlow
B. Franek
M. Jonker
T. Nguyen
P. Van De Vyvre
A. Vascotto*

[REDACTED]
CERN LIBRARIES, GENEVA

CM-P00059923

Presented at the "REAL TIME 89 - Sixth Conference on
real-time computer applications in nuclear, particle and plasma Physics"
Williamsburg, Virginia - May 16-19, 1989

RUN CONTROL IN MODEL: THE STATE MANAGER

J.Barlow, B.Franek, M.Jonker, T.Nguyen, P.Vande Vyvre, A.Vascotto

On-line Computing Group, Data Handling Division
CERN, 1211 Geneva 23, Switzerland

Abstract

The MODEL software is a set of modules for on-line applications, running principally on VAX-family computers. It provides data flow, human interface, process control and error-reporting facilities. Recently, facilities have also been developed to tackle the complex problem of controlling the various activities that constitute the data-acquisition system of a large physics experiment. The approach adopted is based on a State Manager. The physicist describes the experiment in terms of *objects*, i.e. logical subsystems, for each of which a number of states are defined. Commands can be sent to these objects, causing them to perform actions and to change state. The complete description of all objects, states and actions, in a simple language, is used to generate a State Manager for the experiment, which runs as a VMS process.

INTRODUCTION

MODEL [1] is the name of a range of software modules aimed at providing a software environment for data acquisition in High Energy Physics experiments. Originally started as a contribution to the on-line systems of the four LEP (Large Electron Positron Collider) experiments, MODEL is general enough to be applicable to a wide range of other experiments. Facilities provided by MODEL include data-flow organization, error reporting, human interface, process control, and run control. The latter is the subject of this paper.

MOTIVATION

The control of data acquisition in a High Energy Physics experiment is often a difficult problem to solve, in view of the complexity of the various operations, which involve a mixture of hardware and software. In particular, what is usually called *run control* implies the execution and synchronization of the various procedures needed to begin and end a period of data taking under stable conditions. The requirements not only vary widely from experiment to experiment, but may need to be substantially modified within a particular experiment as the setting-up proceeds and experience is gained. Other aspects, such as calibration or cold start of the apparatus, pose similar problems.

In general, any change in working conditions requires interventions in various parts of the data-acquisition system; these normally consist of lists of actions to be tediously executed by the operators running the experiment.

Within the MODEL framework, the approach to the run-control problem is based on the State Manager: the control aspects of an experiment are simulated by a finite-state machine, which represents the model of the equipment being controlled.

STATE MANAGER CONCEPTS

The State Manager (SM) is in essence a computer-based control system for the experiment. The experiment (a set of hardware devices and software programs) is seen by the SM exclusively through computer processes, dedicated to one specific activity, called **associated processes** (fig. 1).

The basic action by which the SM can control activities in the experiment is the exchange of messages with the associated processes. This constraint allows the SM to have a unique interface with the outside world: the specific control mechanisms are dealt with by the associated processes.

The SM may be driven by command messages originated by other processes, called **control processes**. The use of an interactive control process is the normal way by which an operator can interact with the SM.

Figure 1: The State Manager environment

The SM incorporates a model of the different activities to be organized, which is essentially a description of the experiment. The problem of creating a control system is therefore equivalent to that of giving a good description of the experiment in terms of the objects being controlled and the procedures that operate on them. The description of the experiment determines the flow of control generated by the SM; in other words it represents the behaviour of the whole system.

It is important that the end users (the physicists running the experiment) be able to generate the description of the experiment themselves, without the intervention of specialists. In view of this requirement, a **simple language** has been designed and a compiler called the **Translator** is provided.

Object/message model

The approach adopted for modelling the experiment is based on an object-oriented decomposition [2]. The experiment is described in terms of **objects**: an object may correspond directly to a concrete entity in the experiment (such as "gas valve 12"); it may equally represent a logical subsystem, or any abstraction used in describing the experiment provided it can be identified by a *noun* (e.g. "run", "trigger", "central detector").

Figure 2: An example of multi-threaded control

The object-based model closely matches our own model of reality; therefore the experiment can be described with the same concepts and terms as we use in thinking of it.

The main attribute of an object is its **state**, the only one that is visible by other objects. The state is a variable that can assume any one of a list of values enumerated by the users. The normal practice is to define values that correspond to *adjectives* which could qualify the name of the object (e.g. the

object "run" may be in one of the states "active", "dormant" or "paused").

