



The University of Jordan

**School of Engineering
Department of Computer Engineering**

Automatic Translation for Finite State Machine to Java

Supervisor:
Dr. Fahed Jubair

Author(s):

Ammar Abu Yaman

0186436

Firas AlNajjar

0186435

June 4th, 2023

Submitted in partial fulfillment of the requirements of B.Sc. Degree in Computer Engineering

This page is intentionally left blank

ETHICAL STATEMENT

We, the undersigned students, certify and confirm that the work submitted in this project report is entirely our own and has not been copied from any other source. Any material that has been used from other sources has been properly cited and acknowledged in the report.

We are fully aware that any copying or improper citation of references / sources used in this report will be considered plagiarism, which is a clear violation of the Code of Ethics of the University of Jordan.

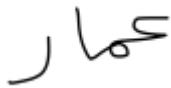
We further certify and confirm that we had no external help without the approval of our supervisor and proper acknowledgment when it is due. We certify and affirm that we never at any point commissioned a 3rd. party to do the work or any part of it on our behalf regardless of the amount charged or lack thereof. We also acknowledge that if suspected and thereafter proven that we commissioned a 3rd. party to do any part of this project that we risk failing the entire project.

We certify and confirm that all results presented in this project are true with no manipulation of data or fraud, that any statistics done, or surveys collected are conducted with the highest degree of scientific fidelity and integrity, and that if proven otherwise, we risk failing the entire project. We acknowledge that for any data collected, we have taken all the steps necessary in applying for proper authorizations if deemed necessary, and that all user data collected is subject to the utmost degrees of privacy and anonymity.

Ammar Abu Yaman

June 4th, 2023

Signature:



Firas AlNajjar

June 4th, 2023

Signature:



This page is intentionally left blank

SUPERVISOR CERTIFICATION

I hereby certify that the students in this project have **successfully finished** their senior year project and by submitting this report they have fulfilled in partial the requirements of B.Sc. Degree in Computer Engineering.

☐

I hereby certify that the students in this project have not completed their senior year graduation project and **I do not approve** that they proceed to the discussion.

☐

I suspect that the students have **violated** one or more of the clauses in the **ethical statement** and I suggest that an investigation committee look into the matter.

☐

Dr. Fahed Jubair

Signature:

This page is intentionally left blank

DEDICATION

This project is dedicated to each person who motivated us, helped us when we needed them and never gave up on us when we were down. This project is dedicated to our families, friends and our teachers who stood by us along this road until this point and will always do.

We thank you all and we will never forget your love and efforts no matter how small your contribution was, we will always be grateful. We must never forget to thank the one who made all this possible, God, for giving us those wonderful, motivational people around us, and for giving us the energy and will to go over the obstacles and difficulties that we encountered in our journey.

SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ANTLR	ANother Tool for Language Recognition
API	Application Program Interface
AST	Abstract Syntax Tree
DNA	DeoxyriboNucleic Acid
DOM	Document Object Model
DSL	Domain Specific Languages
FSM	Finite State Machine
FSMT	FSM Translator
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JAR	Java ARchive
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
SMC	State Machine Compiler
TCP	Transmission Control Protocol
UML	Unified Modelling Lanugage
VSCode	Visual Studio Code
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

ABSTRACT

In software projects, finite state machines (FSM) are used to model and design algorithms and application behavior. They have proven their usefulness in many aspects of software design and unfortunately, they are usually only used in the design phase. In practice FSMs are only used as a reference for implementation, which can look very different to its counterpart where the program logic is scattered around the codebase in many conditional branches and little objects [1] encapsulating different parts of the state. It is often quite disorganized, and the conceptual simplicity of the design's FSM is often lost.

To counter these issues, design patterns such as the State design pattern [2] can be employed to help translate the design directly into code with entities such as state classes and transition functions that help better capture the design in code in an organized manner. However, these patterns usually involve large amounts of boiler plate code and are tedious to implement and have to be changed constantly every time a new state or transition is added to the FSM.

