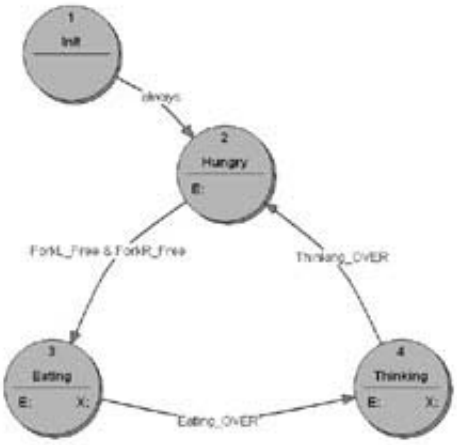



# VFSMML 1.0

## Virtual Finite State Machine Mark-up Language

Date: 04.04.2004
Release: 1.0
Author: Thomas Wagner

# Table of Contents

# 1 Introduction

## 1.1 VFSM and its Notation

A finite state machine (FSM), sometimes called a finite automaton, is a system whose condition depends not just on external stimuli, but on the history of those stimuli. VFSM is a method for specifying FSM as "Platform Independent Models" in such complete detail that they may be executed directly in a run-time system without requiring further transformation.

A very simple FSM example is a keyboard, which might be in the normal, initial state, or the caps-lock state, depending on the number of times the Caps Lock key has been pressed. Although programmers are often introduced to FSMs in the context of parsing input text for compilers, the concept is very much more general, and applies to most "reactive systems" in which internal processes are governed or influenced by external events. In such systems the FSM does not merely run through a sequence, producing an end result, but it normally operates throughout the period when the system is able to function.

The academic definition of an FSM is a "quintuple" $A = <\Sigma, S, S_0, \delta, F>$ where $\Sigma$ is an alphabet, S is a finite, non-empty set of states, $S_0$ is a set of initial states, $\delta:S \times \Sigma \rightarrow \rho(s)$ is a transition function, and F is the set of accepting states (perhaps empty). This is perhaps not too helpful to the practitioner, but quite an amount of theory can be found in the various text-books if he is mathematically inclined. A key point is, however, the input alphabet $\Sigma$ , which defines the stimuli to which the FSM will react.

A weak point of the above definition is the absence of actions. One might think that the task of the state machine is to change states until it reaches an end state, and there is a class of state machines called "deterministic" which need to do this (for instance, parsing text and reporting when specific sequences are detected). The true task of state machine is to trigger actions according to situation defined by the present state and stimuli.

FSMs are normally described in a diagrammatic form, using a circle to represent each state, and lines with arrow heads to represent transitions between the various states. The addition of details explaining what will provoke any transition is often difficult to achieve, and a text description of the FSM is then needed.

As the FSM functions, changing state from time to time, it will provoke actions in other parts of the system, as required for the specific project. The actions can be performed by entering a state (entry actions), leaving the state (exit actions) or they can be triggered by an input (input actions) irrespectively of the state transition. Input actions and entry actions are the basic actions used by state machine specification. A state machine which uses only entry actions is called a Moore model. A state machine which uses only input actions is called a Mealy model. In practice, models which combine all actions: entry, exit and input are preferable solutions of state machines.

An FSM will often seem to be very easy to design, and will require a modest number of states - say about a dozen - to perform its task. Then, when the designer considers what might go wrong in various ways with the external system, he is forced to add many states to handle these errors, and the whole FSM becomes very large and

difficult to deal with. There is a solution: split the FSM into several different FSMs which are linked together, and where each one deals with a part of the problem. In very large systems, one will find that many of the error-handling processes are almost identical, and this can save time in the development phase by permitting re-use of some FSM designs.

A definition of a standard notation for state machines is mainly determined by: the variety of input / output signals and a system of state machines.

This document describes a way in which the FSM concept can be applied in software, and by which FSM designs can be expressed in full detail in XML format. A detailed description about FSM and VFSM can be found at [1].

## 1.2 Virtual environment

Input actions and transitions are controlled by expressions using boolean conditions and they are due if the expressions are true, as for instance:

```
if ((Temperature > MAX_TEMPERATURE) && (Timer == Is_Running) ||
   Door_Closed)
```

In the above control expression using an **if** control statement there are 3 different input values: Temperature which is a floating-point number, Timer which is a state (Running) of a timer and Door which is a digital value (On, Off). In most cases, these signals cannot be used in their original form as variables in a boolean equation (except Door which is a boolean value); instead their control-relevant (boolean) values are calculated using comparison, equal, or other operators. In other words, the boolean condition must be calculated.

The concept which makes it possible to disregard such details when designing FSMs is that of the “virtual environment” as described below.

We introduce input names to “describe” the control values of the input values. Instead of calculating the control values of the input values we assume that the boolean conditions are specified using the names. Of course, “somewhere” the names have to be continuously updated (calculated) representing always the true situation of the inputs. The input names create a special environment where all input conditions are of the same type (just names). We call this environment virtual to underline the fact that we use there not real signals but only a representation of the control feature of the signals.

As will be made clear later, the several possible control values of one physical input can be likened to “states” of a FSM embodying the control-relevant behavior of that input. This concept is developed further, in terms of “pre-defined VFSM” which describe inputs and also outputs in VFSMML.

The control values of all input signals define a virtual input. The virtual input represents complete information about the inputs influencing the state machine behavior.

Similarly, we introduce output names to “describe” actions to be done as a result of a state transition or input condition. The true actions are also of several natures, as for instance: switch on the power, set a voltage to some value, send a message, start a timer, etc. Using output names to describe actions we define again an environment of

uniform control actions that represent only the essence of the action without their implementation specifics. Of course, the output names must be “somewhere” eventually transformed to true output actions. The output names are another part of the virtual environment.

## 1.3 Events and signal life time

State machines are triggered by events. The events are changes of input signals. Some of the signal changes are singular events which can be “forgotten”; we may say they are consumed by triggering the state machine. Other signals cannot be forgotten – they just exist until replaced by another value, for instance the temperature has always some value – it changes only from time to time. The partition between true events and signals that are always present is definite. Several values are neither true events nor always present signals; they just live for some time. For instance, the life time of a command or timeout is not well defined: sometimes these are consumed immediately, sometimes we use them until they are replaced by other values and sometimes they have to be “forgotten” by force.

## 1.4 Positive-logic algebra

The introduction of virtual environment consisting of input and output names gives us a chance to express logical conditions using only boolean equations. The only difficulty arises from the fact that as a rule a true control input has more than two control values. The only input that corresponds to Boolean values: true and false could be a digital input represented normally by two values: on and off. Other inputs have several control features:

- a Temperature may be for instance: ok, too_high, very_high, too_low, unknown,
- a Timer may be for instance: running, over, stopped,
- a Command may be for instance: start, stop, continue, break,
- a Parameter may be for instance: initialized, changed, undefined, defined.

Note that in the examples we say “may be” as there are no absolute definitions of the control values - they are application dependent. For instance, in one application temperature: ok and not_ok are sufficient; in another application we could need more detailed knowledge about the temperature.

Note also that this naming convention allows full description of signal features, for instance the digital input said above to be a boolean one is in fact a 3-valued signal: true, false and unknown.

In all cases where an input has more than two control values the usage of the NOT operator would be ambiguous, for instance, what would mean a negation of temperature = ok?

Thus, we use a limited Boolean algebra where names are treated as boolean values but only AND and OR operators are allowed, and of course parentheses..

The expression

```
if ((Temperature > MAX_TEMPERATURE) && (Timer == Is_Running) ||
   Door_Closed)
```

will be than expressed for instance as:

```
Temperature_very_high & Timer_running | Door_closed
```

## 1.5 State machine execution model

Definition of actions suggests already that for any application several state machines do exist describing exactly the same control behavior. Another factor influencing the state machine is the execution model. The execution of entry and exit actions is clear but the execution of input actions must be defined. The same problem exists with multiple state transitions triggered by a single event. The following state machine execution model is used:

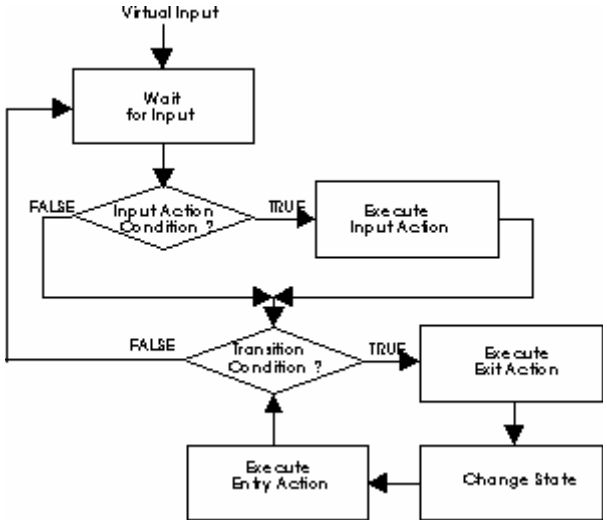



**Figure 1: State machine execution model**

A change of virtual input triggers the execution. The conditions of all input actions are tested and the input actions whose conditions are true are performed. Then, the transition conditions are tested. This process is prioritized and the first found true condition causes a transition to the new state in the following sequence: the exit action is performed, the transition is done and the entry action in the new state is carried out. Then, again, the transition conditions in the new state are tested. If a true condition is found, another 'exit action – transition – entry action' sequence is done. This process is continued until there are no more transitions due and the state machine waits for another change of the virtual input.

## 1.6 System of state machines

Any non-trivial application requires a complex behavioral model – a single state machine will be too large, i.e. to complex and difficult to handle. The solution is to partition the complex model into several smaller state machines which are easier to define and handle. There are two overlapping topics to be solved by a system of state machines: the communication among state machines and the overall structure of the system.

Intuitively, it seems to be obvious that a system of state machines where each state machine may exchange some information with any other state machine will be very difficult to control, specify and maintain. On the other hand, the variety of application tasks requires certain flexibility. The VFSM approach suggests - but does not impose - a hierarchical structure with master(s) in higher control levels and slaves in lower levels. The communication among state machines is defined according to this hierarchical structure: a master sends commands to slaves and uses the slaves' states as control signals. The following example of a VFSM system of state machines

demonstrates this concept, where for instance the Transport state machine is a slave of the master Main and a master of the slaves MotorX and MotorY.

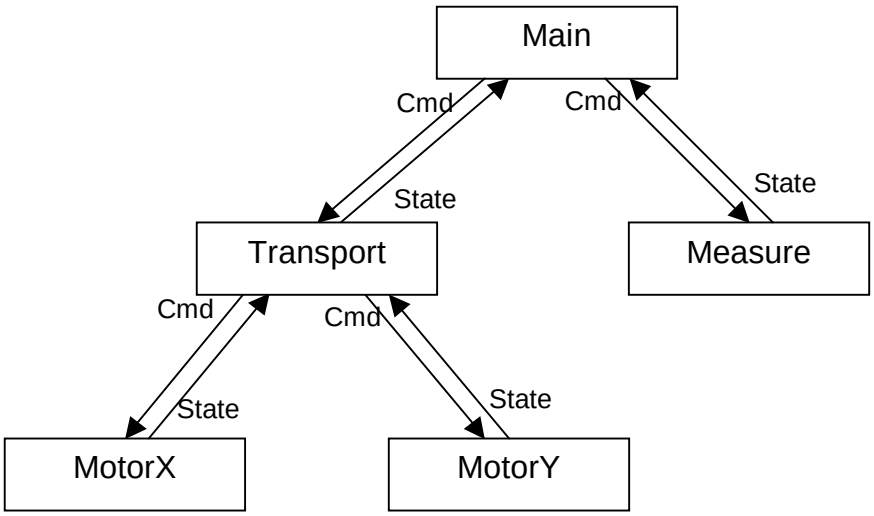



**Figure 2: Master -Slave concept**

## 1.7 Real time data base (RTDB)

The input / output signals are of different types. They carry information like: data, units, scaling factors, etc. In addition they have control features. For instance:

- A Temperature (actually a number representing sensor voltage) is characterized by a **value, scaling factor, unit** and it has a control value, e.g.: **HIGH, OK, LOW**.
- A Timer is characterized by a **timeout value, clock base, running time** and it has a control value, e.g. **OVER, RUNNING, RESET, AND STOPPED**. In addition, it may be started, stopped or reset.
- A Command is a **number** (integer). It may be an input signal (control value) of a state machine or it may be an output signal for another state machine.
- A Parameter is characterized by a **value, initial value, unit, low limit, high limit, category** and it has a control value, e.g. **UNDEFINED, DEFINED, INIT, CHANGED**.

These examples demonstrate that it is possible to define a number of known and often used objects which have some standard properties. These objects can be used as a base of a real time data base which takes care of input / output signal management storing them and filtering the control value from their value. This arrangement can be used to define a system of automatic creation and actualization of control values. Such a data base must of course provide a software interface to expand the object types and to program a link to the true input / output signals and to data processing software.

## 1.8 Design Goals of VFSMML

VFSMML specifies a XML[1] notation for state machines. The main obstacles are the variety of input / output signals. To standardize the input / output signals we use the VFSM concept which defines the virtual environment and the positive-logic algebra for condition expressions. In addition, we assume the usage of the VFSM execution model. Other elements of the VFSM concept: hierarchical structure and Master-Slave interface are suggested elements but they are not enforced by the standard.

The standard uses the concept of RTDB defining attributes (properties) of several objects. This part of the standard is open and can be expanded by new object definitions if desired.

The Figure 3 below shows how VFSMML fits into the XML concept. All details about the XML definition can be found in [2].

