Comparison of Procedural and Object-Oriented Implementations of an FSA

MODULE: CSC1055 – Comparative Programming Languages

YEAR OF STUDY: 3

COHORT: COMSCI

GROUP: Group J

Group Members:

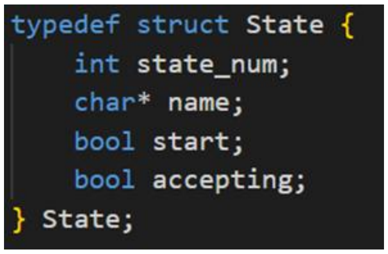
Student Name	**Student ID**	**Programming Paradigm/Language**
Conor Weir	[redacted]	**Object-Oriented/Java**
Andrew Brady	[redacted]	**Procedural/C**

Table of Contents

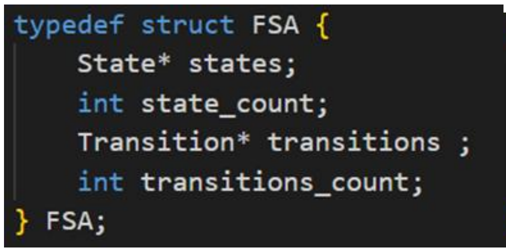
Critique of Procedural Implementation (C)

When tasked with the problem of implementing a Finite State Automata (FSA) in C, I discovered many issues but also realised the benefits this language brings to tackling a problem like this. Using data structures and implementing procedures to perform certain tasks lead to a very clean structure of the program.

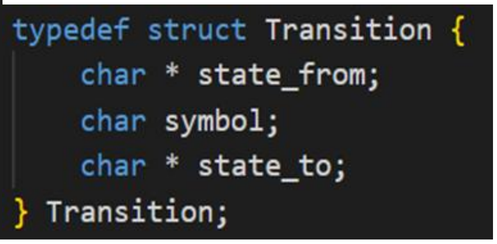
The first data structure used for my implementation of the FSA was the use of structs. I broke the down my FSA struct by having sub structs nested within. These two sub-structs were labelled as State and Transition and this made it easier to point to the necessary data to manipulate within my code.



```c
typedef struct State {
    int state_num;
    char* name;
    bool start;
    bool accepting;
} State;
```

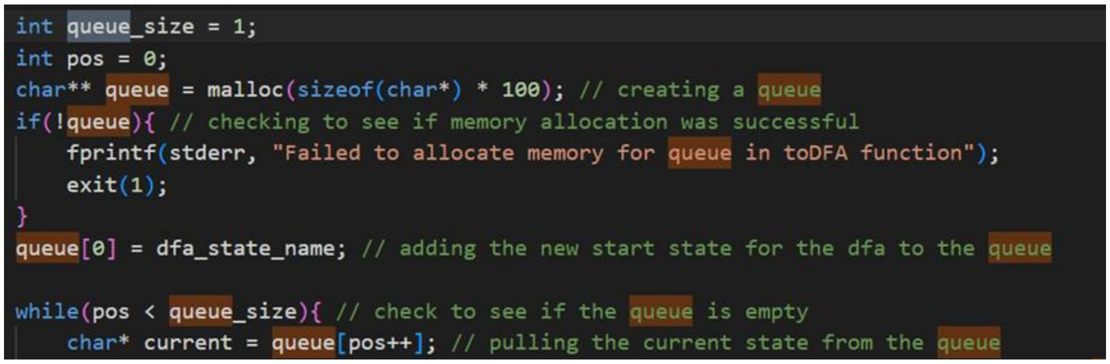


```c
typedef struct FSA {
    State* states;
    int state_count;
    Transition* transitions ;
    int transitions_count;
} FSA;
```



