SMI++*

User Manual

B.Franek, C. Gaspar

* Revision/Version 3.1 26 March 1998

Preface

SMI++ is based on the original State Manager concept which was developed by the DELPHI experiment in collaboration with the DD/OC group of CERN. Although all the tools comprising the SMI++ system were re-designed, in order to allow more flexibility and to cover a larger set of applications and hardware platforms, the original concept stayed the same as well as parts of this manual. The credits for the original ideas and implementation go to M.Jonker, B.Franek, A.Vascotto and P. Vande Vyvre.

1 INTRODUCTION

The problem of controlling the various activities that constitute the data-acquisition and/or the slow control system of an experiment is a difficult one, mainly because of the complexity of the required operations, that involve a mixture of hardware and software. Not only the requirements may widely vary from experiment to experiment, but also within a single experiment it may be necessary to foresee important modifications of the control procedures as the setting-up proceeds and more experience is gained.

In the SMI framework, the adopted approach to the control problem is based on the State Manager (SM): the control aspects of an experiment are simulated by a finite state machine, which represents the behavioural model of the equipment under control. The various procedures necessary to run the experiment are executed by the SM which operates the corresponding state transitions on the state machine.

Beside the so called run control procedures, there are many other procedures in an experiment, requiring organisation and synchronisation of activities performed by independent processes and devices, such as calibrations, control of the apparatus, initial setting-up, etc. The SM approach can also be used for all these cases.

An experiment with limited requirements may consider to use some simpler methods for controlling the runs. A monolithic control program may very well do the job, thus avoiding the learning period necessary to properly use the SM. Still the advantages of using the SM may very well pay-off in the long term even for a small experiment. Aspects of the SM to be taken into account are its flexibility to adapt to different requirements, the inherent modularity of the model adopted to describe the experiment, the fact that the experiment description is easily modifiable, the self-documenting feature.

2 STATE MANAGER OVERVIEW

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 or proxies (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 proxies.

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.

The SM incorporates a model of the different activities to be organised, 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 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 the SMI tools in order to produce a State Manager and its associated proxies.

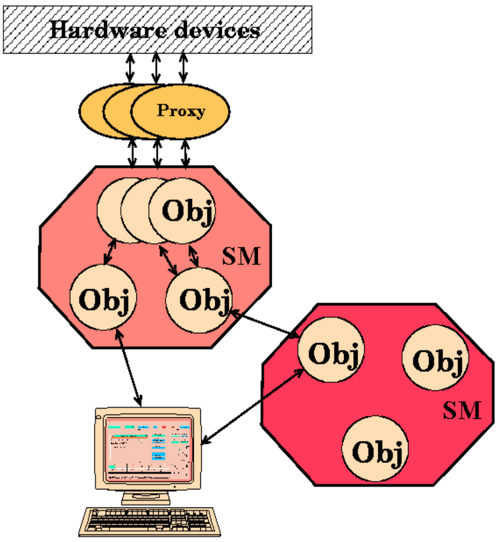

Figure 1: The SMI environment

2.1 Object/message model

The approach adopted for modelling the experiment is based on an object-oriented decomposition. 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”).

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. 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.

2.2 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. 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.

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.

2.3 Communication with the outside world

The SM interacts with other processes through the exchange of messages. A control process may send a message to any object; the SM dispatches the command and initiates the action.

The proxies 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 proxy. Conversely, a state change occurring in the entity controlled by a proxy 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. The communication between the SM and other processes makes use of an underlying package – DIM ; therefore a distributed heterogeneous 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.

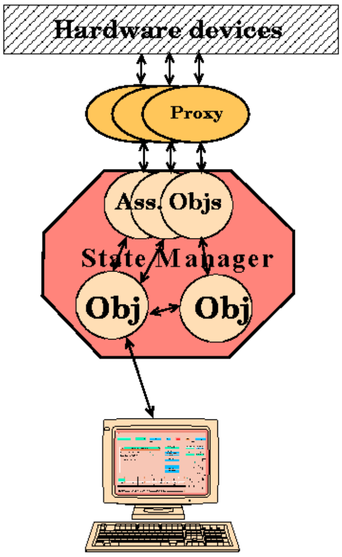

Figure 2: External communications of the State Manager

2.4 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. It is the task of the communications package to use the domain 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 proxies that have the same name.

2.5 Hierarchy of domains

Links can be established between distinct domains, to allow a certain degree of visibility between SMs (fig. 3).

