F. Wagner September 2006

Event driven state machines

Introduction

In our book [1] we have discussed among other topics the misunderstandings about state machines. One of the topics which has to be developed is the event driven idea while designing state machines. This technical note summarizes the problem using simple examples.

We have introduced two types of state machines: Parser and Controller. The Parser type is a state machine whose inputs are true events, i.e. they are used to trigger an (input) action or to change a state and thereafter they are consumed. All information represented by the event has to be stored in states. These type of state machines are used for instance in telecommunication for protocol definition. The Controller type of a state machine uses input signals which are of different flavours depending on their origin. Some inputs are static and exhibit always a value, for instance a digital input can be HIGH, TRUE or UNKNOWN. Other inputs may have a limited life time and it is the task of a designer to define the moment when their value is to be consumed. Most of the state machines used in the industrial control domain are of the Controller type.

Life-time of control signals

The life-time problem of control signals as well as the life-time of actions in event driven system has been discussed in a previous technical note [2]. To remind we show a simple example *Lifetime* which contains all elements of the topic (Figure 1, Figure 2, Figure 3). We want to repeat an action *Send* triggered by the command *Cmd_Do*. The state machine with two states: *Idle* and *Do* does the task.

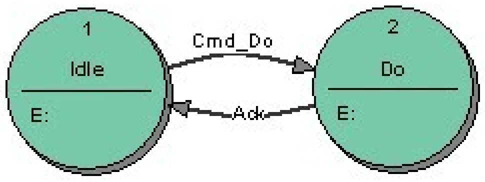
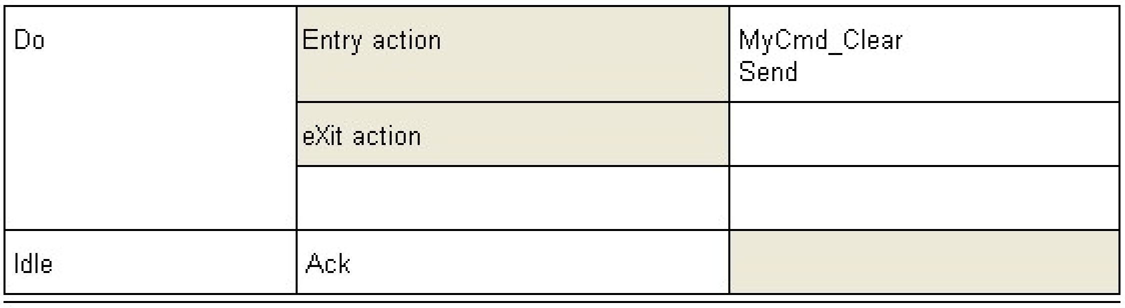


Figure 1: State transition diagram of the state machine Lifetime

The command Cmd_Do forces the transition from the state *Idle* to *Do*. The state machine returns to the state Idle receiving the acknowledgement *Ack*.

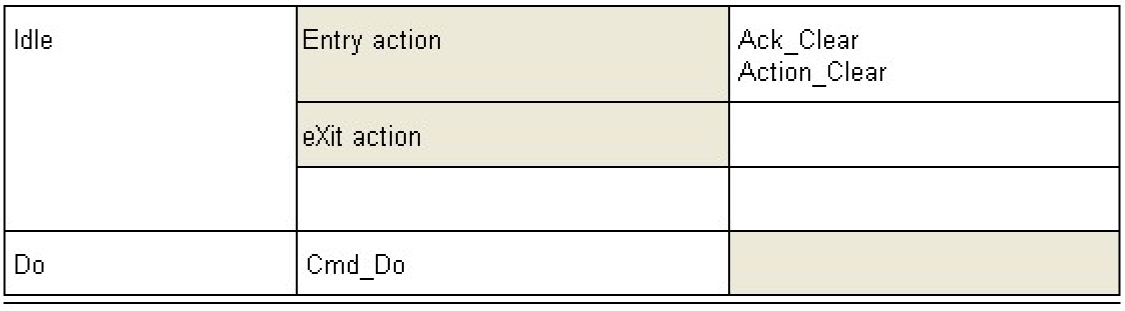


Do	Entry action	MyCmd_Clear Send
	eXit action	
Idle	Ack	

Figure 2: Lifetime – the state transition table of the state Do

On entering the state *Do* the command is cleared (thus being ready for repetition) and the required action (*Send*) is performed.

On entering the state *Idle* both signals: the *Action* and *Ack* are cleared and are ready for repetition (the *Action* may be for instance a command to the IO-Handler).



