# Complement control values in the VFSM concept[1]

## *Introduction*

In the VFSM concept [1] [2] we use control values for expressing input values. A control value is defined as an input data property which may be relevant for a control. As nearly all input types have more than two control values we have to invent a specific algebra for formulating logical expressions for state transitions and input actions. This algebra uses boolean operators AND and OR but forbids the operator NOT as there is no unique value for a negation of a control value in a multivalued control environment. This limitation allows us an implementation of the execution environment based on set theory.

## *Problem*

Many years of experience have shown us that the missing NOT operator is not a serious limitation. Nevertheless, there are situations which we would like to solve in a better way. Let's take an example of a SWIP object which supervises a NI value. The SWIP object has the following control values: OFF, UNKNOWN, LOW, IN and HIGH. If we want to express a condition that means a negation of the other values we use a workaround. For instance let's assume that the value IN means TemperatureOk. Then a TemperatureNotOk (that means explicitly *NOT IN*) condition will be then expressed as *LOW OR HIGH OR OFF OR UNKNOWN* (in cases when it does not matter we could use a simplified expression: *LOW OR HIGH*). This solution works well for all objects which have a limited number of control values; in the standard RTDB objects (excluding CMD, XDA and OFUN of course) there are never more than 6.

This issue becomes a more difficult problem in Master state machines which use states of other state machines as inputs. As state machines can have many states applying directly the previous solution becomes not very practicable. Therefore, we have, for a state machine, a possibility of defining a complex expression, so hiding the long OR expression behind an expressive name. Hence, trying for instance to use a condition *NOT State_Error* we define a complex expression as an OR of all other states. In fact, it is the same solution as for the SWIP shown above but making the specification “lighter” and easier to read.

The NOT prohibition is not a very severe limitation but anyway we decided to simplify the specification task by adding negated control values. We call the negated values complement values in line with the set theory.

Arguments in favour of complement values apply for the VFSM concept implemented as in StateWORKS where most of the standard objects used have a fixed and limited set of control values. In coded implementations (see for instance the AT&T approach [3]) the use of complement values is irrelevant as we are free in defining any control value. Thus, if we need a condition TemperatureNotOk like in the example above we just define that condition in the coded version. In StateWORKS we have to use existing values for defining the condition *TemperatureNotOk* and the complement value is then useful. The decision whether to define *TemperatureNotOk* as a complement of control value IN or to use an expression *TemperatureTooLow OR TemperatureTooHigh OR TemperatureUnknown OR Temperarue SupervisionOff* (where TemperatureTooLow is defined on a control value LOW, TemperatureTooHigh on a control value HIGH, and so on) is maybe a matter of taste.

1 This version replaces the first one from October 2007

Having this in mind we conclude that the use of a complement control value in StateWORKS should be limited to object types with a fixed predefined set of control values. The few types (CMD, XDA, OFUN) that allow the user to define any value needed do not require the complement. In StateWORKS editor we may define conditions of complement of control values for any object type but we should use it carefully.

## *Solution*

In the specification we are going to use two forms of the control value: true and complement. A negated control value - its complement - is denoted by the prefix ~ and means any other value, for that object. For instance, for the above discussed SWIP object the ~IN means any other value: OFF, LOW, HIGH or UNKNOWN. We note in passing that in that case a complement value is always complete, covering all “other” control values. The complement of a state is especially powerful. For the example above, the value ~State_Error means that the Slave state machines is in any state except State_Error.

## *StateWORKS run-time implementation*

This concept makes the implementation of the VFSM Executor more complex. The implementation details are transparent for the user and therefore maybe not very interesting. Therefore, we limit our explanation to some basic observations.

An RTDB object inserts its input name into the virtual input. Without the complements the inputs of all RTDB objects are mutually exclusive. The use of complements has changed this rule: at a given moment more than one input of a given object may exist in the virtual input. For instance, let's take again the SWIP object. It has 5 true control values: OFF, LOW, IN, HIGH and UNKNOWN. Let's assume that we use 3 of them: LOW, IN and ~IN defining on them names: Too_Low, Ok and Not_Ok. Depending on the SWIP value the virtual input contains then :

| **SWIP control value** | **virtual input** |
|---|---|
| OFF | {Not_Ok} |
| LOW | {Too_Low, Not_Ok} |
| IN | {Ok} |
| HIGH | {Not_Ok} |
| UNKNOWN | {Not_Ok} |

The content of a virtual input for the true value OFF might be empty ({}); it is a question of a definition.

Note some specific situations in the specification:

- Ok & Not_Ok = false
- Ok | Not_Ok = true

corresponding to the rules of the boolean algebra.

