Using Finite State Automata in Robotics

Richard Balogh[1] and David Obdržálek[2]

[1] Slovak University of Technology in Bratislava,
richard.balogh@stuba.sk
[2] Charles University, Czech Republic
David.Obdrzalek@mff.cuni.cz

Abstract. In this paper, we describe the use of the finite state machines (FSM, state automata) in robotics, especially for implementing entry-level control systems. The FSM is defined and couple of explanatory tasks are described using it. Further, several examples of real tasks are presented and implementation sketch which uses FSM is shown. The paper aims to show FSM can be a good tool for implementing control system of a robot, powerful as well as easy to be used.

Keywords: finite state automata, educational robotics, robotic competitions

1 Introduction

Finite state machines (FSM) are one of the many types of automata. In their hierarchy, they lie on a low level, i.e. their relative power is not big in comparison with other types (e.g. Push-down automata or Turing machines). However, they are well sufficient for many tasks in various applications and areas (including robotics) where they can significantly help to efficiently implement repetitive tasks. Moreover, they can contribute to implementation readability and maintainability as they represent a methodical way of using simple construction parts for forming a powerful system. Therefore we believe FSMs are of a great use also for introductory robotics, for example in education or for self-reliant beginners. However, a very typical development process in such cases is creation of a simple dedicated system which gets improved over time to cover more functionality and resolve special cases and specific details. This often leads to a system which grows well beyond understandability and maintainability because of its nature of a monolith system with numerous patches and exceptions. In our paper, we would like to show that using FSMs and their enhanced versions is not difficult at all and can bring the aforementioned advantages which in final effect helps the users to focus more on problem solving than on its low level implementation.

The paper is structured as follows: In Section 2 we give theoretical background and definition of Finite State Machine. Section 3 shows examples of FSM used for implementation of two simple automata, a power switch and a code lock. Section 4 shows the setup for a typical introductory level robot task – a robot that follows a line and a more advanced task – a robot which manipulates items in its working space.

2 Finite State Machines

In mathematics, automata theory is a relatively young topic. It was first studied in the second quarter of 20th century with a need for finite description of infinite objects in logic. The very root of this was set in 1936 by A. M. Turing defining a hypothetical computing machine, today called a Turing Machine [1]; this idea has been further developed to form a whole branch of Formal theory of finite automata in the 60's by today famous mathematicians like A. Church, S. Kleene, E. Post, or A. A. Markov. A whole hierarchy of automata was set, Finite State Automaton being one of them. Although it was originally a purely theoretical tool in mathematics, it soon relieved it has a wide practical usage. For example, a special subset of FSM is a sequential function chart (SFC), for processes which consists of relatively simple sequences of steps. It is used in industry environment as a graphical programming language for programmable logic controllers (PLCs). It is one of the five PLC languages defined by the international standard IEC 61131-3 [2].

Fig. 1 shows an example of a simple FSM, represented as an oriented graph with intuitive interpretation (formal definition follows). It is a graphical description of the system consisting of discrete states represented by graph nodes (circles) and transitions represented by graph edges (arrows). When a transition triggers, the state machine changes its actual state and makes a transition (or jump) to another state, according to its transition function. This FSM features three states S_1, S_2, S_3 and four transitions T_{1a}, T_{1b}, T_2, T_3.

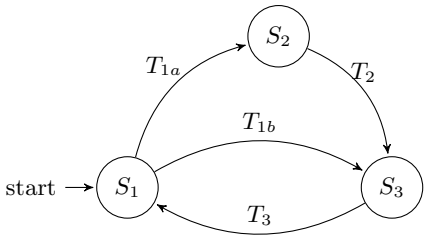

Fig. 1. General example of the state machine diagram.

Formal definition: A FSM is a five-tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite, non-empty set of states,
- Σ is the input alphabet (a finite, non-empty set of symbols),
- δ is the transition function: $\delta : S \times \Sigma \to S$,
- s_0 is the initial state, $s_0 \in S$ and
- F is the final state set, $F \subseteq S$.