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; this is achieved by prefixing the object name with the domain name, the two being separated by a double colon (e.g. 2::A means object A in domain 2).

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.

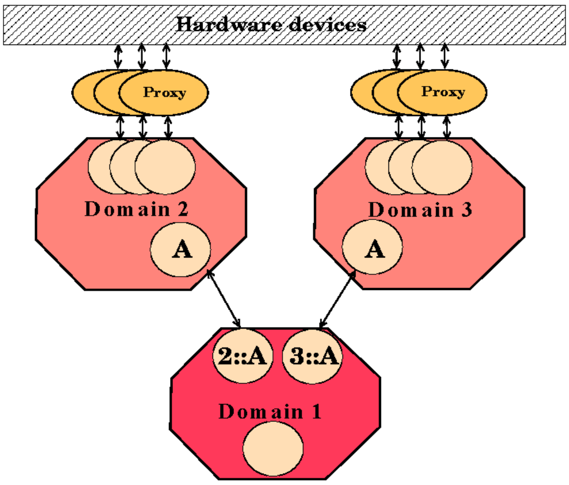

Figure 3: Hierarchy of State Managers domains

3 DESIGNING THE CONTROL SYSTEM

The application of the SM concepts to an experiment should follow a number of steps, where the control problem is progressively narrowed down to a manageable level of complexity.

First of all, one should indicate what are the broad areas where an independent control activity is necessary (independent means that the areas may be operated independently, but interactions between different areas are not precluded); these areas will eventually become SM domains. An examples of these areas in an experiment are “run control” or “detector control”; if the apparatus is composed of several detectors, each of them may belong to an independent SM domain; a particularly complex activity may be partitioned into distinct domains; the coherent running of the whole experiment may require a re-composition of tasks, by an hierarchical structure encompassing several domains. There is a wide freedom in the various choices, but it is important that each area be closely related to entities or groups of activities that are meaningful in the experiment context. Then, for each SM domain, one should determine what are the controllable items: these are the real entities that can be driven by commands and are susceptible of assuming a state; for each of them a service program will have to be provided, which will drive all the activity related to this entity, according to the instructions received by the SM (the program will be associated to an object in the SM). The experimenters may find it useful to define other objects related to logical entities (called logical objects throughout this manual, as opposed to associated objects) in order to represent the various modes of operating the experiment (such as “data-taking”, “calibration” etc.) or the collections of simpler objects (such as “central detector”, which for instance may be “ready” when all the components of the real central detector are ready).

A model of the behaviour of the given domain must be formulated as a SM program (a normal text file, that can be written using any text editor). There is a simple language to be used, which contains all the primitives suitable to this purpose (see figure 4 for an example).

The SM program describes the objects previously defined, the states they can take and the commands they accept in each state. The actions generated by each command must then be described: they consist of a sequence of instructions, each instruction being an executable statement of the experiment description language. The main types of instructions are DO and IF. DO is an instruction that sends a command to another object and therefore allows controlling the process of hierarchically dependent objects; IF is an instruction that allows conditional branching on the state of other objects. Of course Boolean expressions referring to the state of many objects are allowed by the language. The language contains also a WHEN statement, that allows the SM to trigger an action when there is an unsolicited state change in an object. The service programs must be prepared as well: they are the proxies in the terminology used throughout this manual. These programs operate on the apparatus upon receiving command messages. Their behaviour must be represented by a state machine. The generation of the code implementing the state machine is assisted by the SMI tool set; to this purpose, the behaviour of the associated object must be fully described, beyond what is normally needed by the SM. The functional part of the service program is left to the programmer, the communications with the SM program beeing provided by a set of library routines

```
object : RUN
        state : IDLE
                action :  START
                          do BEGIN EVT_BUILDER
                          if (EVT_BUILDER in_state READING) then
                                  do ACTIVATE DATA_LOGGER
                                  terminate_action/state=ACTIVE
                          else
                                  terminate_action/state=ERROR
        state : ERROR
                action :  RESET
                          do RESET EVT_BUILDER
                          terminate_action/state = IDLE
        state : ACTIVE
                when DATA_LOGGER in_state END_OF_TAPE
                          do STOP
                action :  STOP
                          do END EVT_BUILDER
                          do DEACTIVATE DATA_LOGGER
                          terminate_action/state=IDLE

object : DATE_LOGGER/associated
        state : AVAILABLE
                action : ACTIVATE
        state : WRITING
                action: DEACTIVATE
        state : END_OF_TAPE
                action : DEACTIVATE

object : EVT_BUILDER/associated
        state : IDLE
                action : BEGIN
        state : READING
                action : END
        state : ERROR
                action : RESET
```

Figure 4: An example of a simple State Manager program

The implementation phase consists of the generation of the SM (one or many) and of all the associated processes. The SM description is translated to an intermediate format that is then read at run time by a generic State Manager engine, that instantiates all SMI objects and drives the SM model. The proxy skeletons are also generated from the SM description.