The proposed solution to this problem is to create an automation tool that can be used to design the software and model it by utilizing the concept FSMs. It is mainly a compiler engine that transforms the design that it provided either as a visual or textual representation, directly into code using the State design pattern and automatically generates states, transitions and actions. Accommodating changes in the design can be automatically translated and regenerated to code without needing manual rewrites and updates.

Table of Contents

ETHICAL STATEMENT	III
SUPERVISOR CERTIFICATION	V
DEDICATION.....	VII
SYMBOLS, ABBREVIATIONS, AND ACRONYMS	VIII
ABSTRACT.....	IX
LIST OF FIGURES	XII
CHAPTER 1 INTRODUCTION	1
1.1. Problem Definition.....	1
1.2. Proposed Solution	1
1.3. Project Deliverables	3
1.4. Project Impact.....	3
CHAPTER 2 RELATED WORK.....	5
2.1. Similar Tools in the Same Domain	5
2.1.1. State Machine Compiler (SMC)	5
2.1.2. XState	6
2.2. Similar Tools in Other Domains.....	6
2.2.1. Blockly.....	6
2.2.2. ANother Tool for Language Recognition (ANTLR)	7
CHAPTER 3 SOLUTION DESCRIPTION AND IMPLEMENTATION.....	8
3.1. General Overview	8
3.2. Implementation Details.....	9
3.2.1 FSM Logic and Structure.....	9
3.2.2. Web Application Interface	13
3.2.3. Integration with the Backend.....	15
3.2.4. Console Interface.....	16
3.2.5. JSON Specification Layout	17
3.2.6. FSM4J Language Specification Layout	18
3.2.7. FSM4J Parser	18
3.2.8. JSON Parser	19
3.2.9. The Parser Interface	20
3.2.10. Java Generator	20
3.2.11. The Generator Interface.....	22
CHAPTER 4 RESULTS AND DISCUSSION.....	23

4.1. Run Scenarios.....	23
4.1.1. Bit Sequence Detector	23
4.1.2. DNA Sequence Analyzer	27
4.1.3. Transmission Control Protocol (TCP)	29
4.2. Command Line Run Scenario.....	30
CHAPTER 5 CONCLUSIONS AND FUTURE WORK.....	33
5.1. Conclusions	33
5.2. Future Work.....	34
5.2.1. Visual Simulations.....	34
5.2.2. FSM Validation and Error Handling.....	34
5.2.3. Session Saving.....	35
5.2.4. Asynchronous States	35
5.2.5. New Formats and Languages Support.....	36
APPENDIX A USER MANUAL	41
APPENDIX B PROJECT TIME CHART.....	44
APPENDIX C PRESENTATION SLIDES	45
APPENDIX D USB SOFT COPY.....	57

LIST OF FIGURES

FIGURE 1 – ANTLR WORKFLOW DIAGRAM	7
FIGURE 2 – FSMT ABSTRACT WORKFLOW	8
FIGURE 3 – FSMT ACTUAL WORKFLOW	8
FIGURE 4 – FSM GENERAL STRUCTURE UML DIAGRAM	9
FIGURE 5 – FSM SPECIFIC UML DIAGRAM	13
FIGURE 6 – WEB APPLICATION GUI	14
FIGURE 7 – WORKSPACE STATECHART ELEMENTS	14
FIGURE 8 – DIAGRAM MANIPULATION OPTIONS	14
FIGURE 9 – ELEMENT PROPERTIES PANEL	15
FIGURE 10 – CODE EDITING AREA	15
FIGURE 11 – EXPORT BUTTON	15
FIGURE 12 – FSM4J PARSER FLOWCHART	19
FIGURE 13 – JSON PARSER FLOWCHART	19
FIGURE 14 – FSM PARSERS DESIGN UML DIAGRAM	20
FIGURE 15 – JAVA GENERATED CODE UML DIAGRAM	21
FIGURE 16 – CODE GENERATORS SIMPLIFIED UML DIAGRAM	22
FIGURE 17 – WEB INTERFACE RUN SCENARIOS GUIDE	23
FIGURE 18 – SEQUENCE DETECTOR EXAMPLE: ADDING STATES	24
FIGURE 19 – SEQUENCE DETECTOR EXAMPLE: NAMING STATES	24
FIGURE 20 – SEQUENCE DETECTOR EXAMPLE: CONNECTING STATES	25
FIGURE 21 – SEQUENCE DETECTOR EXAMPLE: CONFIGURING TRANSITIONS	25
FIGURE 22 – SEQUENCE DETECTOR EXAMPLE: COMPLETE DIAGRAM	26
FIGURE 23 – SEQUENCE DETECTOR EXAMPLE: JAVA PROJECT STRUCTURE	26
FIGURE 24 – SEQUENCE DETECTOR EXAMPLE: WRITING APPLICATION CODE	27
FIGURE 25 – SEQUENCE DETECTOR EXAMPLE: RUNNING THE CODE	27
FIGURE 26 – SEQUENCE ANALYZER EXAMPLE: COMPLETE DIAGRAM	28
FIGURE 27 – SEQUENCE ANALYZER EXAMPLE: RUNNING THE CODE	29
FIGURE 28 – TCP EXAMPLE: COMPLETE DIAGRAM	30
FIGURE 29 – CONSOLE INTERFACE EXAMPLE: PROJECT SETUP	32
FIGURE 30 – CONSOLE INTERFACE EXAMPLE: RUNNING CODE	32

