F. Wagner

April 2003
revised April 2009

The Virtual Environment

Boolean algebra

We all know Boolean algebra. We use its rules all the time while programming. For the purpose of this note let's write down some basics:

- a Boolean variable takes values **true** or **false**
- a Boolean expression uses three operators: **AND**, **OR** and **NOT**
- any Boolean expression can be represented by an OR combination of AND expressions
- logical statements can be expressed by Boolean expressions or truth tables.

Example

The Boolean condition:

```
f = (x AND y) OR ((NOT y) AND (NOT y))
```

and its equivalent table representation:

x \ y	false	true
false	true	false
true	false	true

Why use another algebra?

In software, Boolean algebra is used in control statements (if, while, for) to determine the execution path. Boolean algebra is fine while working with 2-value (*true*, *false*) logic variables.Then, we write for instance (x is a Boolean variable):

```
if (x)
```

```
while (NOT x)
```

Control statements with pure Boolean values can be replaced by tables. For instance, the condition *f* in the above example can be replaced by a table. In the code, the table may look for instance like this:

```
bool f[bool][bool] = {{true, false}, {false, true}}
```

But in software we often need to use non-Boolean variables in logic expressions.

If the variables have more than the two possible values of *true* and *false* we have to work out the logic value using, for instance, comparison operators. We would write:

```
if (a>5)

while (a<b)

for (i=0; i<=10; i++)
```

The problem which arises is that control statements with non-Boolean values cannot be replaced by tables.

We need to adapt the logic algebra that have to allow any control statement be replaced by a table.

Definitions

Virtual environment

VFSM concept requires a specific environment which we call a *Virtual environment*.

Variables used in software are of several types. A variable that represents a digital input stores a Boolean value (*true*, *false*). A variable that represents an analog input stores a number (for instance: *float*). A parameter stores a value that could be of any numerical type (*integer*, *float*, *string, ...*). Software uses also more complex structures or objects for which the behaviour or value is used in control statements. For instance, a timer or a counter stores a number which changes when it runs or counts.

Let's analyze the true character of some variables or objects from the control statement viewpoint.

- Digital input: both its values (*true* or *false*) are used in control statements as a

direct Boolean value. For consistency with variables discussed below let's name the two values: DI_ON and DI_OFF (in a specific application we will use more expressive names, reflecting their true meaning, for instance: END_SWITCH_REACHED, END_SWITCH_NOT_REACHED).

- Analog input (a *float* number): only a small fraction of all possible float values are used in control statements, for instance in the form:

  $ai < 5.3$
  $ai > 8.1$
  $5.3 \leq ai \leq 8.1$.

  We may "describe" the used input ranges as follows (in parenthesis are examples of some application specific names):

  AI_SMALLER_THAN_5.3 (TEMPERATURE_TOO_LOW)

  AI_LARGER_THAN_8.1 (TEMPERATURE_TOO_HIGH)
  AI_IN_RANGE_5.3_TO_8.1 (TEMPERATURE_OK)

- Timer: whether it is running or not is less interesting. The true interesting moment occurs when the timer expires after the timeout elapses. Let's call this information TIMER_OVER, the other situations being named TIMER_RUN, TIMER_RESET and TIMER_STOPPED.

We may say that in all cases the names represent the true control values of the above variables. The names have two interesting features. First, they are of the same "type", they are just names. Second, the names are not Boolean values as there are, in general, several names which represent the variable control value.

Some variables may have one additional value, which we may call UNKNOWN. The UNKNOWN value is especially important for signals which are delivered from outside world. For instance, digital or analog inputs are coming from peripheral devices. If the devices or drivers which make the link between the peripheral devices and the control software are inactive or if they malfunction the corresponding control variables have the value UNKNOWN. It means that effectively, there are no true Boolean values in control software, as even the digital inputs have 3 values: DI_ON, DI_OFF and DI_UNKNOWN.

The names of control relevant values define a *Virtual environment*.

Operators

Control statements need Boolean values. Hence, the names must be used to produce Boolean results. To achieve this we want to combine them together using Boolean operators. There is nothing wrong with usage of AND and OR operators with their Boolean meaning. For instance, we may write:

DI_ON OR AI_LARGER_THAN_8.1 AND TIMER_OVER

to express the control situation: digital input is on or analog input is larger than 8.1 and timer is over.

We cannot use the NOT operator, because the result of the Boolean negation makes sense only for true Boolean values. The result of, for instance,

NOT (AI_LARGER_THAN_8.1)

would be ambiguous.

Complement control value[1]

To simplify the logical conditions VFSM uses the concept of complement (control) value. The complement as known in mathematics, especially in set theory defines unique values in comparison in relation to an other set. In our case we use a complement of a name that belongs to the set of names that define all control values of a given input variable; the complement of a name means then "all other names".
For example the set of all timer control values contains: RUN, RESET, STOPPED and OVER. If we want to express all values that define the situations before the timer elapses we may use the complement value of OVER: ~OVER. Per default the complement value gets in StateWORKS Studio the prefix NOT_. The use of a name defined for a complement value leads evidently to simpler conditions than using names defined on all control values as:

NOT_TIMER_OVER = TIMER_RUN OR TIMER_RESET OR TIMER_STOPPED.

Of course you may overwrite the default name by any name that is more suitable for your application.

Processing

The intention of the virtual environment is to replace software control statements with

decision tables.

First of all, we assume that names of all those control values which are currently valid are held in a variable called VI (*Virtual Input*). VI is a set of all currently valid names.

On the other hand, the specification which describes the behavior of the application is represented by tables. The following representation of the logical conditions is required for the tables:

- A logical multiplication (AND operation) is represented by a set of conditions.
- A logical addition (OR operation) is represented by a table (group) of sets.

Thus, if we limit the notation to classical OR_of_AND_expressions (the term "disjunctive form" is used in Automata Theory) the processing of the control statements is done in the following way:

- Test whether a set represented by the first AND condition is a subset of the *VI.*
- Repeat this test for all sets until the answer is TRUE.
- If any AND condition is a subset of the *VI* the condition is fulfilled (*true*), otherwise it is *false*.

Example

The following condition:

END_SWITCH_REACHED OR AI_TEMPERATURE_TOO_HIGH AND TIMER_OVER

specifies some action and will be presented as a table of 2 sets:

{ END_SWITCH_REACHED }

{ AI_TEMPERATURE_TOO_HIGH , TIMER_OVER }

Assume that a current VI contains the following:

{ END_SWITCH_NOT_REACHED, AI_TEMPERATURE_OK. TIMER_OVER }

Processing of VI results in a *false* result and no action is taken.

Summary

The concept of a *Virtual environment,* as expressed by the *virtual input* and the *logic algebra,* is the basis of the VFSM (virtual finite state machine) specification method. We cannot use the NOT operator but it is compensated by the availability of a complement value.

We use sometimes the term Positive Logic Algebra to indicate the differences in use of the logic algebra in the VFSM concept.

---

1. For details see the technical note "Complement control values in the VFSM concept"