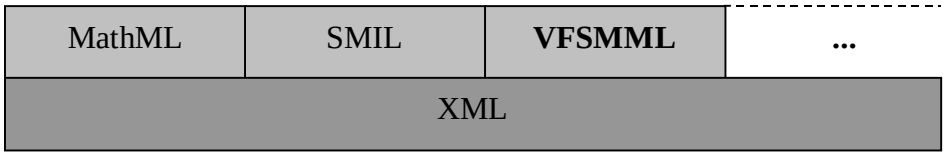



**Figure 3: VFSMML within the XML concept**

[1] For those readers not too familiar with XML we wish to point out that, although XML text is readable by humans, it is rather cumbersome. In practice, an XML document is commonly read with the aid of a style sheet, which drastically alters the appearance, and in many cases removes the XML tags. Such a style sheet, as an XLS file, is for instance available for viewing VFSMML "StateWORKS" files. The full XML format is used in the examples below, so as to explain the structure of VFSMML documents.

# 2 VFSMML Fundamentals

## 2.1 VFSMML Overview

This chapter introduces the basic ideas and describes the overall design of VFSMML. The second section presents a number of motivating examples, to give the reader something concrete to refer to while reading subsequent chapters of the VFSMML specification. The final section describes basic features of the VFSMML syntax and grammar, which apply to all VFSMML mark-up.

The VFSMML mark-up consists of about 23 elements and introduces a small set of attributes.

## 2.2 Virtual input and output

The virtual input are values (names) which are used in the state machine specification to describe behavior conditions, i.e. input actions or transitions. Because the virtual input names describe the condition of each real input, they can be considered as being the names of states of VFSM which represent those inputs, for the purposes of use at higher levels. The virtual outputs are values (names) which are set by the state machine in certain situations, i.e. when entering a state, exiting a state or as input actions.

For instance to represent a simple on/off switch the following VFSM can be defined[1]:

```
<VFSM>
      <Type>switch</Type>
      <Object>
            <Name>switch1</Name>
      </Object>
      <IOid>
            <Input>
                  <Name>high</Name>
                  <Value>1</Name>
            </Input>
            <Input>
                  <Name>low</Name>
                  <Value>0</Name>
            </Input>
      </IOid>
      <State>
            <Name>HIGH</Name>
      </State>
      <State>
            <Name>LOW</Name>
      </State>
</VFSM>
```

The names “high” and “low” represent the virtual input of the switch VFSM. The state names “HIGH” and “LOW” can be used as its virtual output.

One can define a range of state machines which are commonly used, to represent such items as timers, digital input, digital output etc. Those state machines don’t need to be defined in a VFSMML message, as their virtual inputs and outputs are well known on the target system. VFSMML defines a set of such known (predefined)

[1] This definition is not complete, e.g. it does not contain the behaviour description.

VFSM. See Appendix A for more details. To use a predefined VFSM, only its object name definition is required:

```
<VFSM type="predefined">
      <Type>DI</Type>
      <Object>
            <Name>switch1</Name>
      </Object>
</fVFSM>
```

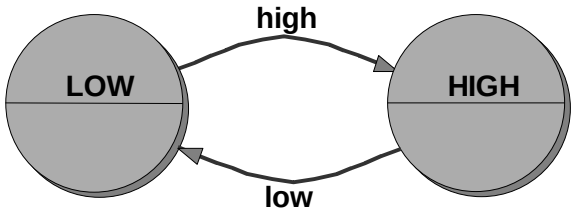



**Figure 4: Digital Input (DI) object state machine**

The "DI" VFSM has exactly the same virtual input and output as the previously defined "switch".

## 2.3 State machine behavior

The behavior of a state machine is given by the description of its states. Each state can set output values (names) based on certain conditions. The conditions are logical expressions[1] created out of the input values (names). Entering or exiting a state can also be used as a kind of condition to set an output value. For each state any condition based transitions can also be specified. To support logical expressions to build conditions, MathML syntax is used.

For instance to specify that the following state machine shall change to state "starting engine" when the air conditioning is running and the start switch is on, the description below can be used:

```
<VFSM>
      <Type>Engine</Type>
      ...
      <State>
            <Transition>
                  <Condition>
                        <apply>
                              </and>
                              <ci>aircond_running</ci>
                              <ci>switch_on</ci>
                        </apply>
                  </Condition>
                  <StateName>StartingEngine</StateName>
            </Transition>
      </State>
</VFSM>
```

[1] See also section 1.4 Positive-logic algebra

The input names used for conditions and output names used for actions are based on objects defined for the given VFSM. For instance the input name ”switch_on” could be created using the definition given in previous chapter 2.2:

```
<VFSM>
      <Type>Engine</Type>
      <Object>
            <Name>Engine1</Name>
            <Property>
                  <Name>switch</Name>
                  <Value>switch1</Value>
            </Property>
      </Object>
      <IOid>
            <Name>switch</Name>
            <Input>
                  <Name>switch_on</Name>
                  <Value>high</Value>
            </Input>
      </IOid>
      ...
</VFSM>
```

## 2.4 VFSMML Examples

### 2.4.1 Microwave Oven

Below a simple example of a microwave oven control is presented. The requirements are as following:

The oven has a ‘Run’ push button to start (apply the power) and a timer that determines the cooking length. Cooking can be interrupted at any time by opening the oven door. After closing the door the cooking is continued. Cooking is terminated when the timer elapses. When the cooking is in progress and also when the door is opened a lamp inside the oven is switched on, otherwise when the door is closed the lamp is switched off.

The control system has the following inputs:

**Run** push button - when activated starts cooking,

**Timer** - while this runs keep on cooking,

**Door** sensor - can be true (door closed) or false (door open).

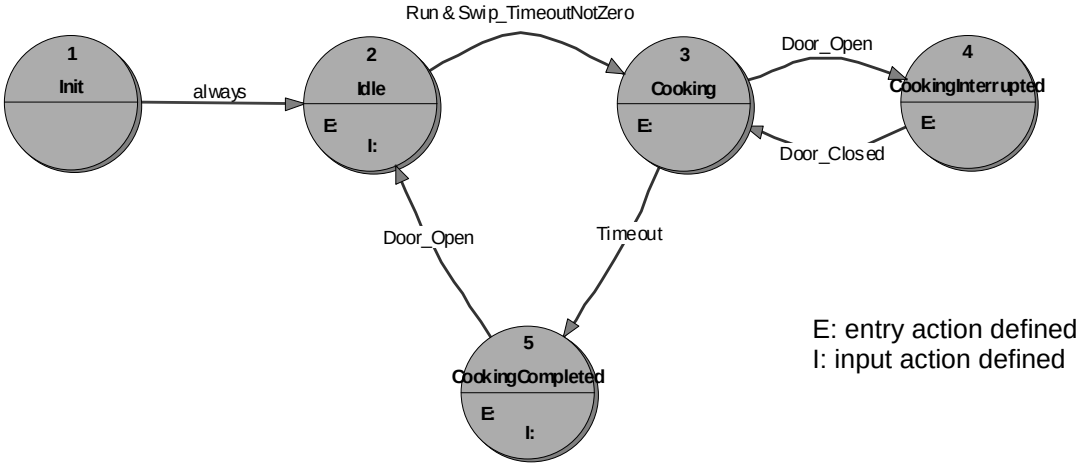
And the following outputs:

**Power** - can be true (power on) or false (power off),

**Lamp** - can be true (lamp on) or false (lamp off).

The knobs to set the power and timeout values are irrelevant for the control state machine. The behavior of the microwave oven control is determined by the Run push button, Timer and Door sensor.

For this specification the following state transition diagram can be designed:




The table below shows the set of objects given for the microwave oven specification. Based on those objects, a dictionary of input and output names can be defined (see also 2.2).

| Object Name | Object Type | Description |
|---|---|---|
| Timer | TI | A timer; While this timer runs, keep on cooking |
| Di_Door | DI | A digital input (high/low); The sensor which shows if the door is opened or closed |
| Di_Run | DI | A digital input (high/low); The push button to activate the cooking |
| Do_Lamp | DO | The lamp inside the oven |
| Do_Power | DO | A digital output (high/low); The power button |
| Swip_Timeout | SWIP | Helping object to support the timer |

**Table 1: Microwave Oven - Object Name Dictionary**

| Input Name | Input Value | Object Name |
|---|---|---|
| always[1] | | |
| Timeout | Over | Timer |
| Door_Closed | Low | Di_Door |
| Door_Open | High | Di_Door |
| Run | High | Di_Run |
| Stop | Low | Di_Run |
| TimeoutNotZero | In | Swip_Timeout |

**Table 2: Microwave Oven - Input Name Dictionary**

| Output Name | Output Value | Object Name |
|---|---|---|
| Timer_Reset | Reset | Timer |
| Timer_Start | Start | Timer |
| Timer_Stop | Stop | Timer |
| LampOff | Low | Do_Lamp |
| LampOn | High | Do_Lamp |

[1] This name (=condition) exists always

| | | |
|---|---|---|
| PowerOff | Low | Do_Power |
| PowerOn | High | Do_Power |
| Swip_Timeout_On | On | Swip_Timeout |

**Table 3: Microwave Oven - Output Name Dictionary**

Besides the transition conditions as shown in the state transition diagram above, an output table is given to completely define the FSM[1]:

| State | Condition | Output | Description |
|---|---|---|---|
| Init | The VFSM starts here | - | In the initialization phase no actions are required |
| Idle | Entering the state (entry action) | Swip_Timeout_On | The swip object has to be activated |
| | Door_Closed | LampOff | |
| | Door_Open | LampOn | |
| Cooking | Entering the state (entry action) | LampOn<br>PowerOn<br>Timer_Start | Any time we enter this state, the timer is started but not reset |
| CookingInterrupted | Entering the state (entry action) | PowerOff<br>Timer_Stop | Any time we enter this state, the timer is stoped but not reset |
| CookingCompleted | Entering the state (entry action) | LampOff<br>PowerOff<br>Timer_Reset | The timer is reset only when the cooking is completed |
| | Door_Open | LampOn | |

**Table 4: Microwave Oven - Output Conditions**

The mark-up representation of the microwave oven is given below:

```
<?xml version="1.0" ?>
<?xml-stylesheet href="vfsmml.xsl" type="text/xsl"?>
<!DOCTYPE vfsmml SYSTEM "vfsmml.dtd" >
<vfsmml project="true">
     <Name>MWOven</Name>
     <VFSM type="predefined">
           <Type>TI</Type>
           <Object>
                 <Name>MW:Ti:CookingTime</Name>
                 <Property>
                       <Name>Const</Name>
                       <Value>MW:Ni:CookingTime</Value>
                 </Property>
                 <Property>
                       <Name>Clock</Name>
                       <Value>sec</Value>
                 </Property>
           </Object>
     </VFSM>

     <VFSM type="predefined">
           <Type>DI</Type>
           <Object>
                 <Name>MW:Di:Door</Name>
           </Object>
```