Reader may find also other, equivalent definitions of a FSM in various textbooks. This basic definition can be further enhanced in many ways. For our use, we can add triggers which allow changing the state only based on a condition or

attach an action to be performed when a transition changes a state or when a state is entered/left[3]. It can be proven that such additions do not affect the theoretical properties (or power) of a FSM, yet they make the FSM to be practically usable.

The notion of state Intuitively, the state of a system covers its condition at a particular point in time. In general, the state affects how the system reacts to inputs. Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs.

FSM representations and implementations For a given FSM, it is possible to use many different representations. For example, the same FMS as described in Fig. 1 can be also described by a table or a tree as shown in Fig.2.

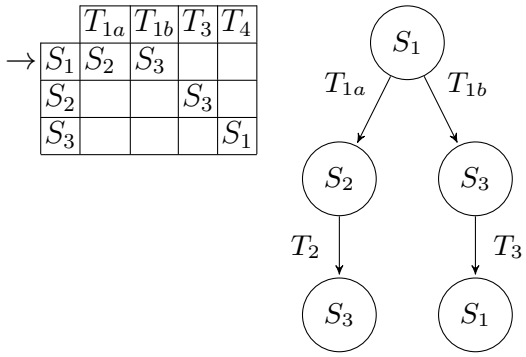

Fig. 2. Alternative FSM representations of the system from Fig.1.

One of the possible implementations of the transition function in the C programming language is shown in the following example:

```
1  enum states {S1, S2, S3};
2  enum states currentState = S1;
3
4  switch( currentState )
5  {
6    case S1:
7      if( T1a() ) currentState = S2;
8      else if( T1b() ) currentState = S3;
9      break;
10   case S2:
11     if( T2() ) currentState = S3;
12     break;
13   case S3:
14     if( T3() ) currentState = S1;
15     break;
16 }
```

[3] Adding an output associated with a state creates a Moore machine, adding an output associated with a state and input (therefore effectively with a transition) creates a Mealy machine.

Only a relevant part of the code is presented; the inputs are implemented as calls to a respective function, returning a boolean value corresponding to the input check.

This coding idea could be very similarly used in other languages like C++, C#, Java, Python and others. In the following text, we will show even the implementation in graphical programming language Microsoft Blocks[4]. Obviously, it is far beyond the scope of this paper to show all the wide varieties of representation and implementation.

Why to use FSM? Finite State Automaton it is a powerful tool for describing sequential behavior of a control program and is easy to understand. It is often used as an communication language between the programmer and process engineer. Using the FSM helps to partition a control problem into a smaller units (partitioning) while at the same time it provides clear and global overview of the process. It also makes it easier for later changes and diagnostics of the system in comparison to monolithic definitions or programming. Last, but not least, when properly implemented, it offers self-documenting features to its users.

Further advantages include: if the program is to be modified, usually it is done at single point only. Modifications to the control section do not affect settings of the outputs. The program is also easy to test. For example, if it is stuck on a certain point, it is easy to determine which transition is not working or which condition is missing. If the real device (a robot) fails, it is usually easy to detect in which state condition it occurred. And last, but not least, it forces an user to structure the programs which in turn minimizes the chance of programming errors (as mentioned e.g. in [3]).

3 Trivial examples and their implementation

General advice for implementing FSM is to split the code into the two parts – one with the transitions only and the second with the execution of outputs. Modified algorithm of the C-code from above is now "rewritten" in Blocks for micro:bit programming language. Please note, that one of the transition conditions was implemented as a timer. This is quite common case, that transition from one state to another is required after certain amount of time.

Typical implementation of the FSM is formed by the nested `if` statements, often replaced by the `switch` statement when available (C, C++). It is perhaps the most popular technique. Signals and states are typically represented as enumerations.

The nested `switch` statement method is simple and understandable and usually has a small (RAM) memory footprint, which is advantageous for embedded systems used in robotics. In the following paragraphs, we show couple of easiest examples.