LIST OF TABLES

TABLE 1 – SMC AND FSMT COMPARISON	5
TABLE 2 – XSTATE AND FSMT COMPARISON	6

This page is intentionally left blank

CHAPTER 1

INTRODUCTION

1.1. Problem Definition

Nowadays, many businesses are more likely to sacrifice software quality to prioritize launching their products quickly and releasing more valuable features as frequently as possible. Usually, the pressure on developers to deliver functionality due to a deadline coming up results in poorly designed code. These businesses fail to realize that the cost of poor-quality software is higher than the expense of quality itself. The consequences of such poor design could lead to a codebase that is difficult to maintain, update or extend, which in turn leads to increased delays and costs in development. When it comes to larger and more complex projects, further resources and focus should be allocated to the design process, as the magnitude of these consequences naturally amplifies in these kinds of projects. According to an infographic published by APEX Global [3], companies worldwide spend more than 300 billion United States (US) dollars each year on software debugging. Therefore, the design phase in the software engineering process shouldn't be neglected nor undermined, as it provides a blueprint for how the software should be built.

Regardless of the design's quality obvious importance, software developers often tend to jump directly to implementation. This behavior is somewhat normalized and justifiable by supervisors and team leaders inside many companies in the IT industry [4]. Spending significant amount of time and effort on design might feel unnecessary and inefficient when the mentality is set to short-term goals rather than long-term planning. So instead, the majority of developers believe they can simply start writing code and figure out the details as they go along. They might lack the necessary design skills and training, or they may not understand the significance of good design practices and the benefits that come along with them in the long run. In addition, novice developers especially might struggle to interpret a software or algorithm design and could find it difficult and tedious to translate it manually into code patterns, thus resulting in incorrect and inconsistent implementations according to the application requirements. Without sufficient experience and expertise, software design can be an error-prone, time-consuming and resource-intensive process altogether.

1.2. Proposed Solution

The complexity of software design and the time it consumes can be a discouraging factor that draws the developers away from it. The aim of this project is to tackle these problems and make software design a much more appealing process by offering an open-source solution that can easily be adopted and incorporated into software development projects with the main goal of aiding programmers to achieve their assigned tasks in a professional manner.

This idea is inspired by many other prominent tools in the industry that accomplish the same purpose. For example, ANother Tool for Language Recognition (ANTLR) [5] is a tool provides

development automation support through its code generation capabilities. It automatically generates lexer and parser code from grammar specifications, reducing the manual effort required for implementing language recognition. Another great tool is the State Machine Compiler (SMC) [6], which provides automation support for generating efficient state machine code from state machine descriptions, reducing the manual effort needed to implement state machine models. These kinds of tools ensure consistent and efficient results while improving development productivity.

