

Building a generic robot engine using the Fischer Technik ROBO Pro environment

This document describes the ROBO Pro program I developed to run my BumberBot model. The BumberBot is based on the ROBO Mobile Set Light Seeker model. My aim was to build a generic robot engine and have the entire robot behavior defined in tables and not in programming code. First I discuss the underlying foundation of the program, a finite state machine. Then I present the state transition diagram that describes the BumberBot behavior. I show how the state transition diagram is translated into a table. I demonstrate how that table is translated into ROBO Pro List elements. Then I explain the algorithm that takes the List elements input to bring the BumberBot alive. Finally, I conclude with some lessons learned and a short wish list for the next version of the ROBO Pro environment.

Finite state machine

I choose a finite state machine as the underlying foundation. A finite state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions. Events trigger the transitions. Typical events for the BumberBot are: 'collision detected' and 'light detected'. A special event I introduced is a timer variable which value has reached zero: 'Time-out'. I used this event to set the duration of motions of the BumberBot, e.g. to make a 90 degree left turn.

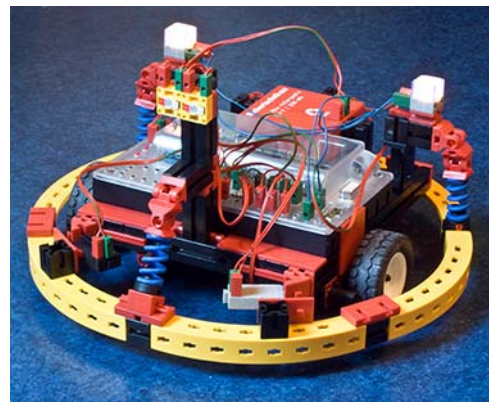


Figure 1 shows the entire state transition diagram describing the behavior of the BumberBot. The main operation of the BumberBot is defined by state 1 and 2. In state 1 the robot drives straight for 5 seconds. In state 2 the robot makes a 270 degree turn. The initial state for the program is state 2. But as the timer variable has not been set, the program automatically activates state 1 and makes the robot drive straight ahead and sets the timer variable to 500ms. The robot flips out of state 1 and 2 when light is detected or when a collision occurs. For a collision state 20 is activated (robot stops, time variable is set to 50ms). When light is detected state 11 is activated (robot homes in on the light, time variable is set to 125ms).

With a frontal collision one of the two switches up front is always pressed first. To be able to detect the frontal decision I implemented a 50ms time delay between state 20 and 21 is needed. The 125ms time value when light is followed serves not to loose the follow state when the light detection is briefly interrupted.