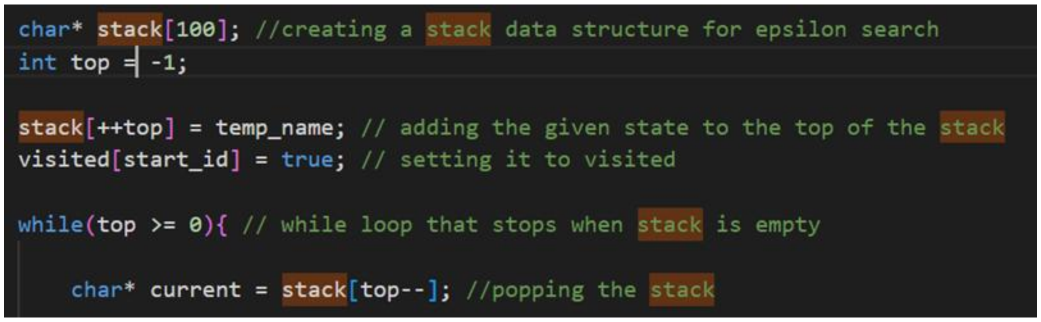
```c
typedef struct Transition {
    char * state_from;
    char symbol;
    char * state_to;
} Transition;
```

I followed this up with my use of stacks and queues to handle traversing through the FSA. The queue was used in the implementation of my toDFA function to follow the First-in First-out (FIFO) principle to allow me to traverse through the previous NFA to create the new DFA names and the transitions between them.



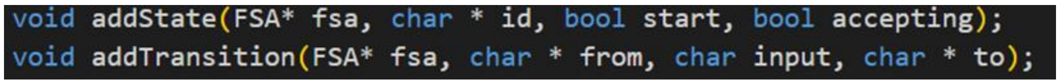
```c
int queue_size = 1;
int pos = 0;
char** queue = malloc(sizeof(char*) * 100); // creating a queue
if(!queue){ // checking to see if memory allocation was successful
    fprintf(stderr, "Failed to allocate memory for queue in toDFA function");
    exit(1);
}
queue[0] = dfa_state_name; // adding the new start state for the dfa to the queue

while(pos < queue_size){ // check to see if the queue is empty
    char* current = queue[pos++]; // pulling the current state from the queue
```

The stack was implemented in my closure function with the sole purpose of traversing through the NFA to perform an epsilon closure on the new state in the stack.



```c
char* stack[100]; //creating a stack data structure for epsilon search
int top = -1;

stack[++top] = temp_name; // adding the given state to the top of the stack
visited[start_id] = true; // setting it to visited

while(top >= 0){ // while loop that stops when stack is empty

    char* current = stack[top--]; //popping the stack
```

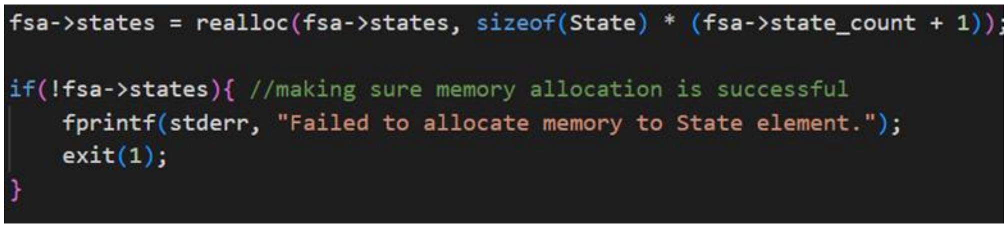
The use of procedures can also be seen in my implementation mainly through the addState and addTransition void functions. Their objective is to take the values of the parameters and add them their respective structs through the main FSA struct. Within these I also have checkers to make sure that there is no repeated states attempting to be created or more than one start state trying to be added to the FSA. Co-ordinating these procedures and integrating them within the code through the use of header files was of great benefit when designing the code which is one of the benefits of C.



```c
void addState(FSA* fsa, char * id, bool start, bool accepting);
void addTransition(FSA* fsa, char * from, char input, char * to);
```

Another procedure used is visible in my utility C file where I implement and in-place quick-sort algorithm avoiding the need to return an array and overusing memory.