[1] Actually, there is no standard notation for complete specification of s state machine behavior. For instance StateWORKS uses a special transition table for this purpose.

```
      <Object>
            <Name>MW:Di:Run</Name>
      </Object>
</VFSM>

<VFSM type="predefined">
      <Type>DO</Type>
      <Object>
            <Name>MW:Do:Lamp</Name>
      </Object>
      <Object>
            <Name>MW:Do:Power</Name>
      </Object>
</VFSM>

<VFSM type="predefined">
      <Type>NI</Type>
      <Object>
            <Name>MW:Ni:CookingTime</Name>
            <Property>
                  <Name>Format</Name>
                  <Value>int</Value>
            </Property>
            <Property>
                  <Name>Unit</Name>
                  <Value>sec</Value>
            </Property>
            <Property>
                  <Name>ScaleMode</Name>
                  <Value>Lin</Value>
            </Property>
            <Property>
                  <Name>ScaleFactor</Name>
                  <Value>1</Value>
            </Property>
            <Property>
                  <Name>Offset</Name>
                  <Value>0</Value>
            </Property>
            <Property>
                  <Name>Threshold</Name>
                  <Value>0</Value>
            </Property>
      </Object>
</VFSM>

<VFSM type="predefined">
      <Type>SWIP</Type>
      <Object>
            <Name>MW:Swip:Timeout</Name>
            <Property>
                  <Name>Input</Name>
                  <Value>MW:Ni:CookingTime</Value>
            </Property>
            <Property>
                  <Name>LimitLow</Name>
                  <Value>1</Value>
            </Property>
            <Property>
                  <Name>LimitHigh</Name>
                  <Value>10000</Value>
            </Property>
```

```
      </Object>
</VFSM>

<VFSM type="predefined">
      <Type>PAR</Type>
      <Object>
            <Name>MW:Par:CookingTime</Name>
            <Property>
                  <Name>Category</Name>
                  <Value>PP</Value>
            </Property>
            <Property>
                  <Name>Format</Name>
                  <Value>int</Value>
            </Property>
            <Property>
                  <Name>Unit</Name>
                  <Value>sec</Value>
            </Property>
            <Property>
                  <Name>LimitLow</Name>
                  <Value>0</Value>
            </Property>
            <Property>
                  <Name>LimitHigh</Name>
                  <Value>0</Value>
            </Property>
            <Property>
                  <Name>InitValue</Name>
                  <Value>0</Value>
            </Property>
      </Object>
</VFSM>

<VFSM type="vfsm">
      <Type>MWOven</Type>
      <Prefix>MEA</Prefix>
      <Object>
            <Name>MW</Name>
            <Property>
                  <Name>MyCmd</Name>
                  <Value></Value>
            </Property>
            <Property>
                  <Name>Timer</Name>
                  <Value>MW:Ti:CookingTime</Value>
            </Property>
            <Property>
                  <Name>Di_Door</Name>
                  <Value>MW:Di:Door</Value>
            </Property>
            <Property>
                  <Name>Di_Run</Name>
                  <Value>MW:Di:Run</Value>
            </Property>
            <Property>
                  <Name>Do_Lamp</Name>
                  <Value>MW:Do:Lamp</Value>
            </Property>
            <Property>
                  <Name>Do_Power</Name>
                  <Value>MW:Do:Power</Value>
            </Property>
```

```
      <Property>
            <Name>Swip_Timeout</Name>
            <Value>MW:Swip:Timeout</Value>
      </Property>
</Object>

<IOid>
      <Name>MyCmd</Name>
      <Type>CMD-IN</Type>
</IOid>
<IOid>
      <Name>Timer</Name>
      <Type>TI</Type>
      <Input>
            <Name>Timeout</Name>
            <Value>OVER</Value>
      </Input>
      <Output>
            <Name>Timer_Reset</Name>
            <Value>Reset</Value>
      </Output>
      <Output>
            <Name>Timer_Start</Name>
            <Value>Start</Value>
      </Output>
      <Output>
            <Name>Timer_Stop</Name>
            <Value>Stop</Value>
      </Output>
</IOid>
<IOid>
      <Name>Di_Door</Name>
      <Type>DI</Type>
      <Input>
            <Name>Door_Closed</Name>
            <Value>LOW</Value>
      </Input>
      <Input>
            <Name>Door_Open</Name>
            <Value>HIGH</Value>
      </Input>
</IOid>
<IOid>
      <Name>Di_Run</Name>
      <Type>DI</Type>
      <Input>
            <Name>Di_Run</Name>
            <Value>HIGH</Value>
      </Input>
      <Input>
            <Name>Di_Stop</Name>
            <Value>LOW</Value>
      </Input>
</IOid>
<IOid>
      <Name>Do_Lamp</Name>
      <Type>DO</Type>
      <Output>
            <Name>Do_LampOff</Name>
            <Value>Low</Value>
      </Output>
      <Output>
```

```
            <Name>Do_LampOn</Name>
            <Value>High</Value>
      </Output>
</IOid>
<IOid>
      <Name>Do_Power</Name>
      <Type>DO</Type>
      <Output>
            <Name>Do_PowerOff</Name>
            <Value>Low</Value>
      </Output>
      <Output>
            <Name>Do_PowerOn</Name>
            <Value>High</Value>
      </Output>
</IOid>
<IOid>
      <Name>Swip_Timeout</Name>
      <Type>SWIP</Type>
      <Input>
            <Name>Swip_TimeoutNotZero</Name>
            <Value>IN</Value>
      </Input>
      <Output>
            <Name>Swip_Timeout_On</Name>
            <Value>On</Value>
      </Output>
</IOid>

<State>
      <Description>Usually, the state machine goes
directly to its Idle state.</Description>
      <Name>Init</Name>
      <Transition>
            <Condition>
                  <ci>always</ci>
            </Condition>
            <StateName>Idle</StateName>
      </Transition>
</State>
<State>
      <Description>Entering the state the state machine
switches off the power and stops the timer. The cooking
continues when the door is closed.</Description>
      <Name>CookingInterrupted</Name>
      <EntryAction>Do_PowerOff</EntryAction>
      <EntryAction>Timer_Stop</EntryAction>
      <Transition>
            <Condition>
                  <ci>Door_Closed</ci>
            </Condition>
            <StateName>Cooking</StateName>
      </Transition>
</State>
<State>
      <Description> Entering the state the state machine
switches on the lamp and applies the power. In addition,
it starts the timer which timeout determines the cooking
time.

      The cooking can be interrupted at any time by
opening the door.</Description>
      <Name>Cooking</Name>
```

```
      <EntryAction>Do_LampOn</EntryAction>
      <EntryAction>Do_PowerOn</EntryAction>
      <EntryAction>Timer_Start</EntryAction>
      <Transition>
            <Condition>
                  <ci>Door_Open</ci>
            </Condition>
            <StateName>CookingInterrupted</StateName>
      </Transition>
      <Transition>
            <Condition>
                  <ci>Timeout</ci>
            </Condition>
            <StateName>CookingCompleted</StateName>
      </Transition>
</State>
<State>
      <Description> Entering the state the Run signal is
cleared. Opening and closing the door switches the lamp
on and off. If the Run signal becomes active and the
Timeout value is not zero the state machine goes to the
state Cooking.</Description>
      <Name>Idle</Name>
      <EntryAction>Swip_Timeout_On</EntryAction>
      <InputAction>
            <Condition>
                  <ci>Door_Closed</ci>
            </Condition>
            <Action>Do_LampOff</Action>
      </InputAction>
      <InputAction>
            <Condition>
                  <ci>Door_Open</ci>
            </Condition>
            <Action>Do_LampOn</Action>
      </InputAction>
      <Transition>
            <Condition>
                  <apply>
                        <and/>
                        <ci>Di_Run</ci>
                        <ci>Swip_TimeoutNotZero</ci>
                  </apply>
            </Condition>
            <StateName>Cooking</StateName>
      </Transition>
</State>
<State>
      <Description>Entering the state the state machine
switches off the lamp and the power. In addition, it
stops the timer. Opening the door switches the lamp on
and the state machine returns to the Idle
state.</Description>
      <Name>CookingCompleted</Name>
      <EntryAction>Do_LampOff</EntryAction>
      <EntryAction>Do_PowerOff</EntryAction>
      <EntryAction>Timer_Reset</EntryAction>
      <InputAction>
            <Condition>
                  <ci>Door_Open</ci>
            </Condition>
            <Action>Do_LampOn</Action>
```

```
            </InputAction>
            <Transition>
                  <Condition>
                        <ci>Door_Open</ci>
                  </Condition>
                  <StateName>Idle</StateName>
            </Transition>
      </State>
</VFSM>

<VFSM type="unit">
      <Type>DI16P</Type>
      <Prefix>DI1</Prefix>
      <Object>
            <Name>MW:DI16P</Name>
            <Property>
                  <Name>CommPort</Name>
                  <Value></Value>
            </Property>
            <Property>
                  <Name>PhysAddr</Name>
                  <Value>1</Value>
            </Property>
            <Property>
                  <Name>Di0</Name>
                  <Value>MW:Di:Door</Value>
            </Property>
            <Property>
                  <Name>Di1</Name>
                  <Value>MW:Di:Run</Value>
            </Property>
      </Object>

      <IOid>
            <Name>Di0</Name>
            <Type>DI</Type>
      </IOid>
      <IOid>
            <Name>Di1</Name>
            <Type>DI</Type>
      </IOid>
</VFSM>

<VFSM type="unit">
      <Type>DO16P</Type>
      <Prefix>DO1</Prefix>
            <Object>
            <Name>MW:DO16P</Name>
            <Property>
                  <Name>CommPort</Name>
                  <Value></Value>
            </Property>
            <Property>
                  <Name>PhysAddr</Name>
                  <Value>3</Value>
            </Property>
            <Property>
                  <Name>Do0</Name>
                  <Value>MW:Do:Power</Value>
            </Property>
            <Property>
                  <Name>Do1</Name>
                  <Value>MW:Do:Lamp</Value>
```

```
                </Property>
            </Object>
            <IOid>
                <Name>Do0</Name>
                <Type>DO</Type>
            </IOid>
            <IOid>
                <Name>Do1</Name>
                <Type>DO</Type>
            </IOid>
        </VFSM>

        <VFSM type="unit">
            <Type>NI4</Type>
            <Prefix>NI4</Prefix>
            <Object>
                <Name>MW:NI4</Name>
                <Property>
                    <Name>CommPort</Name>
                    <Value></Value>
                </Property>
                <Property>
                    <Name>PhysAddr</Name>
                    <Value>5</Value>
                </Property>
                <Property>
                    <Name>Ni0</Name>
                    <Value>MW:Ni:CookingTime</Value>
                </Property>
            </Object>
            <IOid>
                <Name>Ni0</Name>
                <Type>NI</Type>
            </IOid>
        </VFSM>
    </vfsmml>
```

### 2.4.2 Simple Master-Slave Configuration

The following example shows how to present dependencies between various state machines. The predefined VFSM “CMD” is used as an input and as an output and is designed for inter-VFSM communication (see also chapter A.02 in Appendix A).

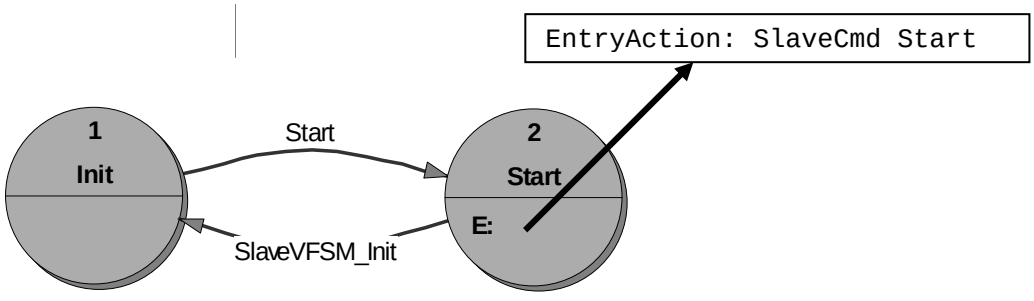


**Figure 5: Master VFSM**

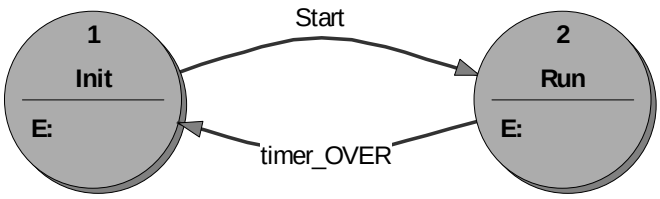



**Figure 6: Slave FVSM**

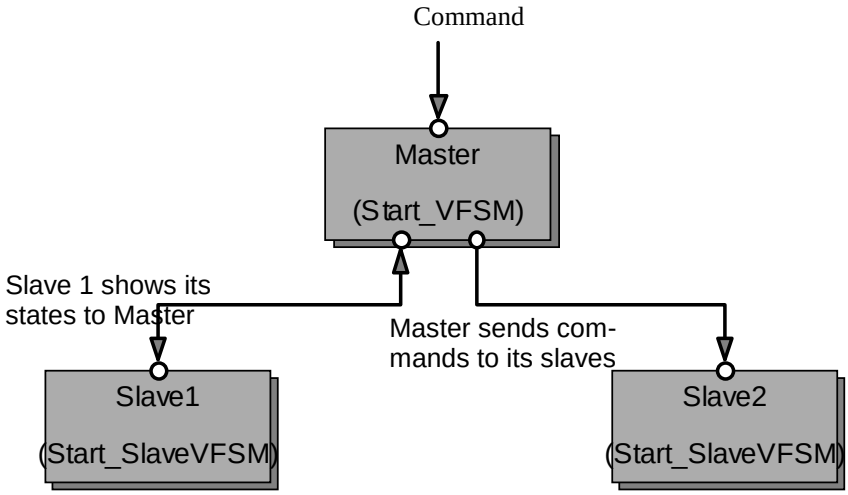



**Figure 7: Dependencies between Master and Slave**

The table below shows the set of objects given for the master VFSM specification. Based on those objects, again, a set of input and output names can be defined.

| Object Name | Object Type | Description |
|---|---|---|
| MyCmd | CMD | User command (incoming command) |
| SlaveVFSM | VFSM | Slave VFSM definition |
| SlaveCmd | CMD | Command to the slave VFSM (outgoing command |

**Table 5: Master-Slave - Object Name Dictionary**

| Input Name | Input Value | Object Name |
|---|---|---|
| Start | 1 | MyCmd |
| SlaveVFSM_Init | Init (Slave state) | SlaveVFSM |

**Table 6: Master-Slave - Input Name Dictionary**

| Output Name | Output Value | Object Name |
|---|---|---|
| SlaveCmd_Start | 1 | SlaveCmd |

**Table 7: Master-Slave - Output Name Dictionary**

The table below shows the set of objects given for the slave VFSM specification. Both slave VFSM instances are of the same VFSM type. Based on those objects, again, a set of input and output names can be defined.

| Object Name | Object Type | Description |
|---|---|---|
| MyCmd | CMD-IN | Command from master |
| timer | TI | A timer |

**Table 8: Master-Slave - Object Name Dictionary**

| Input Name | Input Value | Object Name |
|---|---|---|
| Start | 1 | MyCmd |
| timer_OVER | OVER | timer |

**Table 9: Master-Slave - Input Name Dictionary**

| Output Name | Output Value | Object Name |
|---|---|---|
| timer_ResetStart | ResetStart | timer |
| timer_Stop | Stop | timer |