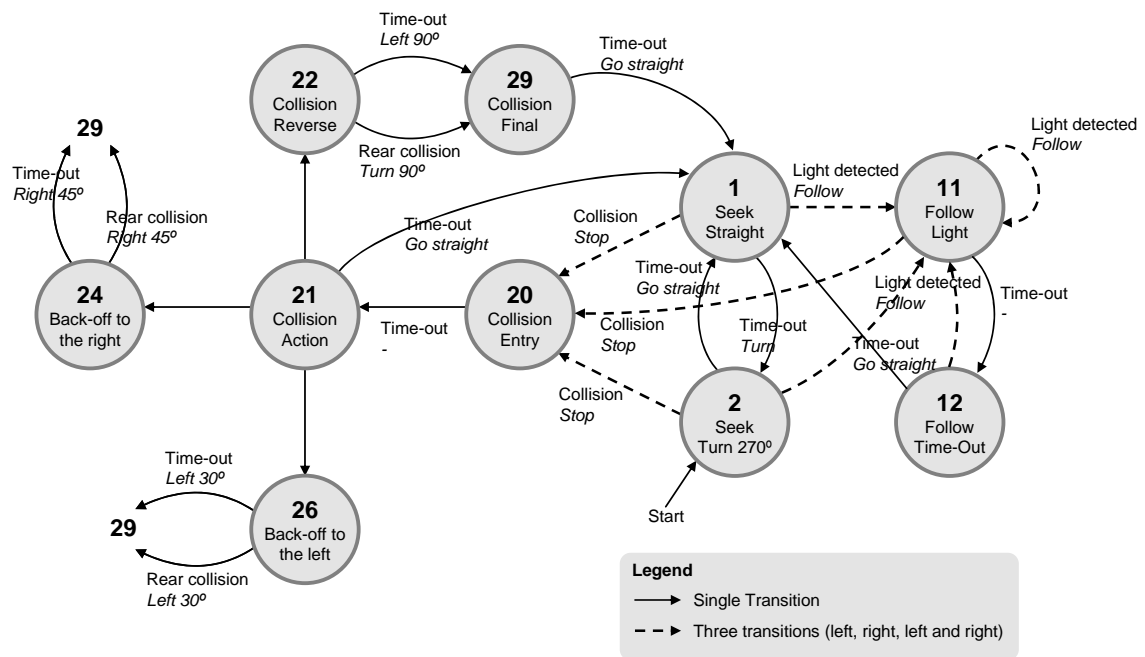


Figure 1 BumberBot State Transition Diagram

Translating the State Transition Diagram into a transition table

I translated the state transition diagram into the transition table as shown below. Each row in the table represents a transition. A speed of '7' indicates that the engine should not be affected, i.e. continue to run as it does.

Bit position	64	32	16	8	4	2	1	128	16	8	1	1	256
		Event Schema						Movement				in 10ms	
		T	I3	I4	I5	I7	I8	M1 - Right		M2 - Left			
Description	State	Timer	FrRight	FrLeft	Rear	TrRight	TrLeft	Direction	Speed	Direction	Speed	Timer Value	Next State
Collision detection	1		1	1					0		0	50	20
	1		1						0		0	50	20
	1			1					0		0	50	20
Seek - Straight	1	1						1	2		2	895	2
Collision detection	2		1	1					0		0	50	20
	2		1						0		0	50	20
	2			1					0		0	50	20
Seek - Turn	2	1							4		4	500	1
Collision Entry	20	1							7		7	0	21
Collision Frontal	21		1	1				1	3	1	3	75	22
Collision Left side	21			1				1	3	1	2	75	24
Collision Right side	21		1					1	2	1	3	75	26
Collision Fake	21	1							4		4	500	1
Frontal C - Back off	22	1							3	1	3	150	29
Left side Collision	24	1						1	3		3	85	29

Bit position	64	32	16	8	4	2	1	128	16	8	1	1	256	
Description	State	Event Schema						Movement				in 10ms	Next State	
		T	I3	I4	I5	I7	I8	M1 - Right		M2 - Left		Timer Value		
		Timer	FrRight	FrLeft	Rear	TrRight	TrLeft	Direction	Speed	Direction	Speed			
Right side Collision	26	1						3		1	3	60	29	
Collision Rear	22	1						3		1	3	150	29	
	24	1						1	3		3	85	29	
	26	1						3		1	3	60	29	
Collision handling	29	1						4			4	500	1	
Seek Light	1	1						5			5	125	11	
	1	1						1			5	125	11	
	1	1						5			1	125	11	
Seek Light	2	1						5			5	125	11	
	2	1						3			5	125	11	
	2	1						5			3	125	11	
Collision detection	11	1		1					0			0	50	20
	11	1						0			0	50	20	
	11	1							0			0	50	20
Seek Light	11	1						5			5	125	11	
	11	1						3			5	125	11	
	11	1						5			3	125	11	
Follow Light	11	1						7			7	0	12	
Still Follow?	12	1						5			5	125	11	
	12	1						3			5	125	11	
	12	1						5			3	125	11	
	12	1						4			4	500	1	

Translating the table into ROBO Pro List elements

It would be possible to define a list element for each column of the transition table. That would lead to 13 different list elements and some rather cumbersome coding. Instead I decided to combine columns into three lists. One for the current state and event (orange), one for the action and next state (green) and one for the value of the timer variable (blue). The colors refer to the bit positions shown in the header of the transition table.