However, implementing this FSA in C was no easy task. For my solution I found myself dependent on arrays an awful lot and they are not very compatible with C in my opinion. Trying to allocate memory and not indexing out of bounds was very difficult throughout the process. I had to implement a lot of error handling to avoid too many segmentation faults trying to access a part of memory that does not exist, or the memory core being dropped when trying to free up memory and freeing up pointers to the array rather than the values themselves.



```
fsa->states = realloc(fsa->states, sizeof(State) * (fsa->state_count + 1));

if(!fsa->states){ //making sure memory allocation is successful
    fprintf(stderr, "Failed to allocate memory to State element.");
    exit(1);
}
```

The reason this is a particular issue in C is that you have to almost know a predetermined size for the array compared to other languages where it is automatically allocated as you go along. This issue then leads in to trying to traverse through the array, I had to pass pointers as parameters for functions that were counters so that it could be incremented within the function and I can get the length of the array to later traverse through.

Critique of Object-Oriented Implementation (Java)

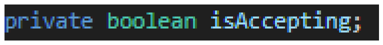
Implementing the Finite State Automata (FSA) in the Object-Oriented Language Java, uncovered many pros and cons for the paradigm in terms of its fundamental principles and mechanisms, constructs, libraries and design patterns.

Object-Oriented Design Patterns

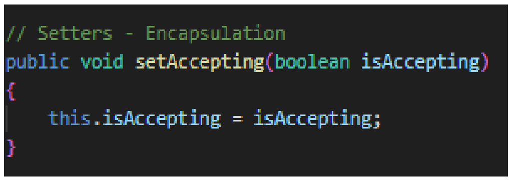
This implementation follows similarly to the OOP **Composite Design Pattern** with slight usage of the **Factory Method** in order to handle State and Transition creation. The Composite Design Pattern intends to compose objects into tree structures, which is exactly what is done in handling our FSA, creating a tree of States being branched to through transitions. These Object-Orientated design patterns made implementation planning easier to understand in the early development stages, which are an attractive aspect of this paradigm's implementation as it allows for better coupling and code reuse.

Principles/Mechanisms

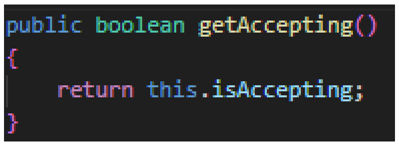
The **OOP design principles** incorporated into this FSA implementation are another enticing reason to implement using this paradigm due to their enforcement of good code practices, reusability and low coupling and high cohesion. **Encapsulation** is utilized to isolate implementation details from what is exposed to the user of the classes, applied to this design through the bundling of attributes and methods that operate on these attributes inside the class, protecting it from outside interference through privatisation of attributes and methods and accessing attributes through "getters" and "setters".



```
private boolean isAccepting;
```

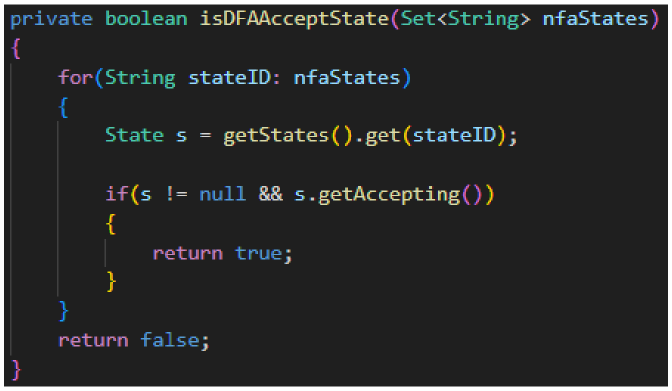


```
// Setters - Encapsulation
public void setAccepting(boolean isAccepting)
{
    this.isAccepting = isAccepting;
}
```



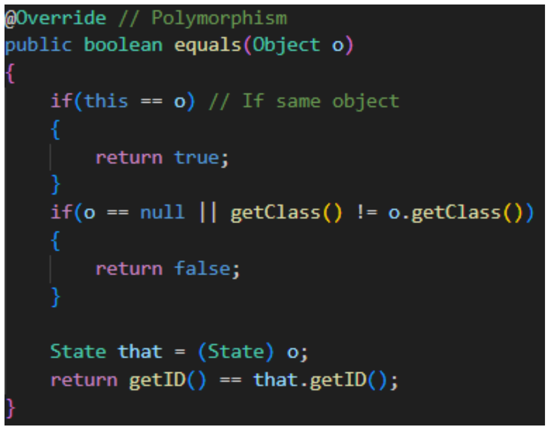
```
public boolean getAccepting()
{
    return this.isAccepting;
}
```

Similarly to encapsulation, **Data Abstraction** is used through “getters” and “setters” while **Process Abstraction** is maintained through hiding internal implementation through private methods.