An object operates on other objects by sending **command messages** to them. The communication mesh linking the objects carries the global flow of control in the system (fig. 2 shows an example). Again, this model of interaction between objects is one that reflects our abstraction of reality. It allows concurrence in the execution of operations and the development of many threads of control.

Object behaviour

Objects behave as finite-state machines. A change of state is brought about by the receipt of a command. Once accepted, a command triggers the execution of an **action**; this eventually terminates when the object reaches a new steady state (fig. 3). An action is identified by a *verb* applicable to the object name (e.g. the object "run" may perform any of the actions "start", "pause", "continue", and "stop"). The corresponding command is identified by the same verb, which may be thought of as being used in the imperative.

Figure 3: Structure of an object

An action consists of a sequence of operations, specified in the SM language by a list of instructions. These are essentially of two types: **DO** and **IF**. **DO** is an instruction that sends a command to another object; **IF** is an instruction that evaluates a Boolean function of the states of other objects, and makes a conditional branch depending on the result. While an object is executing an action, its state is undefined and no further command is accepted until the action is terminated. The final state reached after the execution of an action may depend on the results of tests performed on the state of other objects.

A command is not the only way of triggering an action: a state change in another object may provoke it; this type of dependence is specified in the SM language by a **WHEN** clause.

STATE MANAGER GENERATION

The SM runs as a process on a VAX/VMS machine. The SM program is the implementation of the model defined by the experiment description. This description, made by the

physicist in terms of the SM language, is processed by a compiler called the Translator, which produces the SM program code. This code is compiled and linked to obtain the SM program, which is then launched as a process in the target machine. Each time the experiment description changes, a new version of the SM program must be generated, because the code reflects the definitions given in the description.

first the objects have to be identified, then the action they will perform, finally the interactions between them.

The Translator

The Translator performs the usual functions of a compiler, namely syntax analysis, followed by semantic analysis and code generation.

The Translator itself was generated using a commercial compiler-generator tool called SCAN [4]. The SM language is formally described using the SCAN language; SCAN then generates the executable image of the Translator. This technique allows us to easily maintain and modify the SM language and its Translator.

State Manager program

The SM program code (produced by the Translator) is written in the Ada language [5]. Our choice was based on the fact that Ada is particularly well suited to object-based systems [6]. It was clear to us that the direct correspondence between objects and Ada tasks and between messages and Ada rendez-vous, would make the implementation straightforward. The complete integration of DEC Ada into the VMS environment, and the availability in the public domain of reusable software components, have been of great help in speeding up the implementation.

COMMUNICATION WITH THE OUTSIDE WORLD

The SM interacts with other processes through the exchange of messages (fig. 5).

```

object : RUN /visible
state : DORMANT
action : START
do BEGIN READ_OUT
if (READ_OUT in_state READING) then
do ACTIVATE DATA_RECORDING
terminate_action /state=ACTIVE
else
terminate_action /state=FAILED
end if
state : FAILED
action : RESET
do RESET READ_OUT
terminate_action /state=DORMANT
state : ACTIVE
when DATA_RECORDING in_state END_OF_TAPE
do STOP
action : STOP
do END READ_OUT
do DEACTIVATE DATA_RECORDING
terminate_action /state=DORMANT
object : DATA_RECORDING /associated
state : AVAILABLE
action : ACTIVATE
state : WRITING
action : DEACTIVATE
state : END_OF_TAPE
action : DEACTIVATE
object : READ_OUT /associated
state : DORMANT
action : BEGIN
state : READING
action : END
state : FAILED
action : RESET

```

Figure 4: An example of a simple State Manager program

The language

The SM language contains all the lexical terms required to describe the experiment in terms of objects, states, and actions. Figure 4 shows an example of its use, while a complete description can be found in the SM manual [3].

The model of the experiment can be designed following those guidelines that are valid for any object-based system:

Figure 5: External communication of the State Manager

A control process may send a message to any object; the SM dispatches the command and initiates the action.

The associated processes are seen by the SM through a special type of object in the experiment description, tagged with the attribute **associated**. These are different in that they cannot execute any action: a command received by an associated object is sent to the corresponding associated process. Conversely, a state change occurring in the entity controlled by an associated process is faithfully reflected by the associated object. The associated objects are, within the SM, the agents that make it possible to control the outside world.

SYSTEM CONFIGURATIONS

The communication between the SM and other processes makes use of DECNET [7]; therefore a distributed multi-machine configuration is naturally supported. In such an environment there is one SM process running in one of the machines, while other processes may run on any of the other machines in the configuration. The underlying communication package uses naming conventions to address remote objects correctly.