The run-time system consists of one SM process per SM domain and all its proxies. The order of activation is not relevant, since the mechanisms by which the SM and its partner processes know about each other are irrespective of who has started first.

4 LANGUAGE DESCRIPTION

Figure gives a summary of all the language keywords. A formal definition of the language syntax can be found in appendix A.

Declarative statements		Executable statements	
Item	**Qualifiers**	**Command**	**Arguments**
ACTION		DO	
CLASS	/ASSOCIATED	IF…THEN…ELSE…ENDIF	AND…OR
OBJECT	/ASSOCIATED	WHEN… DO	NOT
	IS_OF_CLASS		IN_STATE
STATE	/DEAD_STATE		NOT_IN_STATE
	/INITIAL_STATE		
SUBOBJECT		TERMINATE_ACTION	/STATE

Figure 5: Keywords in the SM language

4.1 CLASS declaration

Format: CLASS: class_name [/ASSOCIATED]

This statement is used to describe a set of identical objects. The description of the set begins with the CLASS declaration, which specifies also some of the characteristics of the objects. All the statements concerning the set of objects must immediately follow. For the ASSOCIATED class, the same considerations apply as for the associated object (see the OBJECT declaration).

4.2 OBJECT declaration

Format: OBJECT object_name [/ASSOCIATED] [IS_OF_CLASS class_name]

The description of an object begins with the OBJECT declaration, which specifies also some of the characteristics of the object. All the statements concerning one object must follow immediately its declaration.

An ASSOCIATED object does not execute itself the action it receives. It transmits the command to its associated process and updates its state according to the variation of the associated process. Therefore the SM code of an associated object must only contain the list of states and for each of them the list of actions accepted in each state. The associated process may belong to a domain different from the one of the SM itself: in this case the domain must be explicitly specified as a prefix of the object name, separated from it by a double colon (e.g. TPC::HV indicates the object HV in the domain TPC).

The IS_OF_CLASS qualifier specifies that the object inherits the description and all the characteristics of the class.

4.3 SUBOBJECT declaration

Format: SUBOBJECT: subobject_name

This statement is used to define the different parts of an associated object, when the object is constituted of more than one finite-state machine. Sub-objects must be defined within the description of the parent object. Each sub-object has got its own state attribute and its own set of actions. Sub-objects are referenced by prefixing their name with the name of the parent object, separated by the symbol dot (.).

4.4 ACTION declaration

Format: ACTION : action_name [(param [= default_value], …)]

This statement declares a command accepted by the object in a given state (the one preceding the ACTION declaration). If the object accepts the same command in different states, this command must be declared for all the states.

For a logical object the ACTION declaration must be followed by the instructions (executable statements) that the object performs at reception of the command. On the other hand, an associated object does not execute any action but sends the command as a message string to the associated process; therefore there are no executable statements following the ACTION declaration.

If the action may receive a parameter, the parameter must be declared at this stage. This parameter can be used later on DO instructions.

4.5 STATE declaration

Format: STATE : state_name [/INITIAL_STATE] [/DEAD-STATE]

This statement is used to declare one of the possible states an object can take. The STATE declaration may be followed by all the WHEN and ACTION statements that apply to this state.

INITIAL_STATE is a state qualifier whose semantics depends on whether the object is associated or not. In the case of logical objects it indicates that the object must be put in the flagged state when the SM program starts up. In the case of associated objects, the flagged state is the one that the object initially assumes when a process becomes associated with it. At the SM start-up, the object may also assume this state, if the associated process is running and it has not yet specified a state itself. By default, the initial state is the first one declared.

DEAD_STATE is a state qualifier which only applies to associated objects. It designates the state to be assigned to an object when the associated process aborts or in general is not running. The behaviour of an object as well depends on whether there is a state flagged with this qualifier.

If no state is flagged, when the associated process aborts the object goes into a hung state that forbids any further operations. All the command arriving to this object (including IF statements checking its state) will be put in a queue (the originating actions will be suspended) until the associated process resumes the operations. There is no means by which other objects can test this condition.