Idle	Entry action	Ack_Clear Action_Clear
	eXit action	
Do	Cmd_Do	

Figure 3: Lifetime – the state transition table of the state Idle

Pure event driven

There are systems in which input signals are events per definition. In this case we may simplify handling of the input signals: instead of defining in each situation the signal's life-time we do it in one place. This technique will be illustrated by the state machine *AllEvents* (see Figure 4).

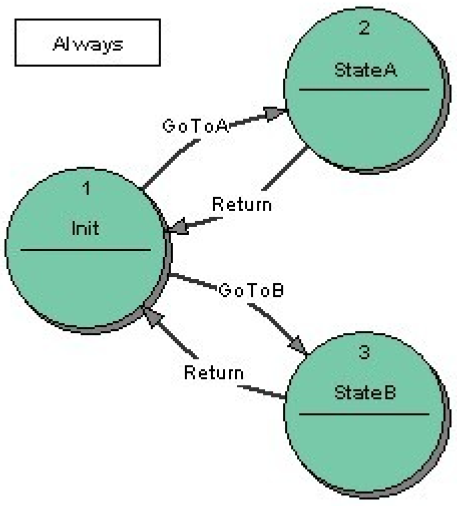
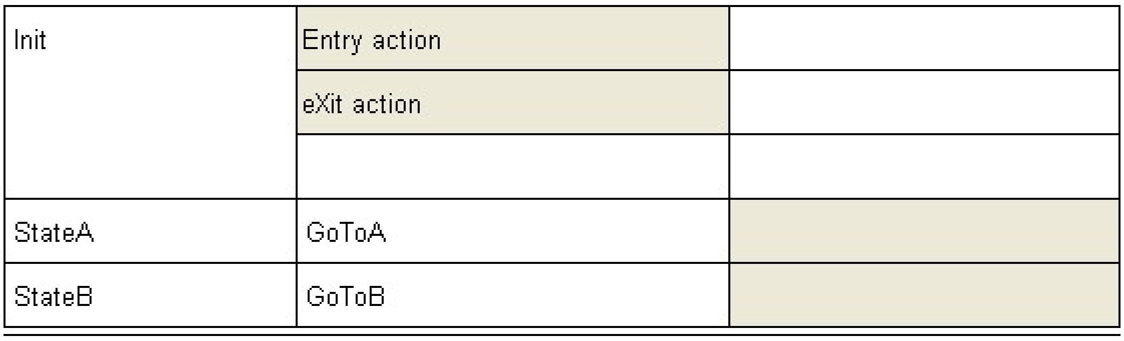


Figure 4: The state transition diagram of the state machine AllEvents

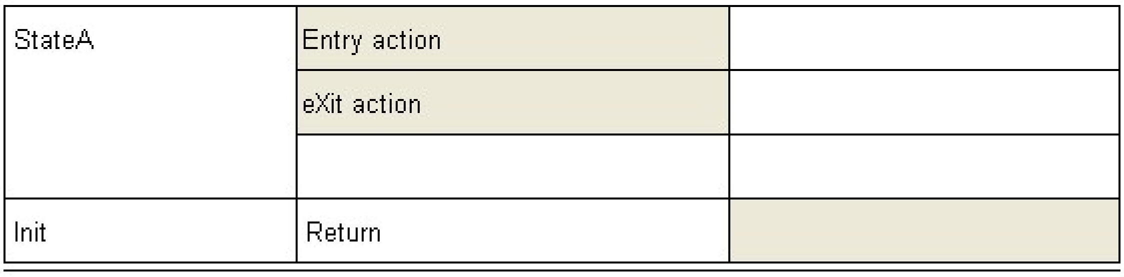
The state machine has two inputs *Input1* and *Input2*. *Input1* generates two control values: *GoToA* and *GoToB*, *Input2* generates a control value *Return*. If the state machine is in the state *Init* (see Figure 5) it goes either to *StateA* or to *StateB* depending on the value of the *Input1*.



Init	Entry action	
	eXit action	
StateA	GoToA	
StateB	GoToB	

Figure 5: AllEvents - the state transition table of the state Init

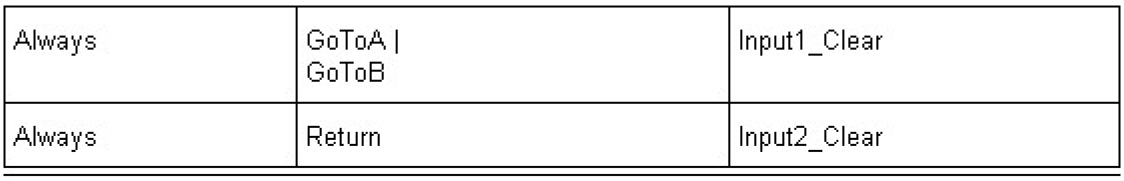
Receiving the input *Return* in the *StateA* (see Figure 6) the state machine returns to the state *Init*. It behaves similarly in *StateB*. Without clearing the input signals in some way the state machine will loop, or oscillate, thereafter because on returning to the state *Idle* it will be forced by a signal *GoToA* (or *GoToB*) to go immediately to the *StateA* (or *StateB*) where it will find the signal *Return*, and so on.