**Table 10: Master-Slave - Output Name Dictionary**

The mark-up representation of the master-slave example is given below:

```
<?xml version="1.0" ?>
<?xml-stylesheet href="vfsmml.xsl" type="text/xsl"?>
<!DOCTYPE vfsmml SYSTEM "vfsmml.dtd" >
<vfsmml project="true">
     <Name>MasterSlave</Name>
     <VFSM type="predefined">
          <Type>CMD</Type>
          <Object>
               <Name>Master:MyCmd</Name>
               <Property>
                    <Name>Type</Name>
                    <Value></Value>
               </Property>
          </Object>
          <Object>
               <Name>Slave:MyCmd</Name>
               <Description>start_slave vfsm</Description>
               <Property>
                    <Name>Type</Name>
                    <Value></Value>
               </Property>
          </Object>
     </VFSM>

     <VFSM type="predefined">
          <Type>TI</Type>
          <Object>
               <Name>Timer1</Name>
               <Property>
                    <Name>Const</Name>
                    <Value>10</Value>
```

```
            </Property>
            <Property>
                  <Name>Clock</Name>
                  <Value>sec</Value>
            </Property>
      </Object>
      <Object>
            <Name>Timer2</Name>
            <Property>
                  <Name>Const</Name>
                  <Value>20</Value>
            </Property>
            <Property>
                  <Name>Clock</Name>
                  <Value>sec</Value>
            </Property>
      </Object>
</VFSM>

<VFSM type="vfsm">
      <Type>Start_VFSM</Type>
      <Prefix>MAS</Prefix>
      <Object>
            <Name>Master</Name>
            <Property>
                  <Name>MyCmd</Name>
                  <Value>Master:MyCmd</Value>
            </Property>
            <Property>
                  <Name>SlaveVFSM</Name>
                  <Value>Slave1</Value>
            </Property>
            <Property>
                  <Name>SlaveCmd</Name>
                  <Value>Slave:MyCmd</Value>
            </Property>
      </Object>

      <IOid>
            <Name>MyCmd</Name>
            <Type>CMD-IN</Type>
            <Input>
                  <Name>Start</Name>
                  <Value>1</Value>
            </Input>
      </IOid>
      <IOid>
            <Name>SlaveVFSM</Name>
            <Type>VFSM</Type>
            <Description>start_slave vfsm</Description>
      </IOid>
      <IOid>
            <Name>SlaveCmd</Name>
            <Type>CMD-OUT</Type>
            <Description>start_slave vfsm</Description>
            <Output>
                  <Name>SlaveCmd_Start</Name>
                  <Value>1</Value>
            </Output>
      </IOid>

      <State>
            <Name>Init</Name>
```

```
                <Transition>
                        <Condition>
                                <ci>Start</ci>
                        </Condition>
                        <StateName>Start</StateName>
                </Transition>
        </State>
        <State>
                <Name>Start</Name>
                <EntryAction>SlaveCmd_Start</EntryAction>
                <Transition>
                        <Condition>
                                <ci>SlaveVFSM_Init</ci>
                        </Condition>
                        <StateName>Init</StateName>
                </Transition>
        </State>
</VFSM>

<VFSM type="vfsm">
        <Type>Start_SlaveVFSM</Type>
        <Prefix>SLA</Prefix>
        <Object>
                <Name>Slave1</Name>
                <Description>start_slave vfsm</Description>
                <Property>
                        <Name>MyCmd</Name>
                        <Value>Slave:MyCmd</Value>
                </Property>
                <Property>
                        <Name>timer</Name>
                        <Value>Timer1</Value>
                </Property>
        </Object>
        <Object>
                <Name>Slave2</Name>
                <Property>
                        <Name>MyCmd</Name>
                        <Value>Slave:MyCmd</Value>
                </Property>
                <Property>
                        <Name>timer</Name>
                        <Value>Timer2</Value>
                </Property>
        </Object>

        <IOid>
                <Name>MyCmd</Name>
                <Type>CMD-IN</Type>
                <Input>
                        <Name>Start</Name>
                        <Value>1</Value>
                </Input>
        </IOid>
        <IOid>
                <Name>timer</Name>
                <Type>TI</Type>
                <Input>
                        <Name>timer_OVER</Name>
                        <Value>OVER</Value>
                </Input>
                <Output>
```

```
                        <Name>timer_ResetStart</Name>
                        <Value>ResetStart</Value>
                  </Output>
                  <Output>
                        <Name>timer_Stop</Name>
                        <Value>Stop</Value>
                  </Output>
            </IOid>

            <State>
                  <Name>Init</Name>
                  <EntryAction>timer_Stop</EntryAction>
                  <Transition>
                        <Condition>
                              <ci>Start</ci>
                        </Condition>
                        <StateName>Run</StateName>
                  </Transition>
            </State>
            <State>
                  <Name>Run</Name>
                  <EntryAction>timer_ResetStart</EntryAction>
                  <Transition>
                        <Condition>
                              <ci>timer_OVER</ci>
                        </Condition>
                        <StateName>Init</StateName>
                  </Transition>
            </State>
      </VFSM>
</vfsmml>
```

## 2.5 VFSMML Syntax and Grammar

VFSMML is an application of Extensible Markup Language (XML), and as such its syntax is governed by the rules of XML syntax, and its grammar is in part specified by the Document Type Definition (DTD). In other words, the details of using tags, attributes, entity references and so on are defined in the XML language specification and the details about VFSMML element and attribute names, which elements can be nested inside each other, and so on are specified in the VFSMML DTD in A.03.

# 3 VFSMML Markup

## 3.1 Element Usage Guide

### 3.1.1 Summary of Elements

The Figure 8 below gives an overview about all defined tags and their hierarchy. All bold italic tags can appear many times inside their parent tags. The (o) means, the element is optional, (a) means there is an attribute defined.

The <Condition> tag can be a single name or a logical expression given using notation as defined in MathML. Here only following tags are used: <apply>, <and>, <or> and <ci>. In the following chapter more accurate description is given.

The dependencies between certain tags are explained in section 3.2.

- ***vfsmml (a)***
  - ⇨ Name (o)
  - ⇨ Description (o)
  - ⇨ ***VFSM (a)***
    - ⇨ Type
    - ⇨ Description (o)
    - ⇨ Prefix (o)
    - ⇨ ***Object (o)***
      - ⇨ Name
      - ⇨ Description (o)
      - ⇨ ***Property (o)***
        - ⇨ Name
        - ⇨ Value
    - ⇨ ***IOid (o)***
      - ⇨ Name
      - ⇨ Type
      - ⇨ Description (o)
      - ⇨ ***Input (o)***
        - ⇨ Init (o)
        - ⇨ Name
        - ⇨ Value
      - ⇨ ***Output (o)***
        - ⇨ Name
        - ⇨ Value
    - ⇨ ***State (a,o)***
      - ⇨ Description (o)
      - ⇨ Name (o)
      - ⇨ ***EntryAction (o)***
      - ⇨ ***ExitAction (o)***
      - ⇨ ***InputAction (o)***
        - ⇨ Condition
        - ⇨ Action
      - ⇨ ***Transition (o)***
        - ⇨ Condition
        - ⇨ StateName
        - ⇨ Action (o)

**Figure 8 VFSMML tags and their hierarchy**

### 3.1.2 Overview of Syntax and Usage

In the following all defined tags are listed alphabetically.

***<Action> Tag***
The VFSM virtual output.

Each output name is defined in the <IOid> tag section and can be used in the <State> tag section as <Action>. Each <Action> tag value must be first defined in the <IOid> tag section (IOid – Output – Name) before it can be used. The <Action> tag is obligatory in the <InputAction> tag section and optional in the <Transition> tag section.

Example:

In following the action “Start” will be set when entering the state or when receiving the command “CmdStart”.

```
<IOid>
      ...
      <Output>
            <Name>Start</Name>
            <Value>HIGH</Value>
      </Output>
</IOid>
...
<State>
      <EntryAction>Start</EntryAction>
      <InputAction>
            <Condition>
                  <ci>CmdStart</ci>
            </Condition>
            <Action>Start</Action>
      </InputAction>
</State>
```

***<Condition> Tag***
The definition of conditions to perform a transition or execute an input action.

The value is a logical expression using notation as defined in MathML. The <Condition> tag is obligatory in the <InputAction> and <Transition> tag section.

Examples:

1. Single input condition. Do something when the input name “Start” is set:
```
<Condition>
      <ci>Start</ci>
</Condition>
```

2. OR condition. Do something, when “Stop” or “Door_Open” is set:
```
<Condition>
      <apply>
            </or>
            <ci>Stop</ci>
            <ci>Door_Open</ci>
      </apply>
</Condition>
```

3. OR and AND condition. Do something, when “Start” and “Door_Open” or “Start” and “Timer_Over” is set, i.e. “Start AND (Door_Open OR Time_Over)”:

```
<Condition>
      <apply>
            </and>
            <ci>Start</ci>
            <apply>
                  </or>
                  <ci>Door_Open</ci>
                  <ci>Timer_Over</ci>
            </apply>
      </apply>
</Condition>
```

***<Description> Tag***
An optional comment allowed for certain tags.

Following tags can contain the optional <Description> sub tag: <vfsmml>, <VFSM>, <Object>, <IOid> and <State>.

***<EntryAction> Tag***
VFSM output name, set when entering a state.

See <Action> tag for more information. <EntryAction> is an optional tag in the <State> tag section only. Any number of <EntryAction> tags is allowed inside a <State> tag.

***<ExitAction> Tag***
VFSM output name, set when exiting a state.

See <Action> tag for more information. <ExitAction> is an optional tag in the <State> tag section only. Any number of <ExitAction> tags is allowed inside a <State> tag.

***<Init> Tag***
Specifies input names valid after the VFSM start-up.

Can be “true” or “false”. The default value is “false”. If “true” is set, the current name is active at the VFSM start-up. <Init> is an optional tag inside the <Input> tag section only. For instance the name “always” should always be active as per definition.

***<Input> Tag***
An input name definition based on possible values of a given Object.

There is any number of <Input> tags allowed inside an <IOid> tag. Each <Input> tag contains three sub tags: <Init> (optional, default value is “false”), <Name> and <Value>.

Examples:

1. Input based on a predefined VFSM (DI). The instance DI_1 is created on the type DI. For this instance the name “door_closed” is defined to represent the value “LOW” of the IOid “Di_door” (see also definition of DI in Appendix A).

```
<VFSM type=”predefined”>
      <Type>DI</Type>
      <Object>
            <Name>DI_1</Name>
      </Object>
</VFSM>
<VFSM>
      <Name>MyVFSM</Name>
      <Object>
            <Name>MyVFSM1</Name>
            <Property>
                  <Name>Di_door</Name>
                  <Value>DI_1</Value>
            </Property>
      </Object>
      <IOid>
            <Name>Di_door</Name>
            <Type>DI</Type>
            <Input>
                  <Name>door_closed</Name>
                  <Value>LOW</Value>
            </Input>
      </IOid>
      ...
</VFSM>
```

2. Input based on a VFSM. The instance vfsm_A1 is created on the type vfsm_type_A. For this instance the name “slave_stop” is defined to represent the state “stop” of the vfsm_A1 VFSM.

```
<VFSM>
      <Type>vfsm_type_A</Type>
      <Object>
            <Name>vfsm_A1</Name>
      </Object>
      <State>start</State>
      <State>stop</State>
</VFSM>
<VFSM>
      <Type>vfsm_type_B</Type>
      <Object>
            <Name>vfsm_B1</Name>
            <Property>
                  <Name>A1</Name>
                  <Value>vfsm_A1</Value>
            </Property>
      </Object>
      <IOid>
            <Name>A1</Name>
            <Type>vfsm_type_A</Type>
            <Input>
                  <Name>slave_stop</Name>
                  <Value>stop</Value>
            </Input>
      </IOid>
      ...
</VFSM>
```

***<InputAction> Tag***
VFSM output name, set when certain conditions (input names) are given.

One <InputAction> tag contains two mandatory sub tags: <Condition> and <Action>. Any number of <InputAction> tags is allowed inside each <State> tag. The <InputAction> tag is optional and possible only inside the <State> tag.

***<IOid> Tag***
An identifier of any kind of VFSM type used inside certain VFSM (Object – Property – Name).

Any number of <IOid> tags is allowed inside a <VFSM> tag. One <IOid> tag may contain the following sub tags: <Name>, <Type>, <Description>, <Input> and <Output>. The <Input> and <Output> tags define the names of its possible values. The <Type> specifies the object type used for this definition (i.e. another VFSM or predefined VFSM).

Example:

An IOid based on a DI object. For instance the name “on” is defined to represent the value “HIGH” (see also definition of DI in Appendix A).

```
<VFSM>
      <IOid>
            <Name>Switch</Name>
            <Type>DI</Type>
            <Input>
                  <Name>on</Name>
                  <Value>HIGH</Value>
            </Input>
      </IOid>
</VFSM>
```

***<Name> Tag***
Defines the name of the content specified inside its parent tag.

The <Name> tag is used with the following tags: <vfsmml>, <Object>, <Property>, <IOid>, <Input>, <Output> and <State>. Each <vfsmml> has a unique name. Each <Object> has a unique name inside its <vfsmml> tag. Each <IOid> , <Input>, <Output> and <State> has a unique name inside its <VFSM> tag.

***<Object> Tag***
Creates and describes the properties of an incarnation of a VFSM.

The <Object> tag contains the following sub tags: <Name>, <Description> and <Property>. Object properties depend on its VFSM. In Appendix A all predefined VFSM and their properties for a default VFSMML document are listed. The <Object> tag is mandatory inside the <VFSM> tag, but any number of <Object> tags is allowed. See also section 3.2 for more information about tag dependencies.

***<Output> Tag***
An output name definition based on possible values of the used object.

There is any number of <Output> tags allowed inside an <IOid> tag. Each <Output> tag contains two sub tags: <Name> and <Value>.

Examples:

Output based on a predefined VFSM (DO). The instance DO_1 is created on the type DO. For this instance the name “close_door” is defined to represent the value “HIGH” of the IOid “door” (see also definition of DO in Appendix A).

```
<VFSM type=”predefined”>
      <Type>DO</Type>
      <Object>
            <Name>DO_1</Name>
      </Object>
</VFSM>

<VFSM>
      <Type>MyVFSM</Type>
      <Object>
            <Name>MyVFSM1</Name>
            <Property>
                  <Name>door</Name>
                  <Value>DO_1</Value>
            </Property>
      </Object>
      <IOid>
            <Name>door</Name>
            <Type>DO</Type>
            <Output>
                  <Name>close_door</Name>
                  <Value>HIGH</Value>
            </Output>
      </IOid>
</VFSM>
```

***<Prefix> Tag***
Prefix of a type definition.

Each VFSM type (besides predefined VFSM) has a unique three-letter prefix inside its <vfsmml> tag.

***<Property> Tag***
Defines a property of a VFSM.

Properties are parameters which stay constant at least for the execution time of the entire system. Properties define also all the objects on which the current VFSM is based (other VFSMs or predefined VFSMs). There is any number of <Property> tags allowed inside an <Object> tag.

Examples:

1. Constant parameter: definition of scanner step-motor properties: the motor is only responsible for the x-direction and the maximum number of steps is 100.

```
<VFSM>
      <Type>motor</Type>
      <Object>
```

```
            <Name>motor_x</Name>
            <Property>
                  <Name>max_x</Name>
                  <Value>100</Value>
            </Property>
            <Property>
                  <Name>max_y</Name>
                  <Value>0</Value>
            </Property>
      </Object>
      ...
</VFSM>
```

2. Objects on which the current VFSM is based: the VFSM contains a timer and is used two times in the system. Each VFSM copy contains an own timer (TI object) with different properties:

```
<VFSM type="predefined">
      <Type>TI</Type>
      <Object>
            <Name>timer1</Name>
            ...
      </Object>
      <Object>
            <Name>timer2</Name>
            ...
      </Object>
</VFSM>
...
<VFSM>
      <Type>Slave</Type>
      <Object>
            <Name>Slave1</Name>
            <Property>
                  <Name>timer</Name>
                  <Value>timer1</Value>
            </Property>
      </Object>
      <Object>
            <Name>Slave2</Name>
            <Property>
                  <Name>timer</Name>
                  <Value>timer2</Value>
            </Property>
      </Object>
      ...
</VFSM>
```

***<State> Tag***

Describes one state of a VFSM.