The list element provides 16 bits of storage. I encoded the three list elements as depicted in Figure 2. These values can be manipulated using binary operators as 'left shift', 'right shift' and 'AND'. The AND operator is needed to filter out bits and to verify if the current event mask matches an event held in the event table. In the ROBO Pro environment a left shift is accomplished by dividing the value by a power of 2. Right shift by multiplying by a power of 2. A bitwise AND operator is not directly available in the ROBO Pro environment. I had to write a subprogram to perform this function.

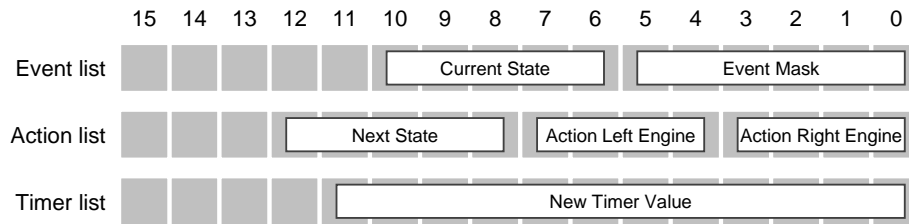


Figure 2 List element encoding

Using the encoding scheme on the first row of the transition table results in these values: event - 88, action - 5120, timer - 50.