StateA	Entry action	
	eXit action	
Init	Return	

Figure 6: AllEvents - the state transition table of the StateA

Instead of clearing the input signals separately in each state we may do it in the section *Always input actions* (see Figure 7) where signals: *GoToA* and *GoToB* generated by the *Input1* clear the *Input1* and the signal *Return* clears the *Input2*. We may also use a single general clear signal which clears both events triggered by any input signal.



Always	GoToA | GoToB	Input1_Clear
Always	Return	Input2_Clear

Figure 7: AllEvents - the Always input actions

Why must we make life difficult?

Let's recollect some basic principles. A state machine models a behaviour of a control task. It does it by changing states according to input changes. A state machine generates output actions. A state represents the history of all input changes (in a condensed form). Other interpretations are a sin, especially a state should not be used to store the present input value.

Using the Parser type of a state machine we are forced to use states for storing input values. Let's look at the state machines shown in Figure 8 and Figure 9. The state machine is to make a transition from the state *Idle* to *Done* if both inputs arrive: *Event1* and *Event2*. In the Parser type solution we use states: *Storing1* and *Storing2* to store the inputs. If we need to do nothing in those states the solution is probably not optimal as there is no reasonable excuse for using states *Storing1* and *Storing2*.

In the Controller type solution the input values are stored (in StateWORKS as a virtual input) and will be cleared if not relevant any more (for instance in the state *Done*). In effect, the Controller type state machine does not need auxiliary states for storing inputs and the transition from *Idle* to *Done* is performed if both events arrive.

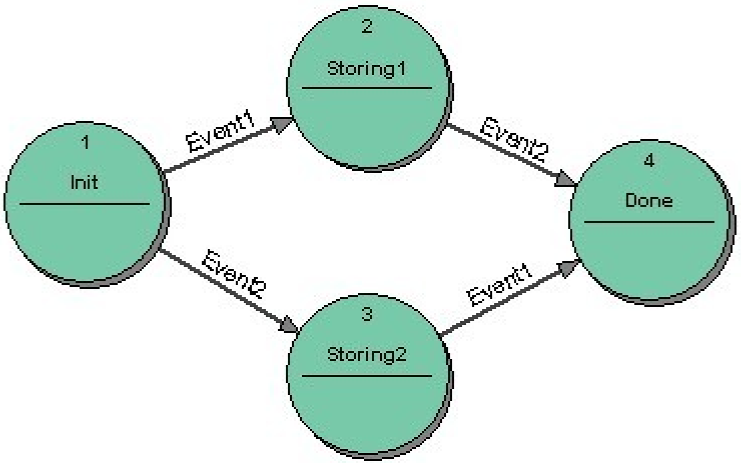


Figure 8: The state transition diagram of the state machine Parser

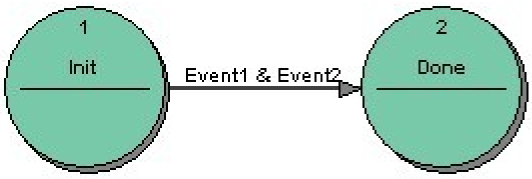


Figure 9: The state transition diagram of the state machine Controller

Conclusions

We should avoid using the Parser type of state machine. Parser type state machine does not allow to use complex transition condition as at any time there is only one input available: the present event. Other control information has to be stored as states that unavoidably leads to a state explosion. Parser state machine should be used if the specification of the inputs indicates clearly that the inputs are used to trigger an action or a transition and loose their meaning after that.

StateWORKS allows definition and implementation of both types of state machines: Parser and Controller. The basic storing mechanism in a form of the virtual input is constructed for static input signals. The designer decides about the life-time of the input signals. In extreme cases a designer may use a pure Parser model consuming all input signals by clearing them in the *Always* input actions.

References

[1] Wagner, F. et al., *Modeling Software with Finite State Machines – A Practical Approach*, Auerbach Publications, New York, 2006.

[2] Technical Note: TN16-Life-time-and-events.pdf. May 2006.