There is any number of <State> tags allowed inside a <VFSM> tag. One <State> tag contains following sub tags: <Description>, <Name>, <EntryAction>, <ExitAction>, <InputAction> and <Transition>. The <State> tag contains one mandatory attribute "always", which can be "true" or "false". The default value of this attribute is "false".

***<StateName> Tag***

One state name from the set of all states of the current VFSM.

See also section 3.2 for more information about tag dependencies.

***<Transition> Tag***
The state transition definition.

A <Transition> tag is optional and contains three sub tags: <Condition>, <Action> (optional) and <StateName>. Any number of <Transition> tags is allowed inside a <State> tag.

***<Type> Tag***
Defines a type name of a VFSM.

Based on this name any number of instances of a given VFSM can be defined. The name of an object is then an instance of this type. The <Type> tag is mandatory.

For instance, a system uses 10 timers of type TI. Then there are 10 objects defined:

```
<VFSM type="predefined">
      <Type>TI</TI>
      <Object>
            <Name>timer1</Name>
      </Object>
      <Object>
            <Name>timer2</Name>
      </Object>
      ...
      <Object>
            <Name>timer10</Name>
      </Object>
</VFSM>
```

***<Value> Tag***
The value represented by the name of <Property>, <Input> or <Output> tag.

For <Input> and <Output>, the value must be from the range of values defined in the appropriate <VFSM> tag. For <Property> of a predefined VFSM only supported property values (see predefined VFSM definitions in Appendix A) are allowed. For a <Property> section of a new VFSM tag any values are possible.

***<VFSM> Tag***
Announces a VFSM definition.

There is any number of <VFSM> tags possible inside a <vfsmml> tag. Each <VFSM> tag contains following sub tags: <Type>, <Object>, <Description>, <Prefix>, <IOid> and <State>. The <VFSM> tag contains one mandatory attribute "type", which can be "vfsm", "predefined" or "unit". The default value of this attribute is "vfsm". For attributes "predefined" and "unit" the sub-tag <State> is not allowed.

***<vfsmml> Tag***
The root VFSMML tag.

The <vfsmml> tag includes a VFSM or a system of VFSMs. Each <vfsmml> tag contains following sub tags: <Name>, <Description> and <VFSM>. This is the top level VFSMML tag. The <vfsmml> tag contains two mandatory attributes:

"project" – can be "true" or "false". The default value of this attribute is "false".

"source" – can be "default" or any other string. The default value is "default". This attribute is used to announce predefined object types used later in the VFSM specification. Appendix A on page 41 lists all predefined types in a "default" system.

### 3.1.3 Element Attributes

In following all defined attributes are listed alphabetically:

***always***
The <State> tag contains the attribute "always" which is usually set to false and means a normal state of an FSM. One of the states of a FSM can contain always="true". This defines not a state of the affected FSM, but is a definition of input actions valid for each state of this FSM (i.e. input actions *always* set). Such an "always-state" does not have a name or entry/exit action, nor it is possible to define a transition to or from this state.

***project***
The <vfsmml> tag contains the attribute "project", which allows to distinguish between complete independent project specifications (value "true") and single VFSM specifications (value "false").

***source***
The <vfsmml> tag contains the attribute "source", which specifies the source of the predefined types. The default value is "default". Appendix A describes all predefined types in a default system. There are any values allowed, however the target system has to know the VFSM definition of the predefined VFSM used, to be able to extract the VFSMML data.

***type***
The <VFSM> tag contains the attribute "type", which allows to distinguish between new VFSM definitions (value "vfsm"), known predefined VFSM definitions (value "predefined") and simple unit definitions (value "unit").

### 3.1.4 Error Message

There is a need for one error message in case the value of the source attribute is unknown. Then the VFSMML message can not be correctly interpreted. The receiver of a VFSMML definition containing an undefined 'source' attribute shall return an empty <vfsmml> tag where the value of the source attribute is set to "unknown":

```
<vfsmml source="unknown"></vfsmml>
```

## 3.2 Linked Information

Some VFSMML tags define values, which are linked together. Following Table 11 gives an overview of tags which uses names previously defined:

| Tag | Linked to | Description |
|---|---|---|
| <StateName> | <State> → <Name> | Specifies a name of a state previously defined |
| <Action>, <EntryAction>, <ExitAction>, <InputAction> | <Output> → <Name> | Specifies a name of an output previously defined |
| <Condition> | <Input> → <Name> | Specifies a name of an input previously defined. Also a set of input names all connected with an OR or AND is possible. See <Condition> tag description on page 30 for more details |

**Table 11: Linked Tags**

There is also some logical information linked together:

The <VFSM>-<Object>-<Name> is one (of many possible) incarnations of the <VFSM>-<Type>. In its list of properties, first of all, any <IOid>-<Name> has to be defined, i.e. the <Object>-<Name> of the VFSM on which this specific <IOid>-<Name> is based. The following example shall explain this dependency:

```
<VFSM>
      <Type>MyVFSM</Type>
      <Object>
            <Name>NewVFSM1</Name>
            <Property>
                  <Name>timer</Name>
                  <Value>timer1</Value>
            </Property>
            <Property>
                  <Name>slave</Name>
                  <Value>SlaveVFSM1</Value>
            </Property>
            <Property>
                  <Name>switch</Name>
                  <Value>switch3</Value>
            </Property>
      </Object>
      <Object>
            <Name>NewVFSM2</Name>
            <Property>
                  <Name>timer</Name>
                  <Value>timer2</Value>
            </Property>
            <Property>
                  <Name>slave</Name>
                  <Value>SlaveVFSM1</Value>
            </Property>
            <Property>
                  <Name>switch</Name>
                  <Value>switch2</Value>
            </Property>
      </Object>
      <IOid>
            <Name>timer</Name>
            ...
      </IOid>
      <IOid>
            <Name>slave</Name>
            ...
      </IOid>
```

This is the first incarnation of the VFSM type ‘MyVFSM’

This is the second incarnation of the VFSM type ‘MyVFSM’

These properties define all objects on which any of the IOid name is created.

Each of the IOid names must appear in each incarnation of this VFSM, to specify the object on which it is based

```
        <IOid>
              <Name>switch</Name>
              ...
        </IOid>
  </VFSM>
```

In other words, each VFSM has its type (VFSM – Type). Based on this type any number of incarnations is possible (VFSM – Object – Name). Each incarnation uses own incarnations of other VFSM, e.g. a “switch” is a typical VFSM used by other VFSM. The description of how to use the foreign VFSM is given in the proper IOid section (VFSM – IOid – Name).

# Appendix A Predefined default VFSM

If no special “source” attribute is given, the default

```
source=”default”
```

is used. The Table 12 below gives an overview of all predefined VFSM, their definitions are presented in the following sub-sections.

| Type | Input Values | Output Values |
|---|---|---|
| AL | - | Coming, Going, Staying |
| CMD | Any positive number >0 | Clear (=0), any positive number > 0 |
| CNT | RUN, STOP, RESET, OVER, OVERSTOP | Start, Stop, Reset, ResetStart, Inc, Dec |
| DAT | UNDEF, DEF, CHANGED, INIT | - |
| DI | HIGH, LOW | - |
| DO | - | High, Low |
| ECNT | RUN, STOP, RESET, OVER, OVERSTOP | Start, Stop, Reset, ResetStart, Inc, Dec |
| NI | UNDEF, DEF, CHANGED, INIT | - |
| NO | - | Off, On, Set |
| OFUN | Any positive number | Any positive number |
| PAR | UNDEF, DEF, CHANGED, INIT | - |
| STR | ON, OFF, SET, MATCH, NOMATCH | Off, Init, Match, Nomatch, Def |
| SWIP | LOW, IN, HIGH, OFF | on, off |
| TAB | - | Any positive number |
| TI | RUN, STOP, RESET, OVER, OVERSTOP | Start, Stop, Reset, ResetStart |
| UDC | UNDEF, DEF, CHANGED, INIT | Clear, Up, Down |
| XDA | Any positive number | Any positive number |

**Table 12: Predefined VFSM and their value ranges (source=”default”)**

Each predefined VFSM contains one or more *properties*. Those are constant parameters which can only be changed during VFSM setup, but not while run-time.

Some of the predefined objects are objects defined to exchange control information with the real world or with other state machines. In particular those objects are: DI, DO, NI, NO, TAB, CMD and XDA. Those objects are defined using the master slave concept. The hierarchy begins with the VFSM control system on top and ends with the real controlled system on the bottom as shown in the diagrams below.

In Figure 9 the objects DI, DO, NI, NO and TAB are slaves from the VFSM control system and masters from the real controlled system point of view. Each master sends commands to its slave. Each slave shows his current state to its master. For instance the DI VFSM is controlled by the states of the real existing device connected to it (e.g. a switch is on, a flag is present etc.) and presents its states to the higher layers of the VFSM control system. The DO VFSM is controlled by the commands from the higher layer of the VFSM control system and sends own commands to the real existing device connected to it (e.g. to a switch).

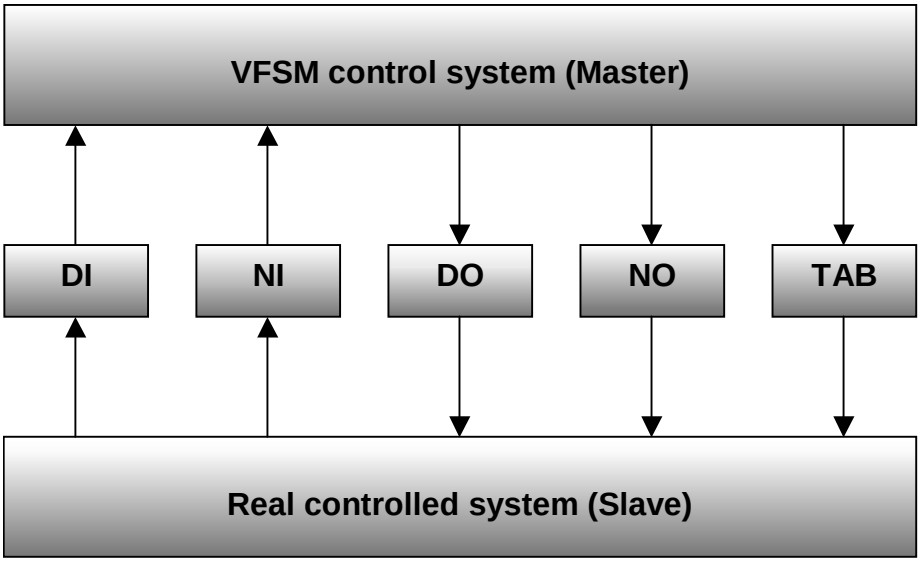



**Figure 9. Objects defined to exchange control information with the real world**

Figure 10 shows the principle of the CMD object used to exchange data between state machines.

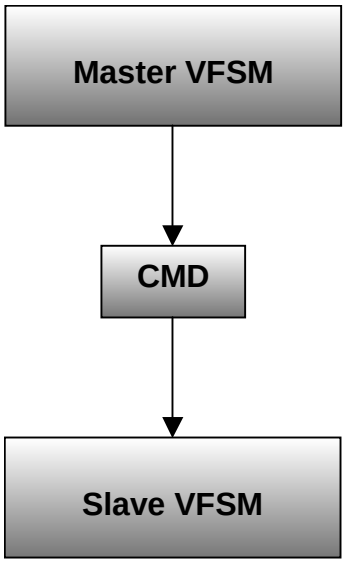



**Figure 10. CMD object defined to exchange control information between state machines**

## A.01 Alarm (AL)

The AL VFSM controls an alarm queue. The following state machine diagram describes the behavior of the AL VFSM:

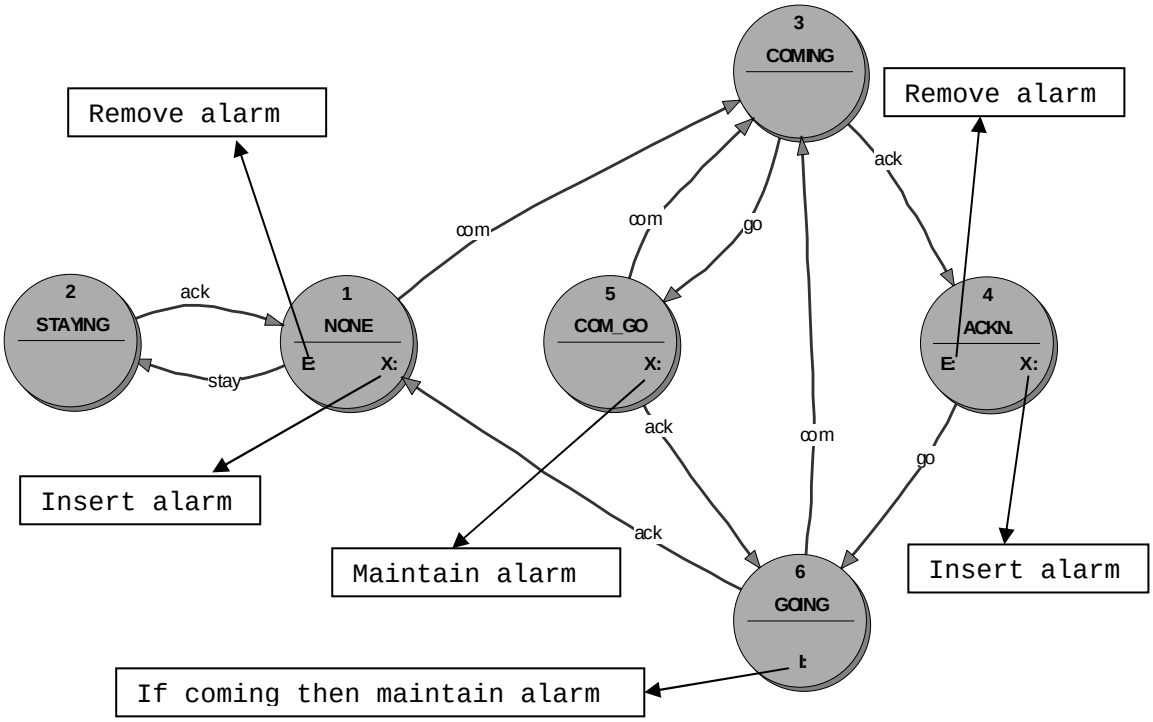



**Figure 11: AL VFSM**

The actions Insert, Remove and Maintain alarm are explained in the table below. E: means an entry action, X: means an exit action, I: means an input action.