Algorithm of the Main Program

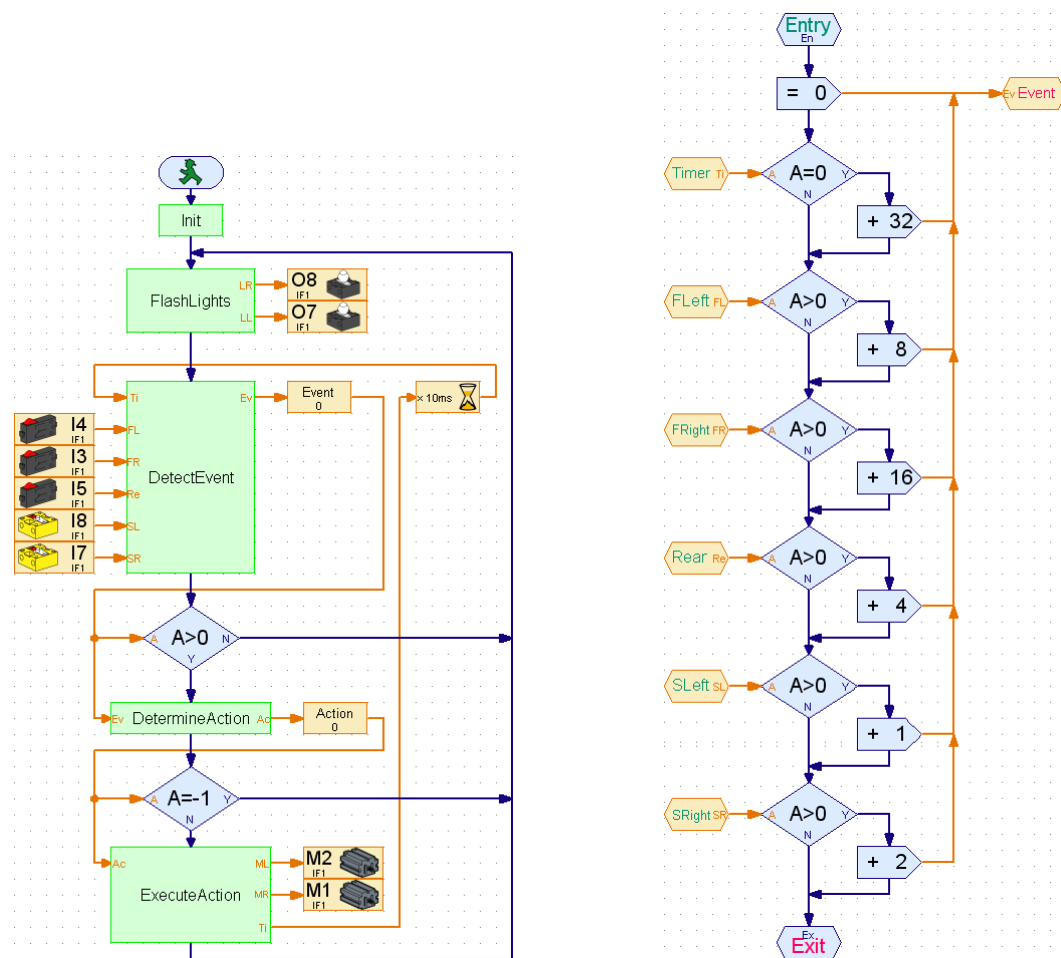


Figure 3 BumberBot Main Program

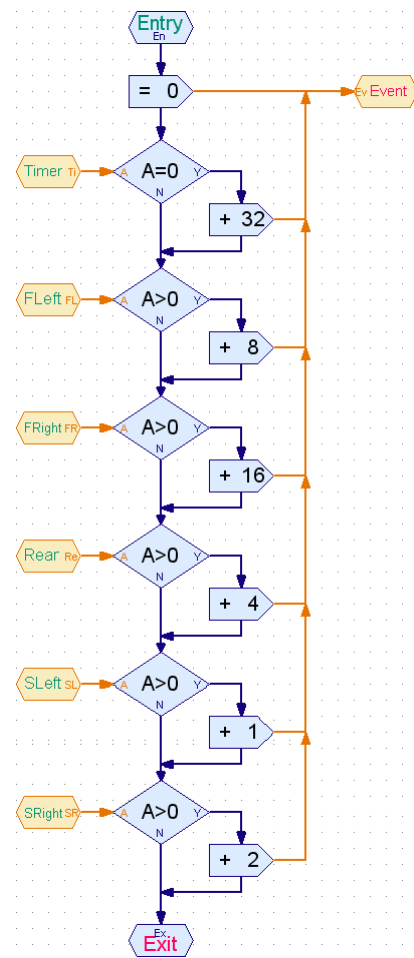


Figure 4 Detect Event

Now that I explained the underlying principles let's have a look at the algorithm of the main program of the generic engine. Figure 3 shows the main flow of the generic engine. It consists of three key steps.

- Step 1 is to detect the current event.
- If an event is detected the engine continues in step 2 to determine the action that needs to be executed.
- If an action is found the engine continues in step 3 to execute the action, set the timer variable to its new value and set the new state of the BumberBot.

The FlashLights routine is a gimmick not relevant for the functioning of the generic engine.

Hence, the main program is non-modal. It continues to run through the steps. Nowhere it waits for input.

Subprograms

The DetectEvent routine is fairly straightforward as depicted in Figure 4. It checks all the inputs. If an input is detected the according bit in the current event variable is set to 1.

The essential routine of the generic engine is shown in Figure 5. It runs through all entries of the event list. For each entry that corresponds with the current state (variable ST) it checks if the current event matches the event necessary for the transition. The reasoning here goes like this: if the current event encompasses all or more of the event bits indicated in the row at hand the row number indicates the transition that needs to take place. This is accomplished in three steps. First the state is filtered out (event list AND 63), then the current event is filtered out (event list AND current event) and finally the routine checks if the remaining bits correspond to the bit mask indicated in the event list. Hence, the order in which the transitions are held in the event list determines the behavior. The action of the first row in the list where all the event bits are covered by the bits of the current event is executed.