Although the theoretical basis is a relatively simple one an implementation of the VFSM Executor is not that simple. The reason is that a straightforward implementation would require the use of a set with elements like *A* and *~A*. As the RTDB uses sets of integers the implementation has to be a bit more sophisticated but that is another story.

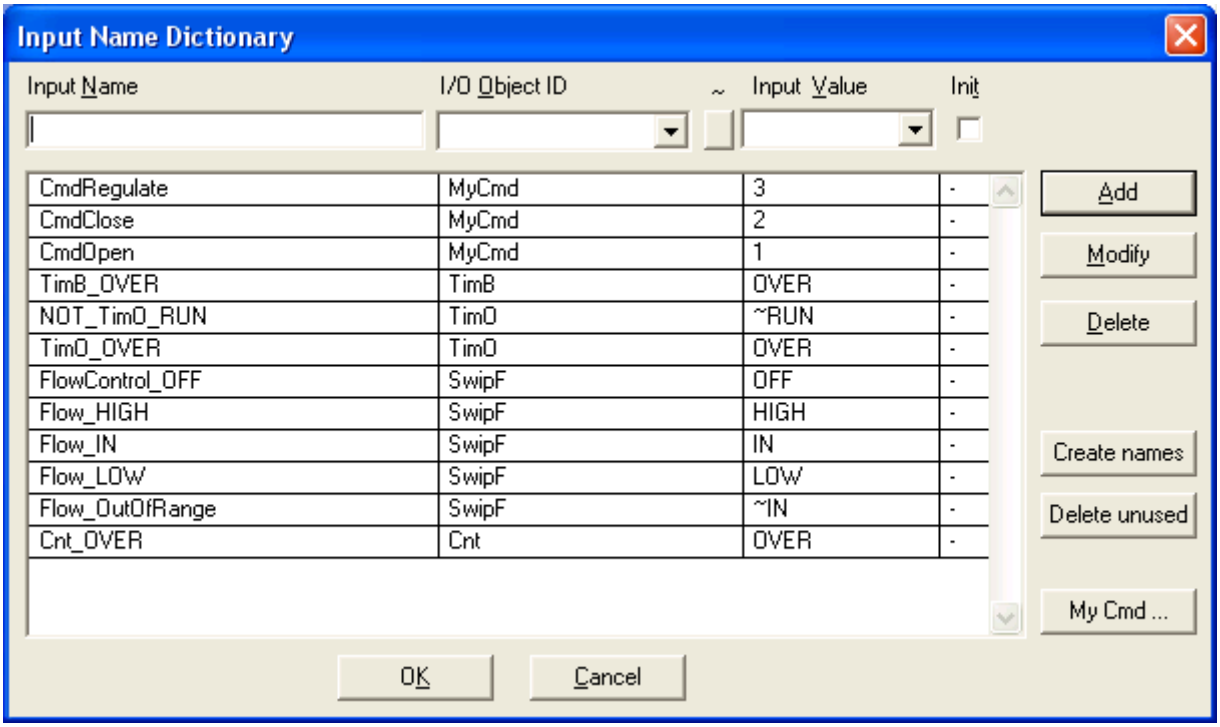


*Figure 1: The Input Name Dictionary with inactive button "~"*

The use of complements within a specification is simple. Opening the Input Name Dictionary dialog window we note one additional button labeled as “~”. This button is used to complement the control values. If the button is not pushed (as in Figure 1) the values used are true (not complemented).