[4] https://makecode.microbit.org/

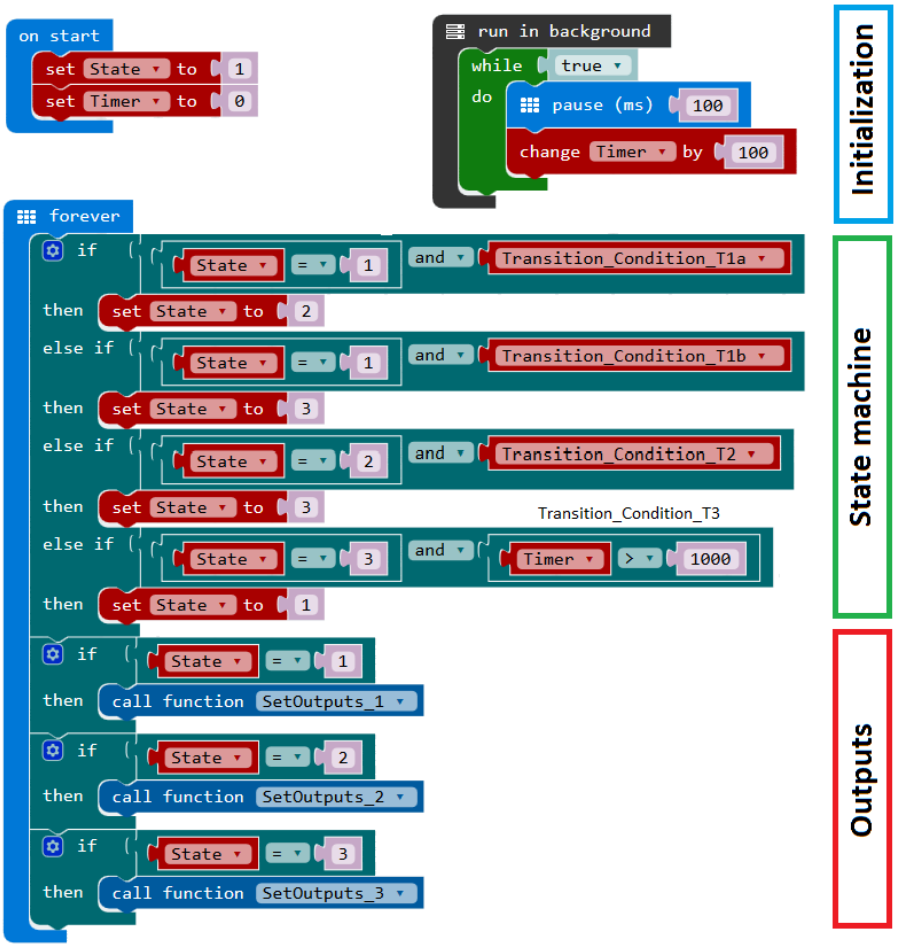

Fig. 3. Generalized implementation of the FSM from the Fig.1.

3.1 On/Off switch – Implementation using conditional statements

The implementation of the simple switch which is used in most of contemporary devices for switching it on and off using the single pushbutton is probably the second program every student wrote after the obligatory LED blink. In the following diagram, the corresponding FSM is shown. There are two states, representing the device ON and device OFF, respectively. A transition occurs when the button is pressed, which is represented by the `board.buttonA.isPressed()` condition in the program below.

We will implement the state automata using conditional statements. This is the simplest way to implement the state machine in the C programming language. It is possible to use the `if-else` or the `switch-case` constructs to check

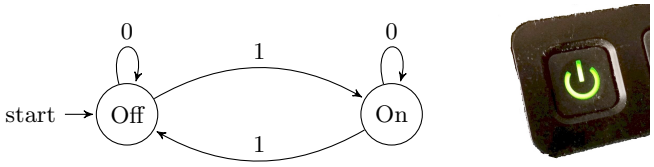

Fig. 4. FSM diagram for on/off switch.