| Action | Description |
|---|---|
| Insert alarm | Add current alarm to the top of the alarm queue |
| Remove alarm | Remove current alarm from the alarm queue |
| Maintain alarm | Move the current alarm to the top of the alarm queue |

**Table 13: AL Output Names**

The input names used for state transitions:

| Input Name | Description |
|---|---|
| Coming | Erroneous situation has occurred (incoming command) |
| Going | Erroneous situation has gone (incoming command) |
| Staying | Erroneous situation has occurred. The situation can not be solved (incoming command) |
| Ack | Situation acknowledged (incoming command) |

**Table 14: Al Input Names**

The AL VFSM has following properties:

***Category***

Defines the alarm severity. Can be any number ≥ 0..65535. Some alarms depend on the current state and input name. The table below specifies the meaning of special categories:

| Category | Input Name | |
|---|---|---|
| | Coming, Staying | Going, Ack, None |
| 1 | Write Error to EventLog | Write Info to EventLog |
| 2 | Write Warning to EventLog | Write Info to EventLog |
| 4 | Write Info to EventLog | Write Info to EventLog |
| other | Do not change EventLog (user defined) | |

**Table 15: AL Alarm Severity**

***Text***

Defines the alarm text. To support different languages and references to other data objects two special identifiers are defined:

%<ObjectName>[1] - this string in the alarm text defines a reference to a data object.

For instance

```
Val. is too high: %NiVoltage V
```

will be displayed as

```
Val. is too high: 8.9 V
```

if an ObjectName NiVoltage is defined and its value at the moment of the alarm generation is 8.9.

IDS_<TextId> - this string in the alarm text will be replaced by a string stored in a resource file used with the system. For instance

```
  IDS_VAL_TOO_HIGH: %NiVoltage V
```

will be displayed as

```
Val. is too high: 8.9 V
```

if an object NiVoltage exists as in the previous example and in a resource file one line like following will be found[2]:

```
IDS_VAL_TOO_HIGH “Val. Is too high”
```

## A.02 Command (CMD)

The CMD VFSM is used to exchange control information between state machines as explained at the beginning of this appendix. The following state machine diagram describes the behavior of the CMD VFSM:

[1] this is not a tag, but a place holder for a defined ObjectName

[2] The format of such a resource file is application dependant and not specified in the VFSMML standard.

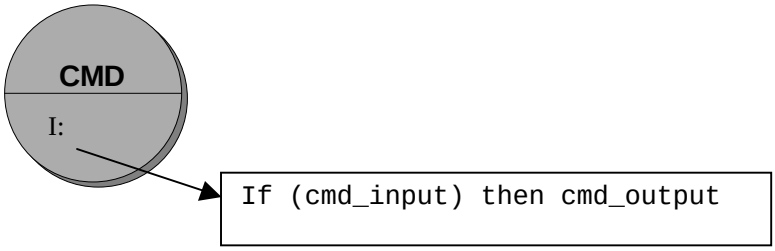



**Figure 12: CMD VFSM**

The output is always same as the input. The system is triggered any time a command was send, also when the same command was repeated. A command is represented by a number. The value 0 is reserved for the ‘clear’ action and must not be used for other control purposes.

The CMD VFSM is used in communication between state machines in a master-slave model:

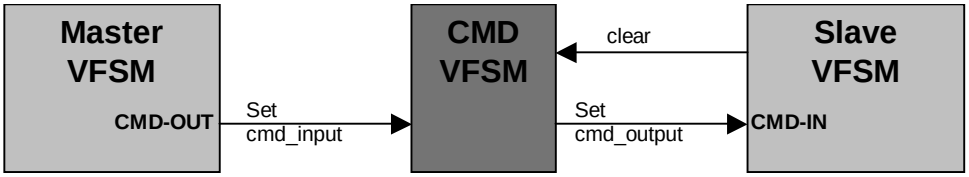



**Figure 13: Usage of CMD VFSM**

From the view of the Master VFSM the CMD VFSM is a CMD-OUT VFSM, i.e. the Master performs the input action ‘cmd’ sending it to CMD VFSM. From the view of the Slave VFSM the CMD VFSM is the CMD-IN VFSM, i.e. the Slave VFSM is the receiver of the output action ‘cmd’ from the CMD VFSM.

The CMD VFSM has following properties:

***Type***
Filename, where all commands (numbers) are mapped to meaningful names, i.e. strings. These strings serve the client to display the data appropriately.

## A.03 Counter (CNT)

The CNT VFSM controls a counting device. The following state machine diagram describes the behavior of the CNT VFSM:

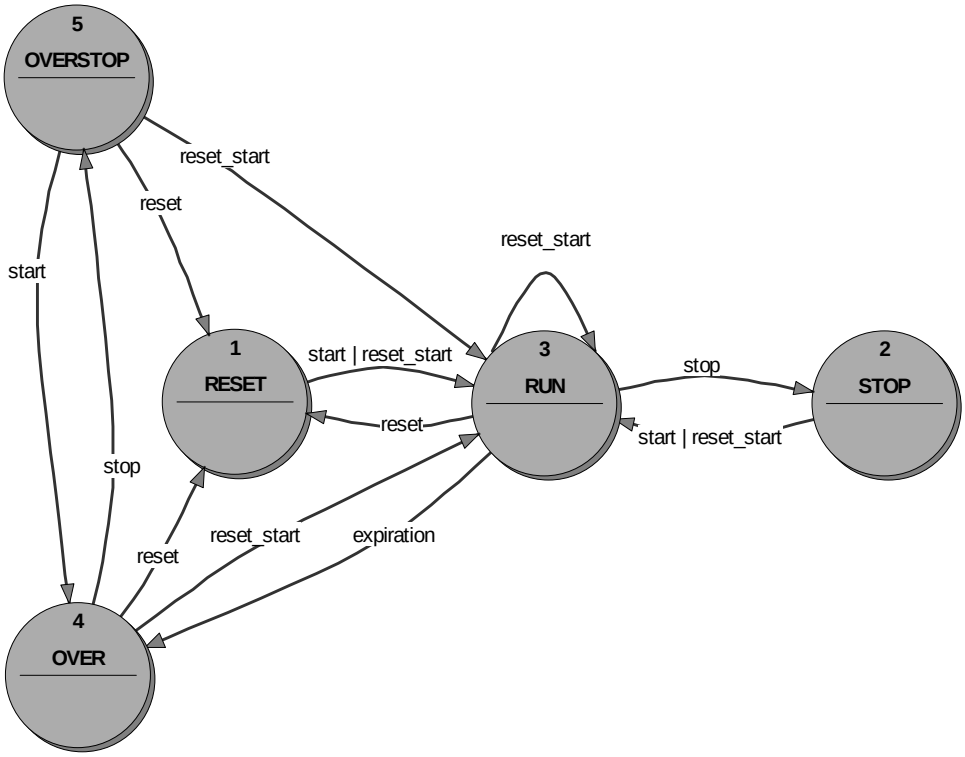



**Figure 14: CNT VFSM**

The input names used for state transitions:

| Input Name | Description |
|---|---|
| start | Incoming command from a master: start counter |
| stop | Incoming command from a master: stop counter |
| reset | Incoming command from a master: reset counter (don't care if counter running or stopped) |
| reset_start | Incoming command from a master: reset and start counter |
| expiration | The counting device reports a new state: limit exceeded |

**Table 16: CNT Input Names**

The CNT VFSM has following properties:

***Const***
Defines the counter limit. Can be a number (e.g. 10) or an NI, DAT or PAR object.

## A.04 Data (DAT)

The DAT VFSM controls a data object which is used to receive and store data. On starting the VFSM is in state OFF. The following state machine diagram describes the behavior of the DAT VFSM:

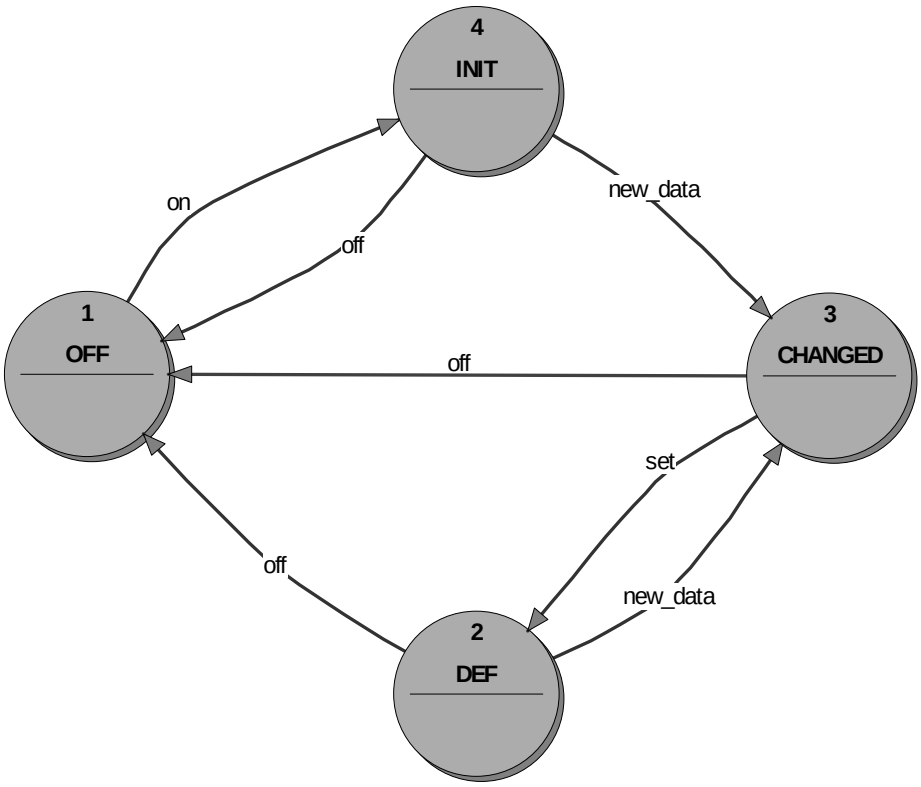



**Figure 15: DAT VFSM**

The input names used for state transitions:

| Input Name | Description |
|---|---|
| on | Incoming command from a master: activate object |
| off | Incoming command from a master: deactivate object |
| new_data | The owned data object reports a new state: new data received |
| set | The owned data object reports a new state: data read (by somebody) i.e. the change has been notified / consumed. |

**Table 17: DAT Input Names**

The DAT VFSM has following properties:

***Format***

Defines the data type of the stored value. The following types are defined:

```
bool: true (1) or false (0)
char: 7 bit character
uchar: 8 bit character (unsigned)
byte: -127..+127 (8 bit signed)
ubyte: 0..255 (8 bit unsigned)
short: -32767..+32767 (16 bit signed)
ushort: 0..65535 (16 bit unsigned)
long: -2147483647..+ 2147483647 (32 bit signed)
ulong: 0..4294967295 (32 bit unsigned)
```

**float: 32 bit floating point ($8.43x10^{-37} \leq |f| \leq 3.37x10^{38}$)**
**double: 64 bit floating point ($4.19x10^{-307} \leq |f| \leq 1.67x10^{308}$)**
**string: a character string of any length**
**pointer: -2147483647..+ 2147483647 (32 bit signed)**
**%E0..%E8: 32 bit floating point, exponent repr. e.g. "3.5e-12"**
**%F0..%F8: 32 bit floating point, fixpoint repr. e.g. "1.234"**
**%G0..%G8: automatic conversion data type**

%En and %Fn are floating point values. The number n (0..8) specifies the number of digits representing the number when sent to a client. It is not the internal representation; this is always the full range. The same is valid for %Gn, however the data type is found automatically.

***Unit***
Defines the unit of the type (e.g. V, mA, Bar, etc.). This string is free selectable and serves the client to display the data appropriately.

## A.05 Digital Input (DI)

The DI VFSM is used to receive states of a real controlled device. The following state machine diagram describes the behavior of the DI VFSM:

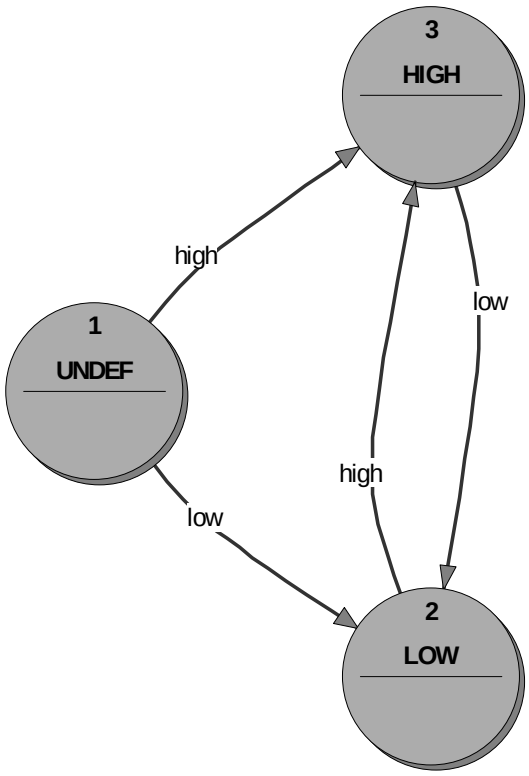



**Figure 16: DI VFSM**

When starting, the VFSM is in state UNDEF, i.e. the digital value (HIGH or LOW) is not defined / known yet. From the control system point of view the DI VFSM is a

slave and shows its status by its states (see also the Figure 9. Objects defined to exchange control information with the real world and its description).