Independently of the physical configuration adopted (single-machine or multi-machine) it is possible to run more than one SM in the system. This is achieved by partitioning the system into domains, as is discussed in the next section.

State Manager domains

A domain defines the scope of visibility for the associated objects' names. As the addressing of messages between SM and other processes is based on the object names, the domain establishes the boundaries of the SM control space. This is achieved by assigning a name to each domain; for every process a logical name [8] is defined with a value equal to the name of its domain. By convention this logical name is the same for all the processes; the translation of the logical name gives different results according to the domain. It is the task of the communications package to translate the logical name and address the right object, indicated by the domain-object pair. The domain structure is superimposed in a completely transparent way; the experiment description contains no mention of the domain, nor need the associated processes have any special code to specify it.

The concept of domain can be applied, for instance, to the case of two SMs running in the same machine; one may be under test, while the other is used to control a production process. Partitioning the system avoids interferences when the SMs address distinct associated objects that have the same name.

Hierarchy of domains

Links can be established between distinct domains, to allow a certain degree of visibility between SMs (fig. 6).

This type of relationship can only be hierarchical; the slave SM does not know by whom it is controlled, while the master SM must know which slave SM it is driving, since it can drive more than one at a time. The links can be established only at specific points in the slave SMs. These points are objects tagged with the attribute **visible**: they allow the slave SM to be seen by another SM as an associated process. Therefore it is possible for a master SM to declare an **associated object** (in its own domain) which will mimic the behaviour of the **visible object** in a different domain. All the commands directed to the associated object will be dispatched to the visible object, and state changes in the visible object will be reproduced by the associated object. The domain must be specified when addressing a visible object, because there may be visible objects with the same name in distinct domains.

Figure 6: Hierarchy of State Managers domains

An example of application of these features is an experiment composed of subsystems (e.g. detectors) that can be run independently: they must be controlled by distinct SMs. At the end of the setting-up phase, the two subsystems will be integrated and the experiment will be run as a whole. There is no need to modify the original SMs: a third one may supervise them.

RUN-TIME LIBRARY

A run-time library SMI (SM Interface) is also provided. The processes that interact with the SM make use of its routines. A complete description of the routines can be found in the SM manual [3].

We list here for reference all the available entry points with a brief explanation.

SMI_INIT initializes the SMI package

SMI_ASSOCIATE associates the calling process with an object

SMI_GET_COMMAND delivers a command to the calling process

SMI_TERMINATE_COMMAND signals the end of the execution of a command and sets the associated object in a new state

SMI_SET_STATE signals an unsolicited state change

SMI_SEND_COMMAND sends a command to an object

An associated process is typically a program that controls a device, but it is also listening to messages coming from the SM and must be prepared to take action in real time. Messages are in the form of character strings; the associated process must parse the strings and interpret the commands; the command format and syntax is a matter of convention between the associated process and the associated object: the programmer defines both. The same holds for the states that the associated process may return to the SM: they must be declared as authorized states of the associated objects, in order to allow other objects to test them.

CONCLUSIONS

We have designed a language to describe the control aspects of High Energy Physics experiments. We have adopted a model that makes use of object-based techniques. The language Translator has been developed using a compiler-generator tool, and Ada has been used for the implementation of the State Manager. This approach has facilitated the development and maintenance of the product. The system is powerful and

flexible enough to meet the needs of a wide variety of experiments. The initial implementation is now coming into use by several experiments at CERN.

REFERENCES

- [1] A. Vascotto, "Modular architectures for on-line systems", in *Proc. CERN School of Computing*, Renesse, 1986, Geneva: CERN 87-04, 1987, p. 147.
- [2] M. Shaw, "Abstraction techniques in modern programming languages", *IEEE Trans. Soft.*, vol. 1, p. 10, Oct. 1984.
- [3] The MODEL State Manager - User Manual, CERN DD-OC-OS, April 1989.
- [4] SCAN is a trademark of Digital Equipment Corporation.
- [5] Ada Joint Program Office, US Department of Defence, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, Washington D.C.: Government Printing Office, 1983.
- [6] G. Booch, *Software Components with Ada*, Menlo Park, Calif.: Benjamin/Cummings, 1986.
- [7] DECNET is a trademark of Digital Equipment Corporation.
- [8] A logical name is a VAX/VMS feature. See the VAX/VMS System Service Reference Manual.