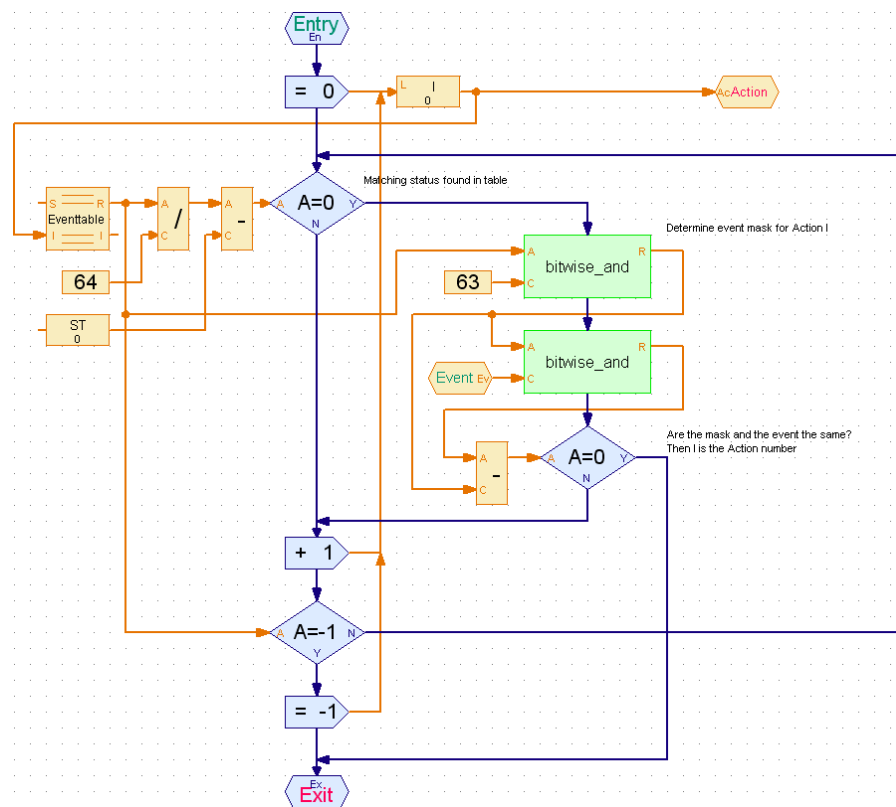


Figure 5 Determine Action

The final subroutine (Figure 6) is to execute the action found. It uses the shift left and bitwise AND operations to derive the speed of both engines and the next state from the action list. It uses the timer list to set the new timer value.