The input names used for state transitions are states of the real controlled device:

| Input Name | Description |
|---|---|
| high | The real controlled device is in state high/on/true (etc.) |
| low | The real controlled device is in state low/off/false (etc.) |

**Table 18: DI Input Names**

The DI VFSM has following properties:

***Invert***
Optional, if not present the default value is FALSE. It can be TRUE if the value shall be inverted for further evaluation. For instance, if a DI is set to LOW, it was received as HIGH.

## A.06 Digital output (DO)

The DO VFSM is used to send commands to a real controlled device. The following state machine diagram describes the behavior of the DO VFSM:

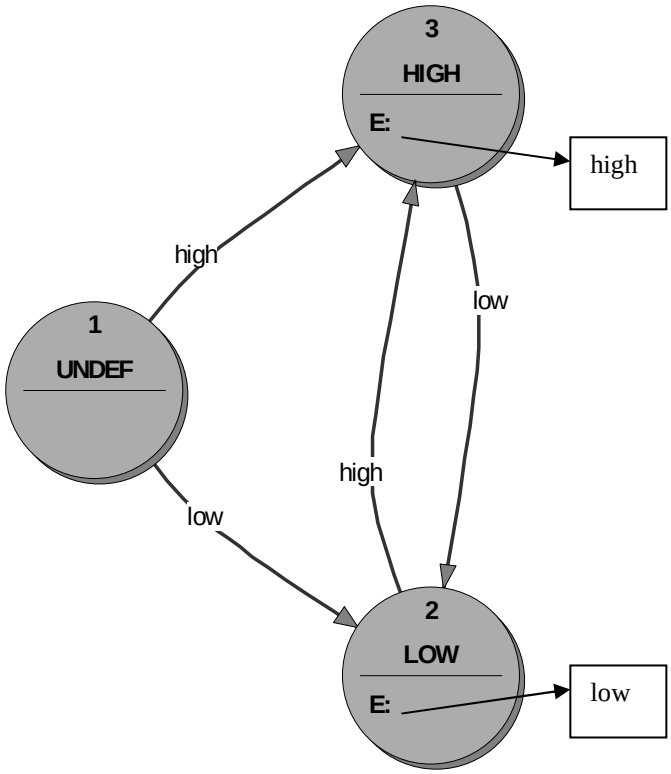



**Figure 17: DO VFSM**

When starting, the VFSM is in state UNDEF, i.e. the digital value (HIGH or LOW) is not defined yet. From the control system point of view the DO VFSM is a slave and receives commands (low or high). From the real controlled device point of view DO is a master and sends commands (low or high) to it.

The input names are commands from the control system (master) and are used for the state transitions as follow:

| Input Name | Description |
|---|---|
| High | Incoming command from a master: high |
| Low | Incoming command from a master: low |

**Table 19: DO Input Names**

The output names are commands to the real controlled system (slave) and are used as entry actions in following states:

| State Name | Description |
|---|---|
| HIGH | Entry action: send command ‘high’ to the real controlled device |
| LOW | Entry action: send command ‘low’ to the real controlled device |

**Table 20: DO Output Names**

The DO VFSM has following properties:

***Invert***
Optional, if not present the default value is FALSE. It can be TRUE if the value shall be inverted for further evaluation. For instance, if a DO is set to LOW, it will be sent out as HIGH.

## A.07 Event counter (ECNT)

The ECNT is a VFSM derived from the CNT VFSM. Its behavior is the same as of the CNT VFSM, however it is used to count any kind of events inside the VFSM control system and therefore a few additional properties are defined:

***Const***
Defines the counter limit. Can be a number (e.g. 10) or another NI, DAT or PAR object.

***Input***
Defines the object used as base for the counter, i.e. VFSM which shall be observed by the counter.

***UpValue***
Defines the event on the observed object, which increases the counter.

## A.08 Numeric input (NI)

The NI is a VFSM derived from the DAT VFSM. Its behavior is the same as of the DAT VFSM. Its purpose is to communicate with a real existing device as its master (see also Figure 9 on page 42 and its description). There are following properties defined:

***Format***
Defines the data type of the stored value. All the same data types as for the Data object besides string (see A.04) are possible.

***Unit***
Defines the unit of the type (e.g. V, mA, Bar, etc.). This string is freely selectable and serves the client to display the data appropriately.

***ScaleMode***
Defines the scale mode of the type:

**lin:** $y = ax + b$
**exp:** $y = e^{ax+b}$
**log:** $y = \log(ax + b)$

***ScaleFactor***
Defines the scale factor for the scale mode (the value of *a*).

***Offset***
Defines the offset for the scale mode (the value of *b*).

***Threshold***
Defines the threshold for the NI value.

## A.09 Numeric output (NO)

The NO VFSM is used to provide data to a real controlled device. On starting, the NO VFSM is in state OFF. The following state machine diagram describes its behavior:

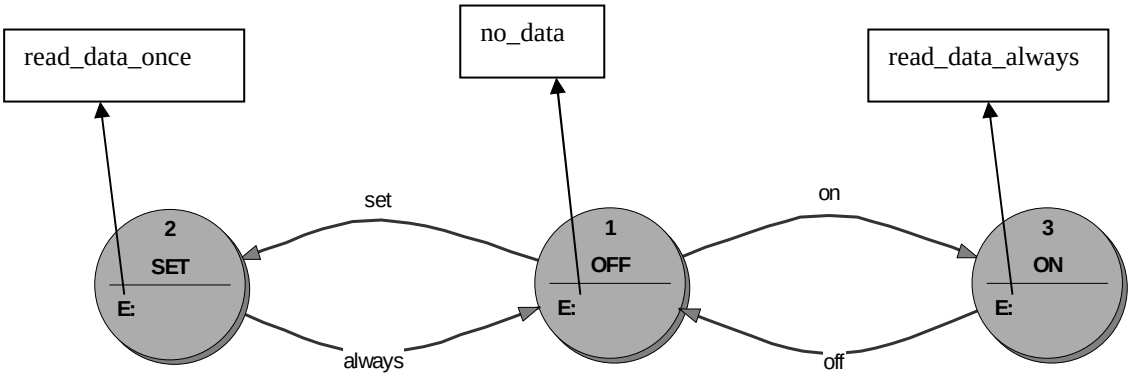



**Figure 18: NO VFSM**

The purpose of the NO VFSM is to send internally stored data to the real controlled device as its master (see also Figure 9. Objects defined to exchange control information with the real world on page 42 and its description).

The input names are commands from the control system (master) and are used for the state transitions as follow:

| Input Name | Description |
|---|---|
| set | Incoming command from a master: activate the object once |
| on | Incoming command from a master: activate the object continuously |
| off | Incoming command from a master: deactivate the object |

**Table 21: NO Input Names**

The output names are commands to the real controlled system (slave) and are used as entry actions as follows:

| State Name | Description |
|---|---|
| SET | Entry action: send command ‘read_data_once’ to the real controlled device |
| ON | Entry action: send command ‘read_data_always’ to the real controlled device |
| OFF | Entry action: send command ‘no_data’ to the real controlled device |

**Table 22: NO Output Names**

The following properties are defined:

***Format***
Defines the data type of the stored value. All the same data types as for the Data object besides string (see A.04) are possible.

***Unit***
Defines the unit of the type (e.g. V, mA, Bar, etc.). This string is free selectable and just serves the client to display the data appropriately.

***ScaleMode***
Defines the scale mode of the type:

**`lin:`** $y = ax + b$

**`exp:`** $y = e^{ax+b}$

**`log:`** $y = \log(ax + b)$

***ScaleFactor***
Defines the scale factor for the scale mode (the value of *a*).

***Offset***
Defines the offset for the scale mode (the value of *b*).

***OutData***
Defines the object (TAB, DAT, PAR, NI or UDC) to be used for output.

## A.10 Output function (OFUN)

The OFUN VFSM is used to enable the control system to evaluate results of certain calculations which are not I/O related. The OFUN VFSM has only one state and its output is a function of the input. The following state machine diagram describes its behavior. The function *f* is user defined.

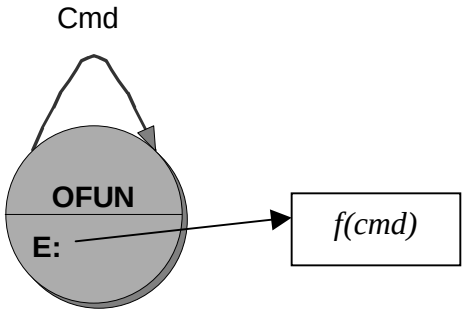



**Figure 19: OFUN VFSM**

The following properties are defined:

***FunctionName***
Defines the name of the function *f* which implements the OFUN object (e.g. a C++ function).

***UnitName***
Defines the unit (interface) or VFSM to be accessed by the coded function.

## A.11 Parameter (PAR)

The PAR is a VFSM derived from the DAT VFSM. Its behaviour is the same as of the DAT VFSM. Its complexity is very similar to the NI object. However the data stored in a PAR VFSM can be saved to be available after a system restart. The following properties are defined:

***Category***
Defines how to save the parameter value. The following categories are possible (PP=Process Parameters, EP=Equipment Parameters):

```
PP: store the parameter temporary (for current sesion)
PP_Coded: exactly the same as PP, however the parameter value is a
   result of a calculation. There is no need to distinguish between
   PP and PP_Coded, besides that the user knows the source of the
   data.
EP: store the parameter permanently for the current user
EP_LM_USER: store the parameter permanently for all users
EP_LM_ADMIN: store the parameter permanently for all users, can be
   change only by a system administrator
```

***Format***
Defines the data type of the stored value. All the same data types as for the Data object (see A.04) are possible.

***Unit***
Defines the unit of the type (e.g. V, mA, Bar, etc.). This string is free selectable and serves the client to display the data appropriately.

***LimitLow***
Defines the lowest accepted value.

***LimitHigh***
Defines the highest accepted value.

***InitValue***
Defines the initial value of the parameter.

## A.12 String (STR)

The STR VFSM is used to control a data object used to evaluate strings. In detail, it compares the received string with a regular expression (RE). The result is a “match”, “no-match” or “error”. The regular expression itself can be a DAT, PAR or a hard coded string. The regular expression allows all special characters as known in UNIX tools like sed, awk. This means that also multiple matches are possible, i.e. the compare result “match” can deliver more then one resulting string. The resulting (sub)string(s) can be stored in other objects such as STR, DAT, PAR or NI. Dependant on the data type of the destination object, the resulting (sub-) string will be converted. In case the conversion is not possible the destination object will be not changed.

The table below describes all allowed regular expressions:

| RE | Meaning | Example |
|---|---|---|
| . | Matches one arbitrary character | a.c matches ‘abc’ but not ‘abbc’ |
| ^ | Matches the beginning of a string | ^ab matches ‘abcd’ but not ‘cdab’ |
| $ | Matches the end of a string | ab$ matches ‘cdab’ but not ‘abcd’ |
| \n | n=1..9, matches the same string of characters as was matched by a sub expression enclosed between () preceding the \n. n specifies the n-th sub expression | (ab(cd)ef)A\2 matches ‘abcdefAcd’ |
| ( ) | sub expression | (\d)A(\d) matches 1A2, 0A4 ... |
| [ ] | Defines a set of characters to be matched | [a-z] matches ‘s’, ‘w’… but not ‘S’, ‘W’… |
| [^ ] | Defines all characters except the characters in the set | [^1-9] matches ‘s’, ‘W’ … but not ‘1’, ‘2’… |
| ( \| ) | Matches one of the alternatives | (ab\|cd) matches ‘ab’ and ‘cd’ |
| RE+ | Matches one or more times the RE | [^1-9]+ matches ‘StateWORKS’ but not ‘Obj5’ |
| RE? | Matches one or zero times the RE | abc? matches ‘ab’ and ‘abc’ |
| RE* | Matches zero or more times the RE | ab* matches ‘a’, ‘ab’, ‘abb’ … |
| RE{n} | Matches exactly n times the RE | ab{2} matches ‘abb’ only |
| RE{n,} | Matches at least n times the RE | ab{2,} matches ‘abb’, ‘abbb’ but not ‘ab’ |
| RE{n,m} | Matches any number of occurrences between n and m inclusive | ab{1,2} matches ‘ab’ and ‘abb’ only |

**Table 23: STR VFSM - Allowed Regular Expressions**

The following state machine diagram describes the behavior of the STR VFSM:

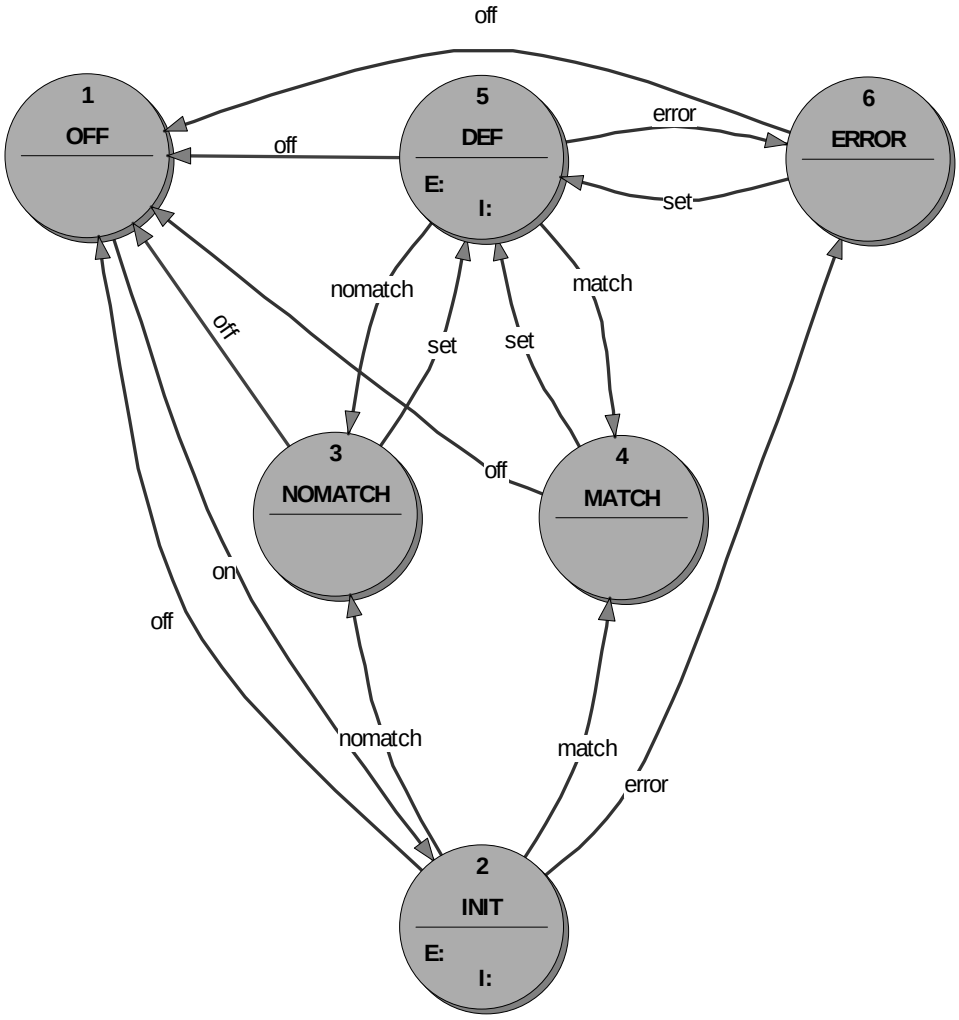



**Figure 20: STR VFSM**

The Input action in states INIT and DEF is always the same: analyze string if a new string has arrived. The Entry action is always the same: reset the analysis function.

The input names used for state transitions:

| Input Name | Description |
|---|---|
| On | Incoming command from a master: activate object |
| Off | Incoming command from a master: deactivate object |
| Set | Incoming command from a master: results evaluated |
| Match | The owned data object reports a new state: sub-string(s) found |
| Nomatch | The owned data object reports a new state: no sub-string found |
| Error | The owned data object reports a new state: error evaluating the RE |

**Table 24: STR Input Names**

The STR VFSM has following properties:

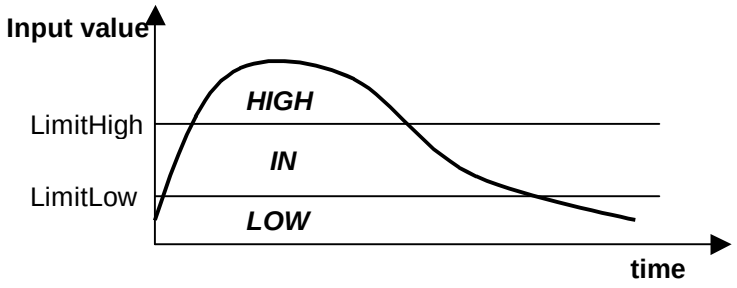
***Input***
Defines the source string to be analyzed. Can be DAT or PAR.

***RegularExpression***
Defines the regular expression to be used. Can be a constant string, DAT or PAR.

***Substring***
Defines the destination object: DAT, PAR, STR or NI. This property is a list, i.e. there can be many destination objects, in case there are many matches possible. The number of parents in the RE gives the number of possible sub strings.

## A.13 Switch point (SWIP)

The SWIP VFSM is used to control a switch point function which is used to evaluate changes of data of other VFSM. The following graph shows the usage of the SWIP VFSM:




The following state machine diagram describes the behavior of the SWIP VFSM:

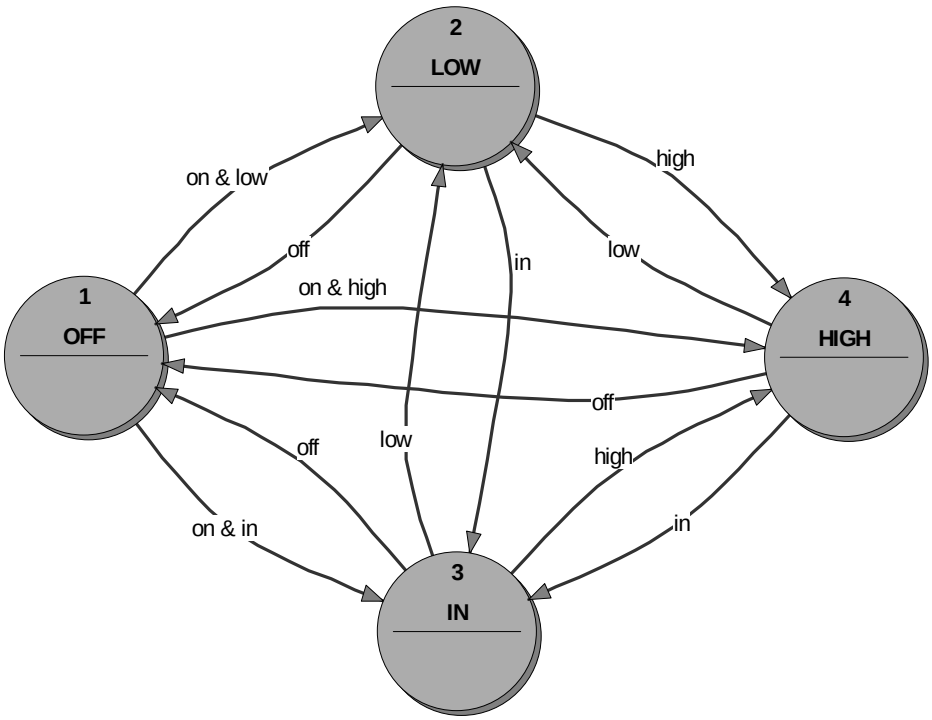



**Figure 21: SWIP VFSM**

The input names used for state transitions:

| Input Name | Description |
|---|---|
| on | Incoming command from a master: activate object |
| off | Incoming command from a master: deactivate object |
| low | The controlled switch point function reports a new state: object value below min. limit |
| high | The controlled switch point function reports a new state: object value above max. limit |
| in | The controlled switch point function reports a new state: object value between min and max (min ≤ value ≤ max). |

**Table 25: SWIP Input Names**

The SWIP VFSM has following properties:

***Input***
Defines the object used as a base for SWIP, i.e. object observed by the switch point function. Can be NI, PAR, DAT or UDC VFSM. Note that more than one SWIP might monitor the same object, so as to obtain more detailed information.

***LimitLow***
Defines the value below which the switch point function reports state LOW.

***LimitHigh***
Defines the value above which the switch point function reports state HIGH.

## A.14 Table (TAB)

The TAB VFSM is used to provide data of various VFSM objects to a real controlled device. The following state machine diagram describes the behavior of the TAB VFSM:

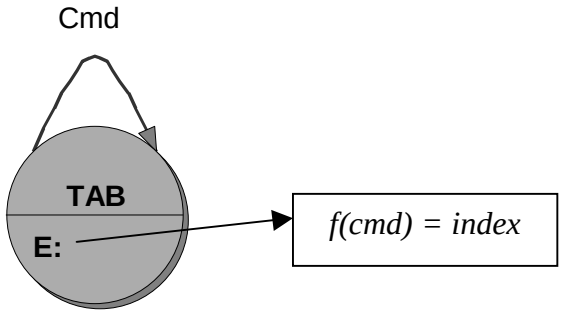



**Figure 22: TAB VFSM**

The entry action specifies the index of a certain VFSM of type PAR, DAT or NO, i.e. TAB works as a multiplexer that maps several VFSMs to one output:

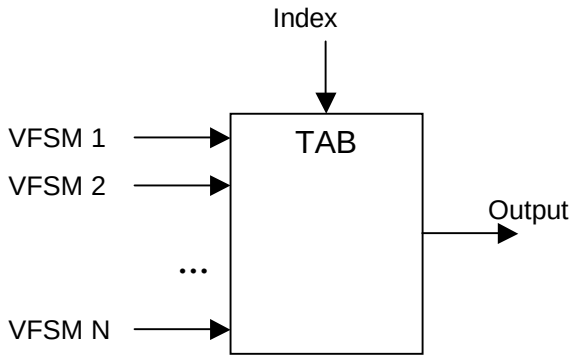



**Figure 23: TAB as Multiplexer**

The following properties are defined:

***Input***
Defines the VFSMs used as base for TAB. This property is a list, i.e. as a rule there are many objects used by TAB.

## A.15 Timer (TI)

The TI VFSM is used to control a timer. The timer is a special counter. So the behavior of the TI VFSM is the same as of the CNT VFSM, however there are more properties required:

***Const***
Defines the counter limit. Can be a number (e.g. 10) or an NI, DAT or PAR object.

***Clock***
Defines the clock base, i.e. the time period after which the counter value gets increased automatically. Can be

**1ms = $10^{-3}$ sec**
**10ms = $10^{-2}$ sec**
**100ms = $10^{-1}$ sec**
**1s = 1 sec**
**1min = 60 sec**
**1h = 3600sec**

## A.16 Up-Down counter (UDC)

The UDC is a counter derived from the DAT VFSM (not CNT). The purpose of this VFSM is to be able to count in both directions. The data type (format) is long, the counter has no limit (i.e. a SWIP VFSM is required to evaluate its value). Its behavior is the same as of the DAT VFSM, however in the states INIT, CHANGED and DEF following incoming commands are possible:

| Input | Description |
|---|---|
| clear | Execute the input action “clear counter” |
| up | Execute the input action “increase counter value” |
| down | Execute the input action “decrease counter value” |

**Table 26: UDC Input Names**

Each of these commands causes the data object owned by the UDC VFSM to go to the state ‘new_data’

The following properties are defined:

***Unit***
Defines the unit of the type (e.g. V, mA, Bar, etc.). This string is free selectable and serves the client to display the data appropriately.

***UpInput***
Defines the object, which is the source for triggering the counter increment operation.

***UpValue***
Defines the value of the object defined in <UpInput> tag, which increases the counter.

***DownInput***
Defines the object, which is the source for triggering the counter decrement operation.

***DownValue***
Defines the value of the object defined in <DownInput> tag, which decreases the counter.

***ClearInput***
Defines the object, which is the source for triggering the counter clear operation.

***ClearValue***
Defines the value of the object defined in <ClearInput> tag, which clears the counter.

## A.17 Any data (XDA)

The XDA VFSM is used as an OFUN VFSM support object, by pointing to a memory segment used by the OFUN VFSM. The XDA VFSM looks the same as the

TAB VFSM. However its output is exactly the same as its input, i.e. *f(cmd) = cmd*. The following property is defined:

***Size***
Defines the size of memory block in bytes.

# Appendix B Units

The unit concept is introduced to allow the access to any VFSM objects. So a unit defines the I/O interface to a VFSM. It does not have any other functionality. The definition of a unit is very similar to a VFSM, however a unit does not have the behavior section.

There are two predefined optional unit properties:

***Address***
Physical address of a unit.

***Port***
The communication port of a unit.

# Appendix C Parsing VFSMML

## A.01 DOCTYPE Declaration for VFSMML

VFSMML documents should be validated using the XML DTD for VFSMML, which is shown below in section A.03. Documents using this DTD should contain a doctype declaration of the form:

```
<!DOCTYPE vfsmml
   PUBLIC “http://www.stateworks.com/dtd/vfsmml1.0.dtd”
>
```

The URI might be changed to that of a local copy of the DTD if required.

## A.02 Use of VFSMML without a DTD

If a VFSMML fragment is parsed without a DTD, i.e. as a well-formated XML, it is the responsibility of the processing application to treat the white space characters occurring outside of token elements as not significant.

## A.03 The VFSMML DTD

The code below is the complete Document Type Definition of VFSMML:

```
<!ELEMENT vfsmml (Name?, Description?, VFSM*)>
<!ATTLIST vfsmml
      project (true | false) "false"
      source (default | CDATA) "default"
>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT VFSM (Type, Description?, Prefix?, Object*, IOid*,
State*)>
<!ATTLIST VFSM
      type (vfsm | predefined | unit) "vfsm"
>
<!ELEMENT Type (#PCDATA)>
<!ELEMENT Object (Name, Description?, Property*)>
<!ELEMENT Property (Name, Value)>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT Prefix (#PCDATA)>
<!ELEMENT IOid (Name, Type?, Description?, Input*, Output*)>
<!ELEMENT Input (Init?, Name, Value)>
<!ELEMENT Init (#PCDATA)>
<!ELEMENT Output (Name, Value)>
<!ELEMENT State (Description?, Name?, EntryAction*,
ExitAction*, InputAction*, Transition*)>
<!ATTLIST State
      always (true | false) "false"
>
<!ELEMENT EntryAction (#PCDATA)>
<!ELEMENT ExitAction (#PCDATA)>
<!ELEMENT InputAction (Condition, Action*)>
<!ELEMENT Condition (ci | apply)>
<!ELEMENT ci (#PCDATA)>
<!ELEMENT apply ((and | or), (ci+ | apply*)+)>
<!ELEMENT and EMPTY>
<!ELEMENT or EMPTY>
<!ELEMENT Action (#PCDATA)>
<!ELEMENT Transition (Condition, StateName, Action*)>
<!ELEMENT StateName (#PCDATA)>
```

# Appendix D References

[1] www.stateworks.com

[2] www.w3c.org