Journal of Data Acquisition and Processing
ISSN: 1004-9037
https://sjcjycl.cn/
DOI: 10.5281/zenodo.7762185

MODEL A FINITE STATE MACHINE AS A CONSTRUCT IN COMPUTER PROGRAMMING

Sachin Kadam[1], Shweta Jogalekar[2], Satyawan Hembade[3]

[1] Professor, Institute of Management and Entrepreneurship Development, Bharati Vidyapeeth (Deemed to be University),
Pune (India)

[2]Assistant Professor, Institute of Management and Entrepreneurship Development, Bharati Vidyapeeth (Deemed to be University), Pune (India)

[3]Associate Professor, Institute of Management and Entrepreneurship Development, Bharati Vidyapeeth (Deemed to be University), Pune (India)

ABSTRACT

The concept of Finite State Machine(FSM) can be used to model state-based computing systems. It provides an elegant way, to many of the complex problems in computing domain. Generally, it also forms an important part of computer science curriculum. But unfortunately, the developers hardly use this beautiful problem modeling technique to implement real life solutions. This research article explores a possibility to bridge this gap between theoretical aspects of FSM and its practical application in computer programming by extending FSM as a construct in computer programming. It follows Case Study research methodology to first analyze a specific scenario in detail. This is then generalized into an algorithm (FSM2Construct) using Design and Creation research methodology. FSM2Construct algorithm is designed using Greedy algorithm design strategy.

Keywords: Finite State Machine, FSM, Computer Programming Constructs, Greedy Algorithm Design Strategy, FSM2Construct Algorithm

1: Introduction

This research article explores a possibility of synthesizing two concepts from computer science namely Finite State Machine and Constructs in Programming.

1.1: Finite State Machine (FSM)

FSM can be perceived as an application of graphs where FSM is a device whose operation is composed of modes called *states* (of which there are only finite numbers). The machine can transit from one state to another as per appropriate input. It consists of a finite number of states and produces outputs on state transitions after receiving output. (Lee & Yannakakis, 1996). FSM plays a vital role in the digital design. Prominently it is used to model sequential circuits, communication protocols etc. (Lee & Yannakakis, 1996). Most digital designs rely on Finite State Machines (FSMs). The central concept behind an FSM is to store a series of distinct states and transition between them in accordance with the values of the inputs and the machine's current state. There are two different forms of FSM: Moore (where the state machine's output is solely based on the variables of the state) and Mealy (where the values of the current state variables and input determines the output). (Wilson & Mantooth, 2013). FSM is also referred

as *finite state automata* or *finite automata*. The states of a finite automata can be depicted with a directed graph known as *state transition diagram*(Aho, Hopcroft, & Ullman, 1974).

1.2: Programming Constructs

The syntax and behavioral aspect of programming languages are concisely specified by its fundamental constructs. (Mosses, 2021) . The theoretical aspect of computer science identifies three fundamental programming constructs namely sequence, decision and iteration (Grogono & Nelson, 1882). Various extensions of these structures (e.g., Switch-Case extension of Decision Construct) are generally provided by almost all empirical programming languages(Wirth, 2005). These extensions are to be considered as theoretically similar with their fundamental counterparts. The other constructs like recursion, multithreading, exception handling or routines covering parameter passing are core concepts in object orientation. (Sajaniemi & Chenglie, 2007).

2: Significance of Research

Computer software is categorized in two fundamental classes; Application and System Software(French, 1985).

- Application Software categorizes applications directly used by end users (e.g. business applications).
- System Software categorizes niche applications like operating systems, compilers and interpreters used by computer application developers to develop Application Software.

FSM's are a commonly used in the design of a logical circuit. They are well-suited to design systems that loop over several different alternative actions based on input. This makes it apt for system software developers to model and develop system software (Aho, Hopcroft, & Ullman, 1974). FSMs can be proved beneficial for embedded systems as well because of their efficiency of using limited resources of a system (Drumea & Popescu, 2004). But Application Software developers generally consider FSM as a theoretical concept in computer science with little practical application to solve real world problems. (Wagner, Schmuki, & Wolstenholme, 2006), applied FSM to software development. It offers a critical evaluation of the idea of employing executable specifications as a foundation for FSM in order to streamline software development processes and boost quality.