On the other hand, if one of the states of an object has been flagged with the DEAD_STATE qualifier, then the object remains active even while the associated process is not running. The received commands are accepted and discarded (since there is no active partner); the IF statements work as usual, namely any object can check for this state.

4.6 DO statement

Format: DO action_name [(parameter=value,…)] object_name

This statement allows an object to trigger further actions on itself or on other objects during the execution of an action.

If the destination object is associated, parameters can be specified, which will be transmitted within the command string. The parameters must be enclosed between brackets; Values can be ASCII strings enclosed in ‘ ” ’ or parameters received by the current ACTION (see ACTION declaration).

Example:

```
        DO start (run_number=”37”) run_control
```

or

```
    ACTION : START_RUN (new_run_number)
        DO start (run_number = new_run_number) run_control
```

If we assume, in the previous example, that new_run_number translates to 37, the object run_control will receive in both cases a command string with the following format:

```
    start /run_number=37
```

4.7 IF statement

Format: IF condition THEN statements [ELSE statements] ENDIF

The IF statement conditionally executes a statement or a block of statements. The condition can be a logical expression combining the state of various objects with logical operators:

Example:

```
        IF (tape1 IN_STATE end_of_tape OR alarm IN_STATE on)
            (AND run NOT_IN_STATE dormant)
        THEN
            DO stop run
        ENDIF
```

The IF statement realises also the synchronisation between all the objects whose state is tested in the condition. Before evaluating the condition, all the objects will execute the pending actions (which may be present in a queue). The state reached after the execution of these actions is then used for the evaluation of the condition and will remain the same up to the execution of the ENDIF. The statements following the THEN up to the ELSE are executed if the condition is true. The statements following the ELSE up to the ENDIF are executed if the condition is false.

4.8 TERMINATE_ACTION statement

Format: TERMINATE_ACTION /STATE = state_name

This statement instructs an object to stop the execution of the current action and to assume a new stable state.

4.9 WHEN statement

Format: WHEN condition DO action_name

The WHEN statement allows an object to react to unsolicited state changes of some other objects. The WHEN statement applies to one state only: if the logical condition becomes true when the object is in this state, the object starts to execute the action specified in the WHEN statement. This action (and all the actions on any other object originated by it) will be processed at a higher priority.

The condition of the WHEN statement can be a logical expression combining the state of various objects with logical operators, e.g.:

WHEN ((beam NOT_IN_STATE on) OR (slow_control IN_STATE alarm)) DO stop

5 SMI TOOLS

A set of tools are available in order develop, implement and test the control system as described in figure 6.

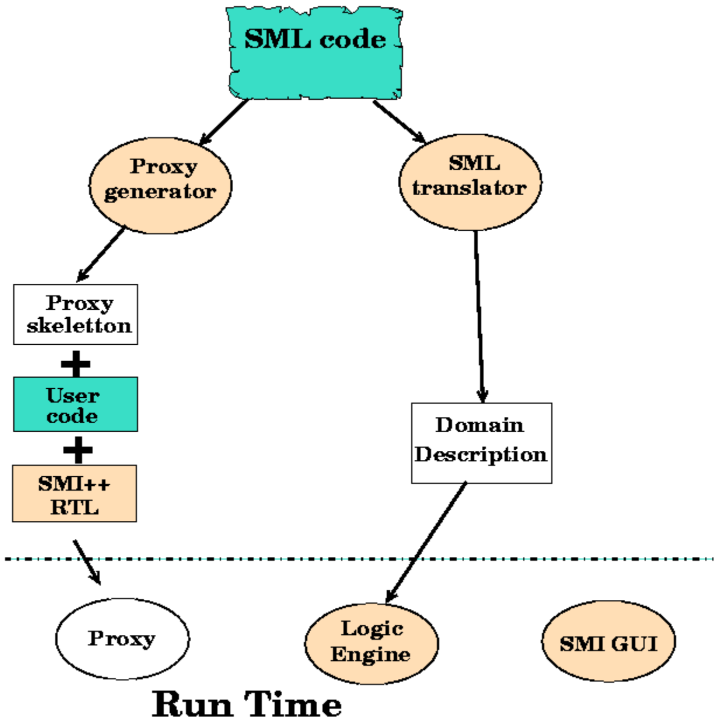

Figure 6: the SMI Tool Set

5.1 Generation and execution of a State Manager

The description of a specific domain in SML is normally stored in a file with extension ".SMI". Each .SMI file corresponds to a domain. The SMI Translator has to be run on this file in order to generate a .SOBJ file which will then be used at run time by a generic SM engine (the logic engine in the figure). The tools involved are:

- **smiTrans <file_name>**

takes <file_name>.SMI and produces <file_name>.SOBJ

- **smiSM <domain_name> <file_name>**

starts a SM for the domain <domain_name> behaving as specified in <file_name>,SOBJ

5.2 Generation and execution of proxies

Proxy skeletons can also be generated from the .SMI files by running the Proxy Generator. The output is a file per proxy called <object_name>_skel.c. I.e. they are "C" files where the possible states and actions are defined and the user can insert it's own code and decide on the next state of the proxy whenever a command is received. The advised procedure is: to copy the file <object_name>_skel.c to <object_name>.c and modify this one, if possible store the user routines in separate files. The resulting C code should be linked with the SMIRTL library (the complete description of the SMIRTL routines will be provided later in this document). At run time this process requires as parameter the <domain_name> as given to the State Manager.

The tools involved in the creation of proxies are:

- **smiGen <file_name>**

Generates as many output files as associated objects each one called <object_name>_skel.c

- **SMIRTL**

The SMI run-time library contains all the routines needed for communication with the SM.

5.3 The SMI generic user interface

Included in the Tool Set is a Generic Configurable User Interface for visualisation of a specific domain. The UI allows the visualisation of the states of objects in a certain domain, as well as sending commands to them. The set of objects to be displayed, it's position and the colour of it's states are configurable. The GUI can be started by:

- **smiGUI <domain_name>**

An example is shown in Fig 7.

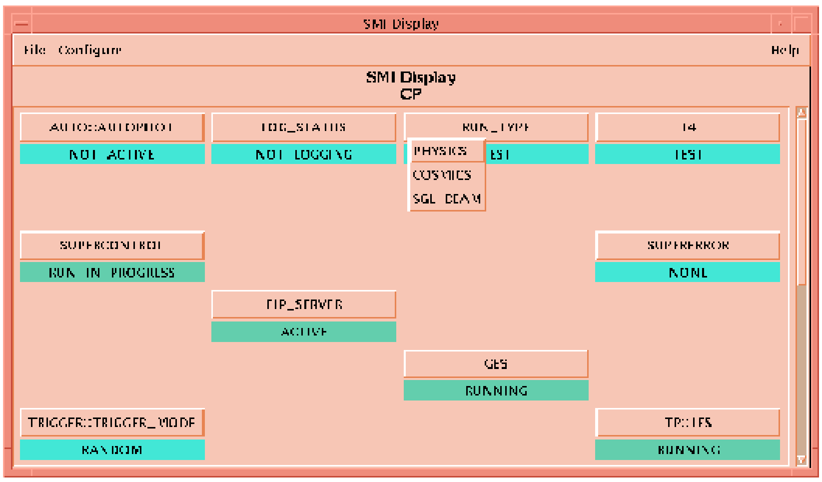

Figure 7: example of an SMI display session

5.4 Building your own User Interface

The Generic User Interface provided in the tool set is very useful for use during the implementation and test phase of a control system, but is not recommended as production user interface for the control system. Normally in the final UI the user would like to combine objects from different domains and also information on quantities like RUN_NUMBER and TRIGGER_RATE that are not available as SMI object states. The SMI Tool Set contains a library that allows easy access to information on objects, states, and available or currently executing actions. The functionality available in this library will be described later in this document). In order to access these functions your user interface should link with:

- **SMIUIRTL**

The SMI User Interface Run-Time library

This library also allows other processes (not UIs) to send commands to SMI objects and/or to inquire on the state of a particular object somewhere in the system.

6 SMIRTL Description

SMIRTL implements the routines used by a proxy in order to connect, receive commands and transmit state changes to its associated object in a SM. These routines are:

- **smi_attach(obj_name, command_handler)**
  Parameters:

  char *obj_name The name of the associated object to connect to. In the form <domain_name>::<object_name>

  void (*command_handler)() The routine to be called when a command arrives from the SM, handle_command is called with no parameters.

  Description:

  smi_attach should be called by a proxy at start-up in order to connect to its associated object in the SM.

- **smi_set_par(par, value, type)**
  Parameters:

  char *par The parameter name.

  void *value The value of the parameter (passed by reference).

  int type The parameter type: (STRING, INTEGER or FLOAT).