This project utilizes the concept of FSMs in order to achieve the aforementioned target. FSMs can be very useful in software design, but they are unfortunately underutilized in this domain. They act as a clear and structured approach to showcase complex systems and algorithms and visualize their flows, which often require managing and controlling large number of states and transitions between them. They are widely used in various fields, such as engineering and bioinformatics. For instance, in traffic control systems, FSMs can model traffic signal states and govern the timing and sequencing of signal changes. In bioinformatics, they aid in modelling complex biological processes such as gene regulatory networks, allowing researchers to simulate and study their behavior.

All in all, the proposed solution for this project is a general-purpose automation tool that processes a FSM model that represents the abstract view of a software's design, and automatically generates code that inhibits an identical behavior to the model by following a structure similar to that of the State pattern, which is a well-known behavioral design pattern. As a key concept in software engineering which was introduced by the Gang of Four [7], design patterns are frequently employed to address recurrent design issues in software development. Moreover, Java was chosen as the primary generated programming language due to its object-oriented nature [7] which has immense popularity thanks to the ability to create extendible and robust software. By using the proposed tool, developers can focus on higher-level design and problem solving while leaving repetitive and mundane code implementation to the tool.

Furthermore, it is worth mentioning that this tool itself is very well designed in terms of software engineering principles in order to enable future contributors, who would like to help expand the project, to easily extend the code and add new features to it with no to minimal modifications to the original codebase. For example, it is possible to add new target languages and extend the state machine logic without the need to understand internal implementation details of other parts of the codebase.

We also added features to expand on state design pattern and expand the use cases of our solution, these features are

- Default state for fallback transitions.
- Start and exit state methods.
- Guard statements for overloaded transitions.
- Modular design for ease of extension.

1.3. Project Deliverables

This project provides a translation tool called the FSM Translator (FSMT), that is free and available for anyone to use. For developers, there are two interfaces provided to interact with it. The first interface is a web application where the user can easily interact with the frontend through a friendly graphical user interface (GUI) and create FSM models, by specifying states, transitions and actions in a visual drag and drop fashion. When the user is satisfied with the FSM design, he or she can submit the created model to be converted into a portable JavaScript Object Notation (JSON) [8] format then sent to the backend framework, where a compiler engine automatically translates the model and generates the Java classes implemented based on the State design pattern. Finally, the files are compressed and sent back to be downloaded on the user's machine. The web application is deployed publicly and can be accessed by clicking on this [link](#).

The second option available is for those who prefer to define the design model using written specifications. Through the command line, the file describing the design model can be passed directly as input to the compiler engine. The command line contains options that can be configured by the user, such as the format of the file which can be either a custom language, called FSM4J, which is the same name given to the tool's compiler, or a JSON file. The other options are the target language, which can only be Java for now, and the output directory where the generated files will be placed. The command line interface can be used by downloading the project hosted as a public GitHub [9] repository on this [link](#), then creating the compiler artifact that is a Java ARchive (JAR) file using Maven [10], which is a popular project management build tool, then lastly running the JAR file to generate the classes.

Using either interface, developers will be able to integrate the generated code into a desired project workspace while keeping their existing codebase clean and maintainable. As for people who would like to contribute, this project is open for collaboration, and they are welcome to make modifications and add features to the tool as per their needs.

This documentation is provided as well as to help the users understand the tool and guide them on how to use it properly. It is divided as follows: Chapter 2 is a literature review that discusses the previous work that this project is inspired by and based on. Chapter 3 explains how the tool is built and discusses the technical implementation details. Chapter 4 presents the results of the project and demonstrates how to use the tool in different experiments. Chapter 5 summarizes the achieved project deliverables and discusses improvements that can be made in the future.

1.4. Project Impact

Creating software is hard and expensive and takes weeks to months from designing to prototyping to implementation. It is estimated that new software can cost between 25 to 250 thousand US dollars within months of development time. Automation tools that can aid in the design and implementation of software can cut down on development time and cost significantly. According to an online article about workflow automation statistics as of the year 2023 [11], 31% of business leaders agree that automation software reduce labor costs, 40% of productivity is lost due to task-switching, and 70% of the business leaders report that they

spend 45 minutes to three hours on repetitive tasks, from an eight-hour workday. The tool presented definitely falls under the category of workflow automation and would prove to be useful to software development companies from an economic perspective.