the states and triggered events. If the combination of states and triggered event match, the event handler for such service is executed and the state is updated to the next state as per the transition function definition. It depends on the requirements whether the states or the events are checked first. In the sample code below, the state is verified first and if the transition condition is met, the state changes. After those checks the event triggered by the states are handled based on the current active state. However, it is possible to swap the procedure: check the event first and only then check the state[5].

```
1  enum states { stateON, stateOFF};
2  enum states currentState = stateON;
3
4  int main()
5  {
6    while(1)
7    {
8    switch (currentState)    // The state machine
9     {
10       case stateON:
11         if(board.buttonA.isPressed())
12            currentState = stateOFF;
13         break;
14       case stateOFF:
15         if(board.buttonA.isPressed())
16            currentState = stateON;
17         break;
18    }
19   switch (currentState)    // Actions
20    {
21       case stateON:
22          board.On();
23          break;
24       case stateOFF:
25          board.Off();
26          break;
27    }
28   }
29 }
```

To show how effective is this coding scheme, just imagine that it is necessary to change the behaviour of the system such that the device is controlled by two buttons: one for ON, second for OFF state. In this case, it is necessary to change only a single line: in the condition at line 15, the `buttonA` has to be changed to `buttonB`. No other place in the code needs to be changed.

[5] It can be proven that such change does not affect the theoretical expressivity of the FSM. For practical reasons, this might be of a good use, however.

3.2 Code Lock – Implementation using the transition table matrix

Following diagram is model of the code lock with opening sequence B-A-B. Wrong combination leads back to the initial state Lock1. Note, that the OPEN state is final, marked with double line circle, meaning that this state is no more changed.

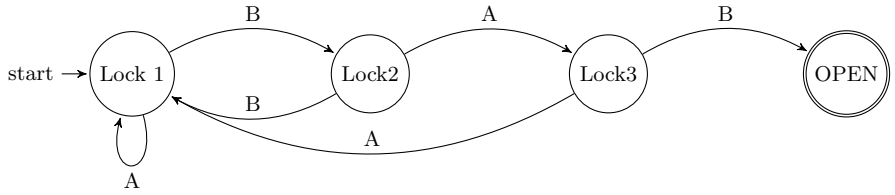

Fig. 5. FSM diagram for simple code lock. Opening sequence is B-A-B.

The transition function can be represented by a table with states on the rows and inputs on the columns, with the required state change (and optionally also the output) in their intersection. For example, for the above code-lock FSM example, the transition table is shown in Tab. 1.

Table 1. Table representation of the FSM above.

	A	B
Lock1	Lock1	Lock2
Lock2	Lock3	Lock1
Lock3	Lock1	Open
Open	Open	Open

This approach gives the opportunity to implement FSM using a transition matrix table and generic table-handling mechanism instead of a long list of `if-else` statements. With this setup, the state machine structure is coded in a table where one dimension describes the states, the other dimension describes the transitions (events), and each element in the table contains the handler for the respective row and column meaning the reaction on that state/event combination. The table is usually implemented using a two-dimensional array of function pointers and the transition function can therefore be implemented as a simple call of a function using current state and inputs as indexes into the table array.

An example implementation for the code lock from Fig.5 is shown below. An advantage of this method is that the handling for every state and event combination is encapsulated in the table. Designer of the system can hold an overall picture of the state machine and software maintenance is also much more under control.

```
enum states { Lock1, Lock2, Lock3, Open };
enum transitions {A, B};

static enum states stateTable[4][2]={
                                    {Lock1,Lock2},
                                    {Lock3,Lock1},
                                    {Lock1,Open },
                                    {Open ,Open }
                                   };

    transition = readButtons();
    nextState = stateTable[currentState][transition];
    currentState =  nextState;

switch(currentState)
 {
  case Lock1:
  case Lock2:
  case Lock3:
    break;
  case Open:
    setOutput();
    break;
  default:
    break;
 }
```

However, the tables used for implementation are usually sparse (see Fig. 2) as many combinations of state/transition are not used or invalid. Then this approach leads to memory wasting. Also, memory requirements increases when the number of states and/or events grows [4].