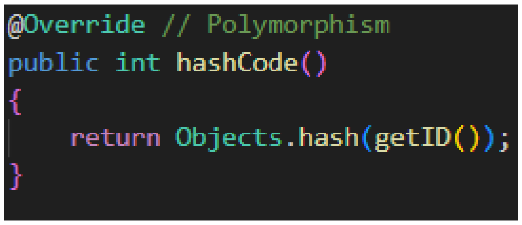


```
private boolean isDFAAcceptState(Set<String> nfaStates)
{
    for(String stateID: nfaStates)
    {
        State s = getStates().get(stateID);

        if(s != null && s.getAccepting())
        {
            return true;
        }
    }
    return false;
}
```

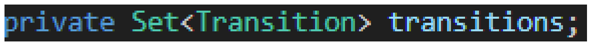
One of the biggest helps for the Java implementation was the use of **Polymorphism** to overwrite how HashCodes and object comparison is handled for developer objects such as State and Transition, allowing them to maintain uniqueness as keys in HashMaps and HashSets, making the handling of the core FSA classes much easier.



```
@Override // Polymorphism
public boolean equals(Object o)
{
    if(this == o) // If same object
    {
        return true;
    }
    if(o == null || getClass() != o.getClass())
    {
        return false;
    }

    State that = (State) o;
    return getID() == that.getID();
}
```

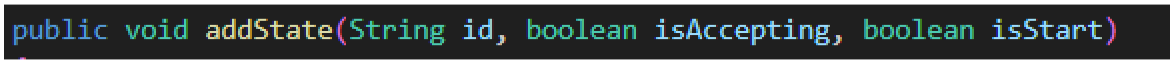


```
@Override // Polymorphism
public int hashCode()
{
    return Objects.hash(getID());
}
```

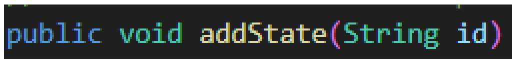


```
private Set<Transition> transitions;
```

Polymorphism also allows for **method overloading** based on parameters given, making the handling of user inputs much easier.



```
public void addState(String id, boolean isAccepting, boolean isStart)
```



```
public void addState(String id)
```

Inheritance was not necessary in this implementation as all classes serve distinct roles, however, another one of the OOP paradigms properties is how easy it is to convert this FSA class to become a generic interface for multiple Automaton types to implement if the system must be expanded in the future.

One of the main other mechanisms that makes the Java implementation so appealing is the **automatic memory management** and **object referencing** that takes place, allowing the creation of many States, Transitions, etc. without manual memory management, as well as automatically doing **parameter passing.** However, despite its simplicity, when working with large FSA’s and specifically in the conversions to a DFA, you lose control over optimizing the processes, leading to worse time and space complexities and runtimes.

Constructs

Java offers many building blocks to design programs, with the main constructs used being **classes** and **objects**. Representing States, Transitions and the FSA as classes made the visualization and understanding of the structure much simpler, by giving each its own attributes and methods which can only be called on instances of the same class.

The large **library** collection Java offers also proposed the ability to utilize them to make the process easier (in the case of my implementation using the Java tuple library), however this arose the complication of the need of files being able to be compiled and ran anywhere, as it can not be assured that these libraries are installed when a user goes to run them.

Comparison of Implementations

Both the procedural implementation (C) and the Object-Oriented implementation (java) of the FSA have widely different benefits to support their implementation, as well as presenting trade-offs stemming from the paradigms of the languages.

For C the implementation of procedural programming concepts like functions, procedures, separate data structures etc offered a more logistical approach to the problem. However, Java's use of OOP concepts allows for a more structured approach to the problem having separate classes and using inheritance and polymorphism to create a better designed and readable code structure. Java also has the added benefit of getters and setters for the OOP design approach which allowed the use of encapsulation to privatise certain attributes or protect them while still being able to access them in other parts of the code. Whereas C is a lot less user-friendly for transporting data around, this can only really be done through parameter passing which became a real limitation in the C programming of our solutions.