  Description:

  Smi_set_par should be used in order to set object parameters for associated objects. If an associated objects has parameters, smi_set_par should be called once for each parameter before the first smi_set_state. Further parameter setting can be done by the proxy at any time but they will only take effect at the next smi_set_state.

- **smi_set_state(state)**
  Parameters:

  char *state The state of the proxy.

  Description:

  Set the initial or new state of the proxy by sending it to its associated object in the SM. An smi_set_state should be done at start-up in order to communicate the proxy's initial state.

- **smi_terminate_action(state)**
  Parameters:

  char *state The state of the proxy.

  Description:

  Set the new state of the proxy after executing an action, by sending it to its associated object in the SM.

- **smi_get_cmnd(cmnd)**
  Parameters:

  char *cmnd The command just received

  Description:

  When inside the comman_handler the user can use this routine to get the command, the command is composed of the action name and any parameters in the form action/par0=val0/par1=val1… .

- **smi_get_action(action, n_params)**
  Parameters:

  char *action The last action received

  int n_params Number of parameters with this action

  Description:

  When inside the comman_handler the user can use this routine to get the action name and nr. of

parameters.

- **int smi_test_action(action)**
  Parameters:
  char *action Action name to compare with the received action.
  Description:
  Test if the action received is "action". The function returns 1 if it is, 0 if not.

- **int smi_get_next_par(param, type, size)**
  Parameters:
  char *param The name of the parameter
  int *type The type of the parameter (STRING, INTEGER, FLOAT)
  int *size The size of the parameter
  Description:
  When inside the command handler this routine can be used to retrieve the parameters present on a received command. The function will return 0 when all parameters have been read, otherwise it returns the parameter name, the parameter type and the size necessary to store the parameter.

- **int smi_get_par_value(param, value)**
  Parameters:
  char *param The name of the parameter
  void *value The parameter value
  Description:
  When inside the command handler this routine can be used to retrieve the parameter value for the parameter returned by smi_get_next_par. The value argument will be filled with either an integer, a double float or a string (the size was specified in the previous routine) depending on the parameter type. This function can also be used directly if the user knows the parameter name, type and size. The function will return 0 if the specified parameter does not exist.

- **smi_volatile()**
  Parameters:
  none
  Description:
  Smi_volatile() makes a proxy live only while its associated SMI Domain is active, i.e., a proxy can call smi_volatile() (normally at start-up) in order to die whenever the SMI domain the proxy is attached to dies.

7 SMIUIRTL Description

SMIUIRTL implements the routines used by a user interface or any other process wanting to get information on the states of objects running in a certain SM and/or to send commands to them. These routines are:

- **smiui_current_state(obj_name, state, busy_action)**
  Parameters:
  char *obj_name The name of the object in the form <domain_name>::<object_name>
  char *state The buffer where the state will be stored on return.
  char *action If the object is busy the name of the action it is currently executing, otherwise a null string.
  Description:
  Used to know the current state of an object in a given domain. It also returns information on whether the object is busy executing an action or idle.

- **smiui_send_command(obj_name, cmnd)**
  Parameters:
  char *obj_name The name of the object in the form <domain_name>::<object_name>
  char *cmnd The cmnd in the form action/par="val"…
  Description:
  Used to send a command to any object in any domain.

- **smiui_connect_domain(domain)**
  Parameters:
  char *domain The name of the domain to connect to.
  Description:
  Used to connect to a specific domain, all successive calls can then specify this domain. It returns the number of objects in the domain.
  Note: If the object has sub-objects each subobject is counted as an object. Also in the next routine if the returned object is a sub-object it will be returned on the form <object>.<subobject>

- **int smiui_get_next_object(domain, obj_name)**
  Parameters:
  char *domain The name of the domain.
  char *obj_name The name of the object in the form <domain_name>::<object_name>.
  Description:
  Get the name of the next object in a domain (several domains can be accessed at the "same time"). This routine returns 0 when all the objects of a particular domain have been retrieved

- **int smiui_book_statechange(obj_name, statechange_handler, param)**
  Parameters:
  char *obj_name The name of the object.
  void (*statechange_handler)() The routine to handle state changes
  int param A parameter to pass to the above routine
  Description:
  Set up a callback routine to be called whenever the specified object changes state. smiui_book_statechange returns an identifier that should be used on the following calls. The statechange_handler routine will be called with two parameters: the identifier mentioned above and param.