Researchers are of the opinion that FSM can also be used as an elegant mechanism by application software developers to solve real world problems. The theoretical rigor intrinsic in a FSM will result in robust application software.

3: Research Problem

FSM is a model of behavior. It is generally considered as an abstract concept defining a protocol for progressing through a limited (finite) number of positions (state) each performing some processing and then selecting the next state, usually based on the next piece of input.

Programming is also considered as an abstract concept in computer science and generally treated independent of a programming language. Various constructs used in a program are also considered to be theoretical in nature and language independent.

A synthesis of these two concepts may provide an opportunity to enhance understanding and application of FSM from practical perspective. This research article proposes to synthesize these two abstract concepts together and provide an applied dimension to FSM in the form of a computer programming construct.

Problem Statement:
An abstract FSM can be converted intoconcrete applicationsoftware by implementing it as a computer programming construct following an algorithm as a model.

4: Research Methodology
This research proposes to develop a new algorithm. Researcher has selected *Design and Creation research methodology* for this research as it provides a structured approach to design and develop a new artifact(Oates, 2005).The learning-through-making approach is the emphasis of the design and creation research strategy (Besoain, Jego, & Gallardo, 2021). This methodology involves six process steps to help a researcher to develop an artifact by exploring functional capabilities of existing systems(Daud & David, 2011);

1. Awareness of the Problem: to understand need for new artifact
2. Suggestions: to explore and use functional capabilities of existing systems
3. Development: to match requirements from first step and suggestions from second step to create a new artifact
4. Evaluation: to access the feasibility of new artifact with respect to requirements from first step
5. Conclusion: to summarize contribution of the research

The research contents of this article and research outputs at various stages can roughly be mapped with process steps involved in Design and Creation research methodology as follows:

Step 1: Sections on *Introduction*, *Significance of Research* and *Research Problem* (Output: *Problem Statement*)
Step 2: Section on *FSM as a Programming Construct* (Output: *Need for an algorithm to convert a FSM into a programming construct*)
Step 3: Sections on *Case Study* and *Algorithm to Model FSM as Programming Construct* (Output: *FSM2Construct Algorithm*)
Step 4: Sections on *Analysis of FSM2Construct Algorithm* (Output:*Analysis of Correctness of the Algorithm, Time Complexity and Space Complexity*)
Step 5: Sections on *Findings and Interpretations* and *Conclusion* (Output: *Conclusive Summary*)

5: FSM as a Programming Construct
Analysis of abstract nature of FSM and abstract nature of Programming has revealed similarity in both the concepts. Both of them can be treated in a similar fashion through a Table metaphor. This metaphor will have a table that holds all possible states of a FSM, and lists the actions to do when you enter each state. The last action is to calculate (often by a further table lookup

based on the state you are in and the next input token) what state to enter next. You start in a state known as the *initial-state*. Along the way, your transition table might tell you to enter an error state, signifying an unexpected or erroneous input. You continue to make state transitions until you arrive at the end state.A further analysis revealed that it is well-suited to programs that loop over several different alternative actions based on input.

6: Case Study

6.1: Problem

Let us consider the problem to check the presence of a substring *s1* in a given input string *s2*. The following two scenarios illustrate the problem with examples:

Scenario #1: If s1= “mat” and s2=“automata”, then the output should be "'mat' substring is present."

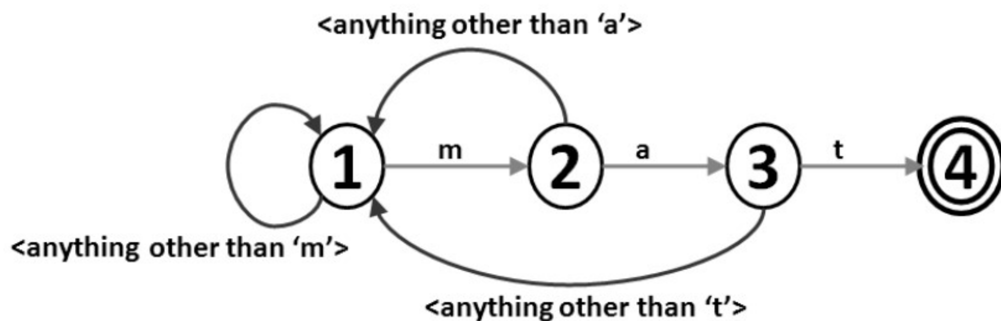
Scenario #2: If s1= “mat” and s2= “automaat”, then the output should be "'mat' substring is not present."