4 Advanced examples

The robot behavior is often easy to model as a state machine. That is, the outputs of the state machine (e.g. motor feed) depend not only on the inputs (e.g. the bump sensor state) alone[6], but also on the current state of the machine or plan execution phase (e.g. whether the robot is going forward or backward).

4.1 Line Following Robot

In Fig. 6, we show a basic line following robot control as this task is a typical first exercise in robotics. For simplicity, we show an example of a very simple robot with only 2 line sensors working as follows. When a sensor detects the line (black), it outputs logical 1, otherwise it outputs 0. With two sensors only, we have 4 different states with four possible output actions. Note that we can specify also the required output action directly with each state (e.g. goLeft in case the robot is too far right in respect to the line). For simplicity, we don't consider any activity after the robot is lost. Implementation was tested on the micro:bot robot kit[7] and programmed in Block language (see Fig. 7).

[6] Such control is usually called *reactive control*.
[7] https://www.sparkfun.com/products/14216

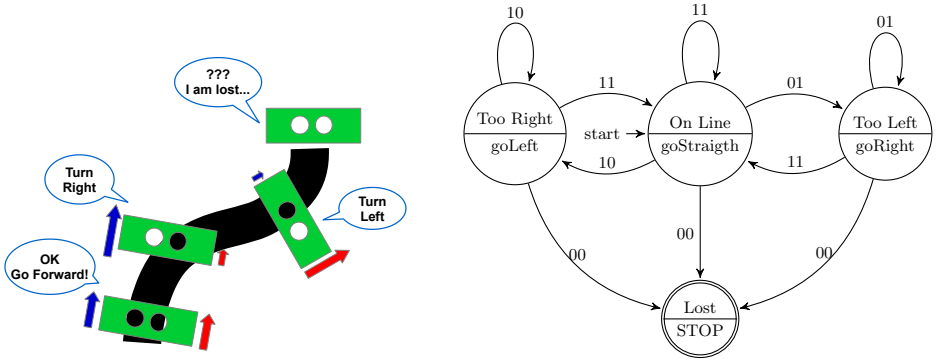

Fig. 6. Line following robot. Sensors detecting line at different phases (left) and corresponding FSM diagram (right).

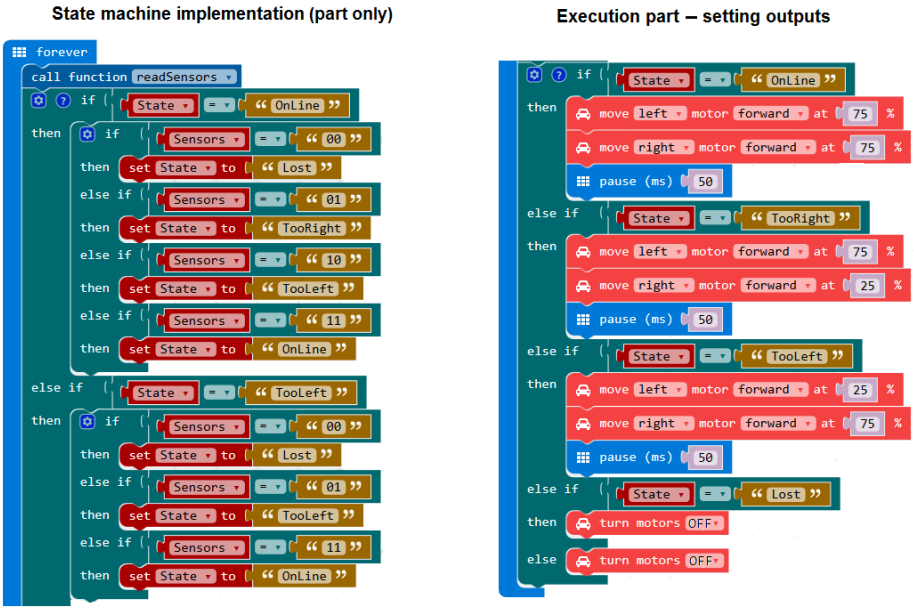

Fig. 7. Implementation of the Line Follower code according the Fig. 6 in Microsoft Make Code Block language.