- **smiui_cancel_statechange(id)**
  Parameters:

int id The identifier returned by the previous call.

Description:

Cancel attention on the object, further state changes will not be signalled.

- **smiui_get_state(id, busi_flag, state, n_actions)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

*Int *busy_flag* Returns a flag indicating if the current object is busy

*char *state* Returns the current state of the object

*int *n_actions* Returns the number of possible actions in the object's current state

Description:

This routine is normally called from the statechange_handler routine in order to get the object state and some additional parameters.

- **smiui_get_action_in_progress(id, action)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

*char *action* The action currently beeing executed.

Description:

Called in order to get the action currently beeing executed by an object whenever the busy_flag in the previous routine is 1.

- **int smiui_get_next_action(id, action, n_pars)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

*char *action* The action name.

int n_pars The number of parameters this action takes.

Description:

This routine retrieves the next action available to an object in the current state.

- **smiui_get_first_action(id, action, n_pars)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

*char *action* The action name.

int n_pars The number of parameters this action takes

Description:

Positions the next_action pointer back to the first action and retrieves it. smiui_get_next_action can be called again from then on.

- **int smiui_get_next_param(id, param, type, default_value_size)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

*char *param* The action's parameter name.

*int *type* The parameter type (STRING, INTEGER, FLOAT)

*int *default_value_size* The size of the parameter's default value if a default has been specified, otherwise 0.

Description:

This routine retrieves the next parameter specified for the current action available to an object in the current state. It returns 0 when all parameters have been returned.

- **int smiui_get_param_default_value(id, default_value)**

Parameters:

int id The identifier returned by smiui_book_statechanged.

void *default_value The parameter's default value.

Description:

This routine retrieves the parameter default value, for the parameter received from smiui_get_next_param. This routine should only be called if default_value_size was not zero. It returns 0 if the parameter did not have a default value. The contents of default_value can be of type integer, float or character string according to the previous routine.

- **int smiui_book_alloc_statechange(domain, alloc_statechange_handler, param)**

Parameters:

char *domain The SMI domain name.
void (*alloc_statechange_handler)() The routine to handle state changes
int param A parameter to pass to the above routine

Description:

Set up a callback routine to be called whenever the allocation state of the specified domain changes state. The alloc_statechange_handler routine will be called with the single parameter - param. smiui_book_alloc_statechange returns an identifier that should be used on the following calls.

- **smiui_cancel_alloc_statechange(id)**

Parameters:

int id The identifier returned by the previous call.

Description:

Cancel attention on the allocation state of the domain, further state changes will not be signalled.

- **smiui_get_alloc_state(id, busy_flag, state)**

Parameters:

int id The identifier returned by smiui_book_alloc_statechange.
Int *busy_flag Returns a flag indicating if the domain allocation object is currently busy
char *state Returns the current state of the allocation object

Description:

This routine is normally called from the alloc_statechange_handler routine in order to get the allocation state of the domain.

- **int smiui_allocate_tome(domain)**

Parameters:

char *domain The domain name.

Description:

This routine can be used by a user interface or another process in order to allocate an SMI domain to itself. An smiui_connect_domain should have been done before hand. Smiui_allocate_domain returns 1 if the domain was successfully allocated and 0 if it was not connected or already allocated to someone else.

- **int smiui_allocate(domainA, domainB)**

Parameters:

char *domainA The domain to be allocated.
char *domainB The domain to allocate to.

Description:

This routine can be used by a user interface or another process in order to allocate an SMI domain to another SMI domain. An smiui_connect_domain should have been done before hand to the allocator domain (domainB). Smiui_allocate_domain returns 1 if domainA was successfully allocated to domainB and 0 if it domainB was not connected or domainA already allocated to someone else.

- **int smiui_release(domain)**

Parameters:

char *domain The domain name.

Description:

This routine can be used by a user interface or another process in order to release a previously allocated SMI domain. An smiui_connect_domain should have been done before hand. Smiui_release_domain returns 1 if the domain was successfully released and 0 if the domain was not allocated to this process.