6.2: Solution to Problem

The scenarios stated in the case study can be modeled as a FSM.The first thing required to follow the FSM approach is to identify different states involved in the system. There must be finite number of these states. In our case study we can identify four such states as follows:

State #1: Start state
State #2: If input ‘m’ is recognized
State #3: If input ‘a’ is recognized after ‘m’
State #4: If input ‘t’ is recognized after ‘ma’

This can be represented as a state transition diagram as follows (Fig. 1):



(Fig. 1: State Transition Diagramrepresentation of the Case Study)

A state transition diagram is a directed graph. Fig. 1 can be textually described as follows:

1. State-1 is the starting state (initial state).

2. If input “m” is recognized when in state-1 thsen the system will advance to state-2. If any input other than “m” is recognized when in state-1 then the system will remain in state-1.
3. If input “a” is recognized when in state-2 then the system will advance to state-3. If any input other than “a” is recognized when in state-2 then the system will return back to state-1.
4. If input “t” is recognized when in state-3 then the system will advance to state-4. If any input other than “t” is recognized when in state-3 then the system will return back to state-1.
5. If the system is in state-4, then it indicates that the substring “mat” is recognized. Thus state-4 is the end state depicting the success.

This illustrates that the working of a finite state machine is governed by the current state and the current input. Their combination decides the next state of the machine. The above mentioned state transition diagram (Fig. 1) can be represented as a state transition table as follows (Table1):

Current State	**Input**			
	m	**a**	**t**	**<any other character>**
State 1	State 2	State 1	State 1	State 1
State 2	State 1	State 3	State 1	State 1
State 3	State 1	State 1	State 4	State 1

(Table1: State Transition Table representation of the Case Study)

In the above state transition table the table values represent the next states for given current state and input values.

6.3: Implementation of State Transition Table in a Computer Application

Now let us consider the implementation aspect of this FSM. Because the finite state machines are represented with state transition diagrams, which are directed graphs, we can implement them using the graph data structures. The graph data structures provide the most natural and flexible way to implement the FSMs, but they are relatively difficult to represent in a computer program.

An alternative approach can be selected which will focus on state transition table corresponding to the FSM. The Switch-Case programming construct can be used to represent the state transition table given in Table 1. The use of Switch-Case construct to implement state transition table will make it fundamentally similar to Decision construct.

7: Algorithm to Model FSM as Programming Construct

An algorithm provides a procedure consisting of finite number of simple and unambiguous steps to solve the problem. It is represented in the form of natural language. It is then represented into pseudo-code for ease of conversion into a computer program(Dromey, 1994). The specific problem discussed in case study can be generalized as an algorithm to be applicable for any FSM.

7.1: Algorithm

Algorithm Design Method:

This research proposes to use one of the established methods of algorithm design. The state-to-state movement of an FSM suggests Greedy Algorithm Design Method to be suitable here.

Greedy Method: In this algorithm design method we try to design an optimal solution in stages. We make a best possible decision at each stage which cannot be changed at latter stages(Skiena, 2007)(Cormen, Leiserson, Rivest, & Stenic, 2008).

Listing 1 provides an algorithm to convert a FSM into a programming construct. Researcher has named this algorithm as *FSM2Construct Algorithm*.

FSM2Construct Algorithm:

1. A FSM consists of one initial-state, multiple intermediate-states and one or more exit-states. Represent each state in the form of an individual subroutine.
2. Set the current-state to initial-state and input-variable to NULL.
3. Accept input to FSM in input-variable.
4. Execute an appropriate subroutine based upon current-state and value of input-variable.
5. Repeat Step 3 and 4 till you reach one of the exit-states.