4.2 Ketchup cans collecting robot

The FSM has been successfully used also for implementing the core control of the "MART Friday Bot". This robot was prepared for the "Ketchup House" competition at Istrobot event [5, 6]. Thanks to the FSM, it was possible to adapt the original program from a choreography-based robot performance "Robot Dance"

to the Ketchup can collecting task in just a few hours[8]. The robot fulfilled the competition rules and (to quite a surprise even of its authors) placed third in 2016 and with a small enhancement for opponent collision avoidance it placed second in 2017. As relieved from talks to other robot builders in this competition, their software implementations differed most notably in lack of global code structure and contained typically lots of code functionalities merged into a single big program which was hard to debug and close to impossible to adapt. On the other hand, the MART Friday Bot code was composed of simpler independent functions which served the FSM in performing individual actions, copying the FSM graph structure. When a problem arose during the preparation and testing, it was simple to find the responsible code and correct it.

The main control state machine of MART Friday Bot is shown in Fig. 8. The robot itself is in Fig. 9.

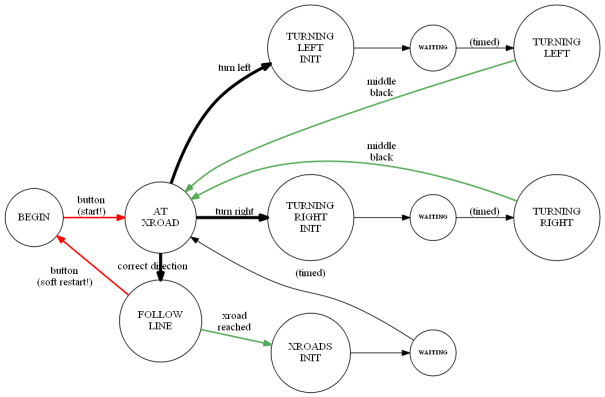

Fig. 8. Main state machine for the MART Friday Bot

5 Conclusions

In this paper, we have presented the Finite State Automata as a possible tool for implementing the core of a robot control. We have shown several examples which should give the reader basic understanding of FSM usage in this area. Our experiences show there are numerous task types where a FSM can significantly help at the implementation phase even if the task seems to be very simple at the first glance. Using the FSM can make the controlling code be readable, understandable and maintainable which is very important especially in education. Let us informally cite one student: "I have daubed everything somehow for the first exercise and it somehow more or less worked. Then I learned about FSMs and implemented the last exercise using it. Had I known it earlier, I would have used it straight on, it was so easy!"

[8] Namely, it was adapted during the 300km car journey from Prague to Bratislava.

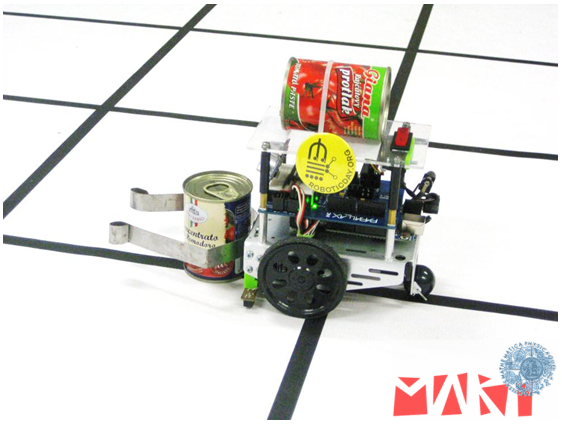

Fig. 9. MART Friday Bot

Advantages of using FSM: The learning curve for using FSM is minor; it is quite likely the reader already knows about FSMs from math or logic courses. Secondly, the integration itself is almost painless, especially when one takes the FSM into account from early stages of the design. On contrary, the transition from spaghetti code to FSM, especially in case there is a lot of stages, may be cumbersome.