8 AVAILABILITY

SMI++ is available on mixed environments composed of any of the following platforms:

VMS - VAX

- ALPHA

UNIX - HP-UX

- IBM-AIX

- SUN-Solaris

- SUN-SunOS

LINUX

WINDOWS NT

In addition Proxies can also be run on the following Operating Systems:

OS9

LynxOS

VxWorks

For information on distribution and installation procedures please refer to Appendix B

APPENDIX A - SYNTAX OF THE STATE MANAGER LANGUAGE

The syntax is described in BNF form with the following formalism:

Terminals are the basic symbols from which a program is formed: languages keywords, operators... They are written in upper case. The following punctuation marks are terminals of the SMI language: colon “:”, double colon “::”, comma “,”, parenthesis “()”, equal “=” and quotes “””. Non-terminals are syntactic variables that define sets of strings. They give a hierarchical structure to the language, which is useful for the translation. The notation used is lower case words between “<” and “>”.

Each production rule consists of a non-terminal followed by “::=” followed by a string of terminals and non-terminals. The syntax of a program is correct if it can be obtained by the use of the production rules from the non-terminal start symbol : “<program>”. The braces “{}*”denote the repetition zero or more times of the enclosed items and “{}+” at least once. The square parenthesis “[]” are used for optional terminals or non-terminals in production rules.

The vertical bar “|” is used for a choice between different terminals or non-terminals.

Definition of a SMI program

<program> ::= { <class def> | <object_def> }+

Definition of a class

<class def> ::= <class decl> { <logical object descr> | <associated class spec> }
<associated class spec> ::= { /ASSOCIATED <associated object descr> }

Definition of an object

<object def> ::= <internal object def> | <external object def>
<internal object def> ::= <internal object decl> { <logical object spec> | <associated object spec> }
<external object def> ::= <external object decl> | <associated object spec>
<logical object spec> ::= { <reference to a class> | <logical object descr> }
<associated object spec> ::= <reference to a class> | { /ASSOCIATED <associated object descr> }
<reference to a class> ::= IS_OF_CLASS <class identifier>

Description of an object

<logical object descr> ::= { <logical state def> }+
<associated object descr> ::= {<associated subobject def> }+ | {<associated state def> }+

Definition of a sub-object

<associated subobject def ::= <subobject decl> { <associated state def>}+

Definition of a state

<logical state def> ::= <state decl> [/INITIAL_STATE] { <when instructions> }*
{ <logical action def>}+
<associated state def> ::= <state decl> [/DEAD_STATE] { <action_decl> }*

Definition of an action

<logical action def ::= <action decl> { <standard instruction> }*

Definition of an instruction

<when instruction> ::= WHEN <condition> DO <action identifier>
<standard instruction> ::= <do instruction> | IF <condition> THEN { <standard instruction> }*
[ELSE { <standard instruction> }*] ENDIF

<do instruction>	::=	DO <action identifier> [({<parameter>}+ {, <parameter> }*)] <object reference>

Logical expression

<condition>	::=	{<factor> { { AND <factor> } | { OR <factor> } }*
<factor>	::=	[NOT] <term>
<term>	::=	{ <object reference> [. <subobject identifier>] { IN_STATE | NOT_IN_STATE } <state list> } | { (<condition>) }

Declarations

<class decl>	::=	CLASS: <class identifier>
<internal object decl>	::=	OBJECT: <object identifier>
<external object decl>	::=	OBJECT: <domain identifier> :: <object identifier>
<subobject decl>	::=	SUBOBJECT: <subobject identifier>
<state decl>	::=	STATE: <state identifier>
<action decl>	::=	ACTION: <action identifier> [({<parameter decl>}+ {, <parameter decl> }*)]
<parameter decl>	::=	<formal parameter identifier> [= <string literal>]

Syntactic elements

<state list>	::=	<state identifier> | { <left curly bracket> <state identifier> { , <state_identifier> }* <right curly bracket> }
<parameter>	::=	<parameter identifier> = { <formal parameter identifier> | <stringliteral> }+
<object reference>	::=	[<domain identifier> ::] <object identifier>
<identifier>	::=	<alpha> {[_] | <alphanumeric> }*
<string literal	::=	“ {all characters except “ }* “
<integer>	::=	<digit> {<digit>}*
<alphanumeric>	::=	<digit> | <alpha>
<digit>	::=	0|1|2|3|4|5|6|7|8|9
<alpha>	::=	A| B| C| D| E| F| G| H| I| J| K| L| M| N| O| P| Q| R| S| T| U| V| W| X| Y| Z| a| b| c| d| e| f| g| h| i| j| k| l| m| n| o| p| q| r| s| t| u| v| w| x| y| z
<left curly bracket>	::=	{
<right curly bracket>	::=	}
<comment>	::=	! {all characters }*

APPENDIX B - DISTRIBUTION AND INSTALLATION