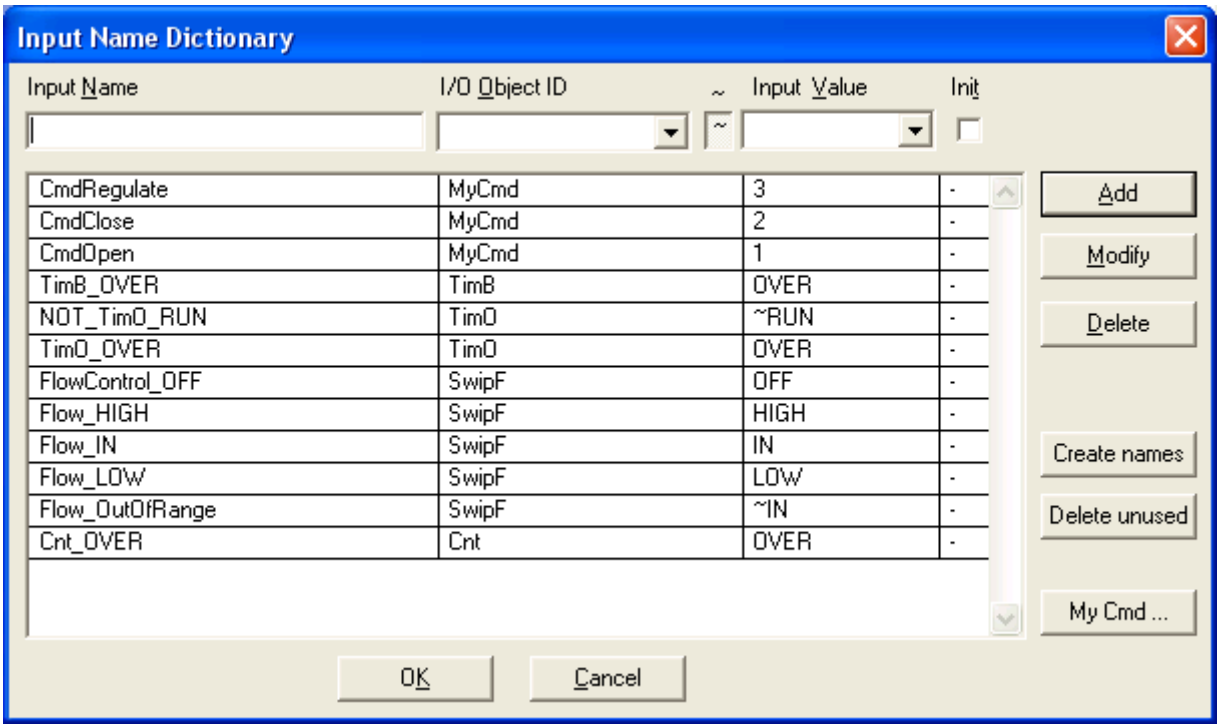


*Figure 2: The Input Name Dictionary with active button "~"*

Pushing this button (as in Figure 2) it presents the caption “~”and the values used are complemented. By default the input names generated on complement values receive the name with a prefix NOT_ (of course we may rename the default value with any string). We may use the button “~” when adding (button *Add*) or modifying (button *Modify*) input names.

The button *Create names* creates names on all true values if the button “~” is inactive and names on all complement values if that button is active. Of course, as in the previous Studio version creation of all names does not cover objects without a defined number of control values, i.e. objects of type: CMD, XDA and OFUN. The discussion in the section “Problem” suggested anyway that for these object types an assignment of complemented control values is not recommended and difficult to justify.

## Naming

The naming governs the understandability of the specification. Therefore, we discuss it very often, stressing the importance of names used. We know also that there is no unique solution: it depends strongly on designers and their preferences. The use of complement of control value intensifies the importance of choice of input names. Let's discuss it using examples in Figure 1 or Figure 2. They show two names defined on complemented values.

The name *NOT_TimO_RUN* covers all situations except RUN that is: RESET, STOP, OVER and OVERSTOP. The default name *NOT_TimO_RUN* “shows” explicitly the meaning. If we rename it for instance to a name *TimO_Stopped* this will be misleading as it would suggest that the timer is halted. Actually, the timer may not run (being in states: RESET, STOP and OVERSTOP) or run (being in the state OVER): this nuance may be important for understanding the meaning. Maybe the name *TimO_NotRunningOrOVER* would be a better choice.

The name *Flow_OutOfRange* covers all situations except IN that is: LOW, HIGH, UNKNOWN and OFF. We have decided to use a simple name as the consequences are trivial. In that application all control values except IN are equivalent and therefore it seems acceptable to “think” that flow is out of range also in situations when the switchpoint is disabled (being in the state OFF) or signals that the input signal is not available (being in the state UNKNOWN). Wouldn't it be nitpicking to invent a name *Flow_IsInStateLOWorHIGHorUNKNOWNorOFF?* In cases when those situations must be clearly differentiated we have to use precise conditions defined on true values and built exact expressions ORing them to get required conditions, for instance *Flow_LOW OR Flow_HIGH*.

## Conclusion

The introduction of the complements is a major extension to the VFSM concept as implemented in StateWORKS. It simplifies the creation of a specification, especially in Master state machines which use Slaves' states as input conditions.

As the complement operation compensates to a large extent the creation of complex expressions for conditions in Master using Slave states we have abandoned that feature.

## References


1. Wagner, F. et al., *Modeling Software with Finite State Machines – A Practical Approach*, Auerbach Publications, 2006.
2. Wagner F.: Technical Note: “The Virtual Environment”, April, 2003.
3. Flora-Holmquist, A.R., Morton, E., O'Grady, M.G., Staskauskas, M.G., “The virtual finite state design and implementation paradigm”, *Bell Labs Technical Journal* (1997): 97-113.