C’s implementation also involved a lot of comparative operators imported from libraries such as strcmp() from the string.h library. C does not automatically compare multiple types using comparative operators such as “==” whereas higher level languages are much more suited for this being able to handle different types of comparison through inference of value types. C does also have a huge benefit though of being a lower-level language and this is evident throughout our code from its time complexity. Implementing a solution to this problem involved many for loops and nested for loops creating time complexities as bad as worst-case n^3. A lower-level language such as C is much more efficient in its execution and handling of large time complexities making it seem like less of an issue.

Recommended Implementation

For our final recommendation we decided between us that the best way to go about implementing an FSA was using the procedural programming C. We feel that the principles of procedural programming are much more suited towards a logistical task like this one. Also, with the added benefit of efficiency it can hastily execute the code even when there are large time complexities. A solution to a problem such as this one relies much more in the method of execution rather than the design. C’s use of features such as procedures, functions and structs tackles this problem head on by allowing the programmer to easily convert the theory of this solution into a practical implementation as the code is designed to think in a similar manner as one would go about solving it themselves.

References

[1] D., Lewis. "Is Java Pass-By-Reference or Pass-By-Value?" *Sentry*, 12 July 2022, https://sentry.io/answers/java-pass-by-reference-or-value/

[2] Maheshwari, Anil, et al. *Introduction to Theory of Computation*. 2019, https://cglab.ca/~michiel/TheoryOfComputation/TheoryOfComputation.pdf

[3] Mihai Andronache. "Validating Input with Finite Automata in Java | Baeldung." *Baeldung*, 28 Mar. 2017, www.baeldung.com/java-finite-automata

[4] Sinha, Prince. "Functional vs. Procedural vs. Object-Oriented Programming." *Scoutapm.com*, 2021, www.scoutapm.com/blog/functional-vs-procedural-vs-oop

[5] Kudriavstev, Mikhail. "CSC1031 – Object-Oriented Programming", 2024.

[6] Hamilton, Geoff. "CSC1055 – Comparative Programming Languages", 2025.

[7] Wilkinson, Daniel. "Object-Oriented Principles Explained." *CodeX*, 18 May 2021, https://medium.com/codex/object-oriented-principles-explained-2d1d4bdd3be7

[8] Singhal, Akshay. "Converting NFA to DFA | Solved Examples | Gate Vidyalay." *Gatevidyalay.com*, 28 July 2019, www.gatevidyalay.com/converting-nfa-to-dfa-solved-examples/ . Accessed 30 Oct. 2025

[9] Kumar, Amitesh. "What Is Object-Oriented Programming and Why Is It Useful?" *Emeritus Online Courses*, 15 Feb. 2023, https://emeritus.org/blog/coding-what-is-object-oriented-programming/

[10] Refactoring Guru. "Design Patterns." *Refactoring.guru*, 2014, https://refactoring.guru/design-patterns

[11] "Converting NFA to DFA in Java." *Javathinking*, 16 Oct. 2025, www.javathinking.com/blog/convert-nfa-to-dfa-java/ Accessed 31 Oct. 2025.

[12] Yonay, Ori. "Hashmap Using Tuple as Key." *Stack Overflow*, 10 Aug. 2019, https://stackoverflow.com/questions/57445053/hashmap-using-tuple-as-key

[13] baeldung. "Baeldung." *Baeldung*, 15 Mar. 2017, www.baeldung.com/java-tuples

[14] GeeksforGeeks. "How to Eliminate Duplicate User Defined Objects from LinkedHashSet in Java?" *GeeksforGeeks*, 5 Jan. 2021, www.geeksforgeeks.org/java/how-to-eliminate-duplicate-user-defined-objects-from-linkedhashset-in-java/ Accessed 25 Oct. 2025.

[15] Pankaj. "What Is Abstraction in OOPS? | DigitalOcean." *Www.digitalocean.com*, 3 Aug. 2022, www.digitalocean.com/community/tutorials/what-is-abstraction-in-oops