(Listing 1: FSM2ConstructAlgorithm)

7.2: Pseudo-Code representation of the Algorithm

The FSM2Constructalgorithm in Listing 1 can be represented in the form of pseudo-code (Dromey, 1994)to ease its implementation in any empirical programming language supporting Switch-Case construct (Listing 2).

subroutine State1 (**var**InputToFSM)
begin
processing as per requirement of this state with respect to value of InputToFSM variable
{**assert:** set the CurrentState as per FSM}
end

subroutine State2 (**var**InputToFSM)
begin
processing as per requirement of this state with respect to value of InputToFSM variable
{**assert:** set the CurrentState as per FSM}
end

subroutineStateN (**var**InputToFSM)

```
begin
        processing as per requirement of this state with respect to value of InputToFSM variable
        {assert: set the CurrentState as per FSM}
end

program FSM (input, output)
        varCurrentState {variable to keep track of current state}
        varInputToFSM { variable to accept input value for FSM}
begin
        {assert: State1 is initial-state}
        {assert: set the current-state as State1}
        {assert:InputToFSM is NULL}
        repeat
        begin
                get InputToFSM
                {assert:InputToFSM is a valid value}
                switch (CurrentState)
                begin
                        case State1: goto subroutine State1(InputToFSM)
                        case State2: goto subroutine State2(InputToFSM)
                        . . .
                        caseStateN: goto subroutine StateN(InputToFSM)
                end
                {assert:CurrentState is a valid state}
        until (CurrentState is not an Exit State)
        {assert:CurrentStateis in an exit state}
end
```

(Listing 2: Pseudo-code forFSM2Construct Algorithm)

8: Analysis of FSM2Construct Algorithm

This research has proposed FSM2Construct algorithm to model a FSM as a construct in programming language. Analysis of an algorithm is performed as per three parameters – Correctness, Time Complexity and Space Complexity (Sedgewick, 2008). In this section FSM2Construct algorithm is analyzed accordingly.

8.1: Correctness

Correctness of an algorithm verifies that the results obtained from the algorithm (i.e. output) are in accord with formally defined output specifications (Dromey, 1994). Correctness of an algorithm can be verified through a mathematical-proof and/or a trace-table. The State Transition Table corresponding to a FSM provides an intrinsic trace-table automatically verifying its correctness. Researcher has practically tested this by successfully modeling the FSM discussed in case study. Researcher also has successfully implemented the same as a computer program following the corresponding pseudo-code (Listing 2).

8.2: Time Complexity

Time Complexity of an algorithm provides a hardware independent metric to identify possible amount of computer time it needs for execution (Sahni, 2000). It follows a four-step approach to do so.

1. Identify *key-operation* involved in the algorithm.
2. Identify *best-case*, *average* and *worst-case* execution scenarios for the algorithm.
3. Calculate execution frequency of*key-operation*for these scenarios.
4. Establishes a relation between number-of-inputs and execution frequency of the key-operation.

Changing the Current-State as per input is identified as the key-operation for FSM2Construct algorithm. Its execution frequency is directly proportional to number of inputs. This results into linear function for Time Complexity.

$$\text{Time Complexity} = O(n), \text{ where } n \text{ is number of inputs}$$

8.3: Space Complexity

Space Complexity of an algorithm provides a hardware independent metric to calculate space required for its execution(Sahni, 2000). It is stated as;

$$S(P) = C + Sp(\text{instance characteristics})$$

where,

- S(P) is total space requirement of program P.
- C is a constant denoting fixed part of space requirement independent of instance characteristics. It includes space for instruction set, fixed-size variables and constants.
- Sp denotes variable component of space requirement dependent on dynamically allocated space as per problem instance.

Analysis of FSM2Construct algorithm reveals that it does not require any dynamically allocated memory. Therefor its total space requirement is constant.

$$\text{Space Complexity} = C, \text{ where } C \text{ is a constant}$$

9: Findings and Interpretations:

The findings and interpretations of this research can be summarized as follows:

Correctness of the algorithm ensures that a FSM can be modeled as a programming construct. This provides a guideline framework to application programmers to use FSM approach in programming. It will encourage application programmers to model their applications on FSM.