From a societal context, this tool can be used in education to demonstrate a system's functionality by modelling its behavior and simulating its actions. The concept of FSMs can help high school students learn how to think systematically, model problems and create solutions for real life examples [12], such as the operation of a vending machine. Many computer-related courses in universities could utilize such a tool, as many courses require FSMs as a fundamental concept in various subjects [13]. For example, in Embedded Systems course, it can be used to design and implement control systems for devices such as sensors, actuators and microcontrollers, where students can interact with the hardware by using the FSM model to respond to events and handle inputs and outputs.

In addition, since it will be available for anyone to use for free, there is a great chance that developers from around the globe will get their hands on it conforming to a paper presented in the 2017 IEEE International Conference on Software Quality, Reliability and Security [14]. From the results of this study, it can be seen that employed instances of the State design pattern have a significant correlation with reusability, flexibility, and understandability. Another conclusion from the said study is that the use intensity of the State design pattern can improve the design quality in terms of reusability and flexibility attributes. Therefore, the proposed tool is recommended to be used with most systems since it generates consistent code that implements the State design pattern.

CHAPTER 2

RELATED WORK

Many of the design decisions in the work presented have been inspired by automation tools that have achieved similar goals. Two of them have been mentioned briefly in the previous chapter. The following sections will dive into more details and mention two more tools that provided a baseline for several ideas in this project.

2.1. Similar Tools in the Same Domain

2.1.1. State Machine Compiler (SMC)

SMC is a tool developed by Charles W. Rapp. It takes in an input file describing a finite state machine and generates well-structured and efficient source code that consists of state pattern classes. It can be considered an improvement on the original project invented by the famous Robert Martin [15], also known as Uncle Bob, who is most recognized for promoting many software design principles that are used today. The idea of the backend engine in this project is mainly influenced by the SMC's concepts and methodologies. Table 1 compares the main differences between SMC and the proposed tool.

The SMC provides many features that contribute to its usability and effectiveness. Its textual representation is constructed in a simple language that offers many features. For example, there are seven different types of transitions that can be easily distinguished from one another, adding a lot of potential personalization and customization details to the functionality of the system. The target language could be chosen from fifteen supported programming languages, such as PHP, Python, C++ and Java. The tool is also extensible, allowing developers to add new code generators or modify existing ones to suit their needs.

Table 1 – SMC and FSMT Comparison

Title	SMC	FSMT
Price	Free	Free
User interface	Console	Console and web
Codebase size	Large	Small
Room for extendibility	Small	Large
Learning curve	Moderate	Mild
State logic	Expansive and complex	Basic and simple
Target languages	15 languages	Java only
Input file formats	SM	JSON, FSM4J
FSM persistence	Supported	Not supported

2.1.2. XState

XState [16] is a powerful JavaScript library for creating FSMs and modelling them in real time. It was created by David Khourshid and designed to be used in both frontend and backend JavaScript applications. The frontend made in this work is basically derived from the idea behind this library and its main features. It introduces the concept of state charts, which are a visual representation of state machines, offering a clear and intuitive way to design and reason about application states and transitions.

This tool aims to make working with state machines easier by offering several features. It ensures type safety and reliability since it uses TypeScript which eliminates runtime errors. It provides a built-in visualization tool that allows seeing state machine in action, including the current state and all possible transitions. It also allows defining interactive state machines, which can be used to create complex user interfaces that respond to user input or any other external events.

On the other hand, integrating XState into existing codebases or frameworks that have their own state management solutions can be challenging. Migrating an application to fully utilize XState may require significant refactoring and careful consideration of the existing architecture.