Another advantage is that it is possible to test the state machine even before the real hardware testing. There exists simulators, or even the state machine audit trail (let's name e.g. MathWorks Satteflow[9] as an example) so one can test a lot before even obtaining the real hardware. The experiences show that it is possible to remove an important portion of repeating and unnecessary code compared to classical programming approach.

For those who are looking for more sophisticated examples of using FSMs in robotics we can mention e.g. [7] where the authors propose this style of sequential control is a very effective method of implementing complex robot behaviors. Another example which combines the outputs of an artificial neural network for visual navigation with FSM which identifies the robot's current state and defines which action the robot should take according to the processed image frame is given in [8]. Even an example of a humanoid robot behavior coordinated by a hierarchical and asynchronous finite state machine can be found; see [9]. It is also worth mentioning that FSM was one of the approaches successfully tested over five years by the Canadian Space Agency as one of the various autonomy techniques on typical autonomous robotics scenarios [10].

Those who want to implement the above presented techniques do not have to implement everything from scratch. The good news is there are several existing

[9] https://www.mathworks.com/products/stateflow.html

libraries implementing the structure, e.g. for Javascript[10] or even for Arduino[11] which is otherwise infamously know for being used as a "first shoot then look" kind of tools where basic functionality is rapidly made using whichever style and then continuously upgraded and patched resulting in a total mess.

On contrary, it should be also noted that FSM is not a universal magic tool capable of solving everything. There are many areas where using FSM is not appropriate or where it can not be used at all. If nothing else, one should be always aware of its theoretical properties: taking into consideration the Automata theory and Formal grammars, the FSM is able to handle only regular languages which might not be sufficient for solving real world tasks.

Acknowledgements. This paper was supported by the grants of the Research Agency of the Ministry of Education, Science, Research and Sport of the Slovak Republic (VEGA) 1/0819/17 Intelligent mechatronics systems (IMSYS).

References

1. Turing, A. M.: On Computable Numbers with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Ser. 2, Vol. 42, 230–265 (1937)
2. IEC 61131-3 International Standard. Programmable controllers – Part 3: Programming languages. International Electrotechnical Commission, Geneva, Switzerland. ISBN 978-2-83220-661-4. (2013)
3. MicroSystem Simatic S7-200. Two Hour Primer. Siemens AG, Automation and Drives, Nuremberg. (2000)
4. Amlendra: How to implement finite state machine in C. Available on-line: https://aticleworld.com/state-machine-using-c/ (2017)
5. Balogh, R.: Ketchup House – A Promising Robotic Contest. In 3rd Conference on Robotics in Education 2012. Praha. Volume 3, pp. 41-45. Matfyzpress. (2012)
6. Robotika.SK: Istrobot 2017 Ketchup House Competition. (2017) http://www.robotika.sk/contest/2017/indexEN.php?page=rules&type=ketchup
7. Blank, D., Kumar, D., Meeden, L. and Yanco, H.: The Pyro toolkit for AI and robotics. AI magazine, 27(1), 39. (2006)
8. Sales, D. O., Shinzato, P., Pessin, G., Wolf, D. F. and Osorio, F. S.: Vision-based autonomous navigation system using ANN and FSM control. In Proceedings of 2010 Latin American Robotics Symposium and Intelligent Robotic Meeting (LARS), pp. 85-90, IEEE. (2010)
9. Tellez, R., Ferro, F., Garcia, S., Gomez, E., Jorge, E., Mora, D. and Faconti, D.: Reem-B: An autonomous lightweight human-size humanoid robot. In Proceedings of the 8th IEEE-RAS International Conference on Humanoid Robots, pp. 462-468, IEEE. (2008)
10. Dupuis, E., Allard, P. and L'Archevêque, R.: Autonomous robotics toolbox. In Proceedings of International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), Munich, Germany. (2005)

[10] https://github.com/jakesgordon/javascript-state-machine
[11] http://playground.arduino.cc/Code/FiniteStateMachine