The proposed algorithm exhibits *O(n)* Time Complexity. This is linear in nature. This ensures that it will take finite time in proportion with number of inputs for completion. Thus it is applicable to solve real-world problems involved in application software domain.

The proposed algorithm exhibitsconstant Space Complexity. This ensures that a FSM model can be converted into an application program which will execute on a generic computer system. Thusit is viable to solve practical problems using hardware available with genericcomputer systems.

Observation of Listing 2 reveals programming language independent nature of the pseudo-code. It can be easily converted into a computer program using any empirical programming languagelike C, C++, Java, C# etc.

The solution modeled around a FSMis generally modular in nature making it flexible. This can be illustrated through a new scenario with respect to discussed case study.

Scenario #3: The substring search should be case insensitive.

The modular nature of FSM model allows this change through a simple modification in Switch-Case construct in pseudo-code. This emphasizes the elegance of proposed model.

10: Conclusion

FSM is considered as a tool to provide technology independent understanding of fundamental concepts involved in computer science. This approach overshadows practical applications of FSM in programming domain. This research article explored FSM as a modeling technique to implement application software by extending FSM as a programming construct. A guideline framework was proposed in the form of FSM2Construct algorithm to do so. FSM2Construct algorithm was then analyzed for its correctness, time complexity and space complexity. The analysis established its practical viability in application software domain. This was further supported with its flexibility due to modular nature.

11: References

1. Aho, A., Hopcroft, J., & Ullman, J. (1974). *The Design and Analysis of Computer Algorithms.* Wesley Publishing Company Inc.
2. Besoain, F., Jego, L., & Gallardo, I. (2021). Developing a virtual Museum: Experience from the Design and Creation Process. p. 12(6):244.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stenic, C. (2008). *Introduction to Algotithms.* Prentice Hall of India Private Limited.
4. Daud, A. M., & David, S. (2011). Design Science Research Methodology: An Artefact - Centric Creation and Evaluation Approach. *ACIS 2011 Proceedings.*
5. Dromey, R. G. (1994). *How to solve it by Computer.* Prentice - Hall of India Privalte Limited.
6. Drumea, A., & Popescu, C. (2004). Finite state machines and their applications in software for industrial control. *Intenational Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress.1*, pp. 25-29. Bankaya, Bulgaria: IEEE. doi:10.1109/ISSE.2004.1490370

7. French, C. S. (1985). *Computer Science - An Instructional Manual.* DP Publications Ltd.
8. Grogono, P., & Nelson, S. H. (1882). *Problem Sovling and Computer Programming* . Addison Wesley Publishing Company Inc.
9. Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines - A Survey. *Proceedings of the IEEE*, (pp. 1090 - 1123).
10. Mosses, P. D. (2021). Fundamental constructs in Programming Languages. *Lecture Notes in Computer Science*, *13036.* doi:10.1007/978-3-030-89159-6_19
11. Oates, B. J. (2005). *Researching Information Systems and Computing.* SAGE Publications.
12. Sahni, S. (2000). *Data Structures, Algorithms and Applications in C++.* McGraw-Hill International Edition.
13. Sajaniemi, J., & Chenglie, H. (2007). Teaching Programming: Going beyond "Object First"., (pp. 255-265). Finland.
14. Sedgewick, R. (2008). *Algorithms in C++.* Pearson Education Inc.
15. Skiena, S. S. (2007). *The Algorithm Design Manual.* Springer Science + Business Media Inc.
16. Wagner, F., Schmuki, R., & Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach* . Auerbach Publications.
17. Wilson, P., & Mantooth, A. (2013). *Model-Based Engineering for Complex Electronic Systems.* doi:https://doi.org/10.1016/B978-0-12-385085-0.00006-3.
18. Wirth, N. (2005). *Algorithm + Data Structures = Programs.* Prentice - Hall of India Private Limited.