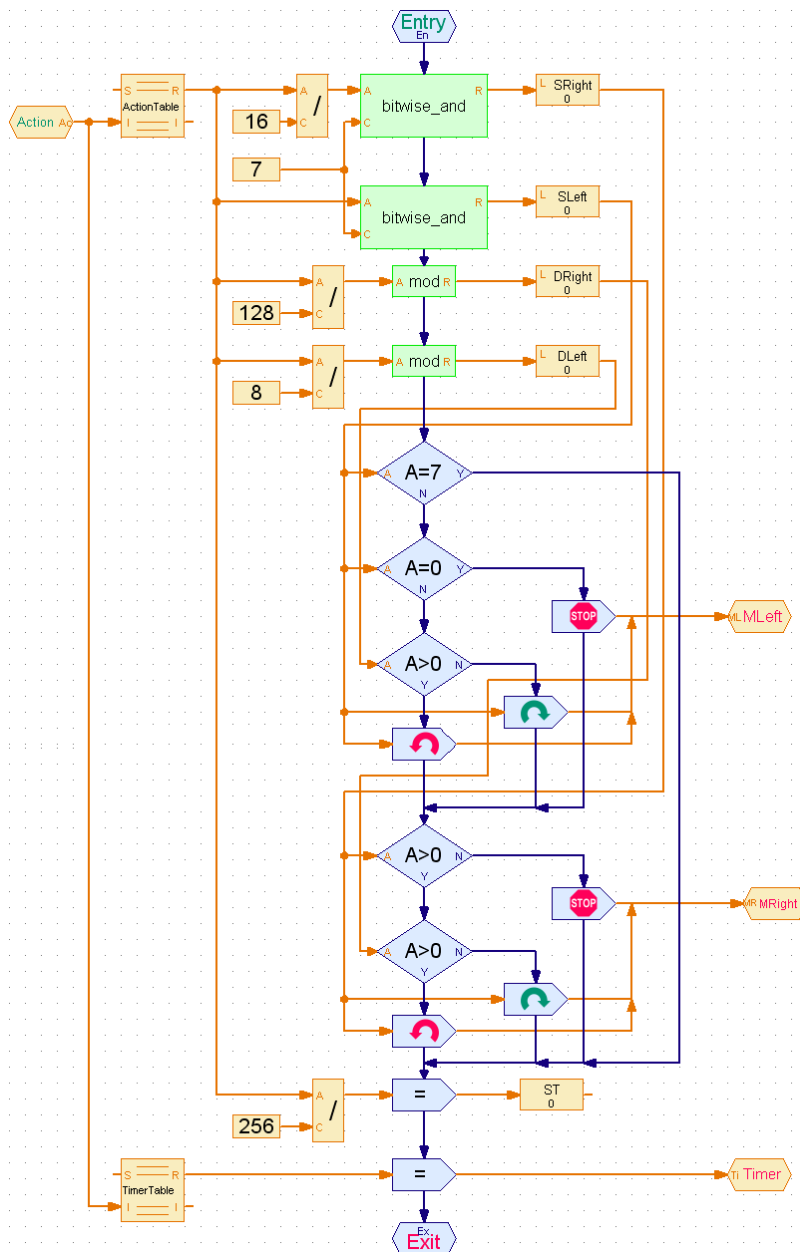


Figure 6 Execute Action

Lessons learned

The BumberBot shows that it is possible to build a relatively 'low weight' generic engine to implement complex robotic behavior using the Fischer Technik ROBO Pro environment. The program would have been a lot more complex and a lot more difficult to maintain if I would have hard coded the behavior.

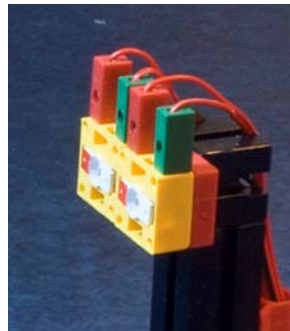
I discovered that the timer variable is not accurate enough to use for robot motions. As the battery power decreases a turn of 90 degrees becomes 80 degrees and less. A next version of the generic engine will hold two other count-down events. One if the pulse counter for the left engine reaches zero and one if the pulse counter for the right engine reaches zero. The transition table will also be expanded with two columns holding the values to set for the left and right pulse counters.

I was a little bit puzzled by the behavior of the ROBO Pro engine commands. I would have expected a value of zero to result in an engine stop and a negative value in a counter clockwise motion. But the

ROBO Pro environment requires different commands for these actions. A next version of the generic engine will include a subprogram that encapsulates this.

The bitwise AND operator I implemented consumes on average 120 CPU cycles. This would be only 1 CPU cycle if the ROBO Pro environment would have provided access to the native bitwise AND instruction of the Renesas M30245 microprocessor. On average I estimate that my program runs 700 times slower than when I could have used the built-in operator. That is significant!

I would like to thank FischerTechnik and Knoblauch for providing the ROBO Interface and programming environment. I enjoy the technology very much. I would like to ask FischerTechnik and Knoblauch to consider to add the native bitwise operators (AND, OR and XOR) in a next release of the ROBO Pro environment so I can speed up my generic engine.



Guido van der Harst

8 February 2009, Doorn, The Netherlands

guido_ruis@hotmail.com