Table 2 – XState and FSMT Comparison

Title	XState	FSMT
Price	Free	Free
User interface	Web	Console and web
Codebase size	Small	Small
Room for extendibility	None	Large
Learning curve	Mild	Mild
State logic	Basic and Simple	Basic and simple
Target languages	None	Java only
Input file formats	JSON	JSON, FSM4J
FSM persistence	Not supported	Not supported

2.2. Similar Tools in Other Domains

2.2.1. Blockly

Although Blockly [17] has a lot of differences from the technological point of view, it shares many of the goals that the proposed tool strives to replicate and achieve. It is an open-source library that was developed by Google and Massachusetts Institute of Technology (MIT). It provides many features that come with several benefits to the software development industry.

For example, it significantly contributes to software design by offering a visual programming editor that simplifies the development process. Its block-based interface allows users to create programs by dragging and dropping blocks, enabling a visual representation of code logic. This

visual approach enhances understanding and facilitates the design of complex algorithms and workflows. Additionally, its code generation capabilities bridge the gap between visual design and actual implementation, making it a valuable resource for both beginners and experienced developers.

2.2.2. ANother Tool for Language Recognition (ANTLR)

ANTLR is a powerful open-source framework used for building parsers, interpreters, and translators for programming languages and domain-specific languages (DSLs). It is used in this project to generate the parser that is used for translating the custom language that describes FSM model specification, in which details are discussed in the next chapter. Parsers can be quickly defined and refined using an intuitive syntax, which in turn accelerates the development process and facilitates iterative refinement of grammar designs. Fig. 1 illustrates the basic workflow of using ANTLR.

First, the developer defines the language grammar using ANTLR's intuitive syntax. Next, the grammar is compiled, generating parser and lexer classes. When input code is provided, the lexer scans the code, producing a stream of tokens based on defined grammar rules. The parser then analyzes the token stream, constructing an abstract syntax tree (AST) that represents the structure of the code. Developers can manipulate and process the AST, perform error handling and recovery, and generate code or output in various formats. ANTLR's workflow enables efficient and accurate parsing, making it a valuable tool for language recognition and translation tasks.

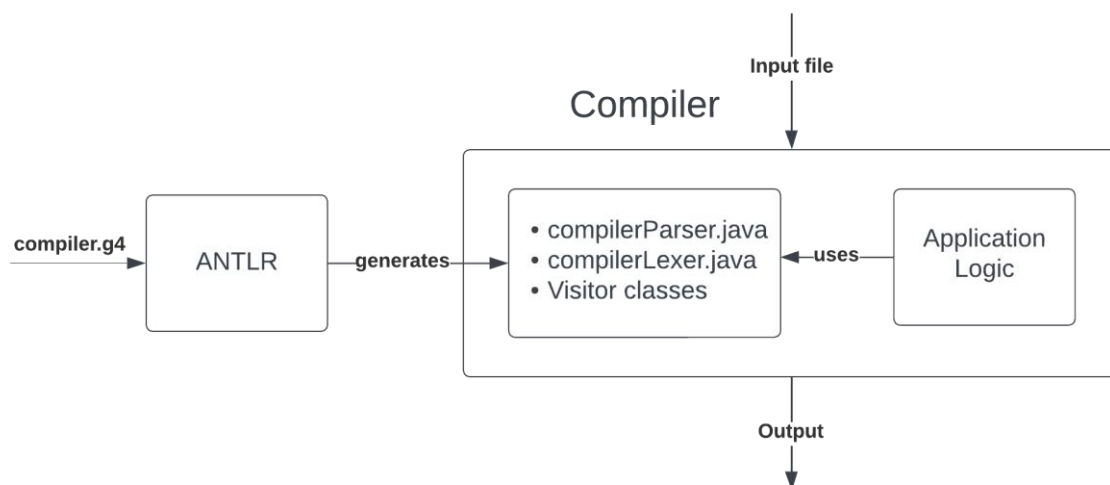


Figure 1 – ANTLR Workflow Diagram

CHAPTER 3

SOLUTION DESCRIPTION AND IMPLEMENTATION

3.1 General Overview

In this project, the translation process goes through several steps. These steps include the interaction between three major components of the system. Each component has its own separate tasks that need to be carried out in order to produce an outcome that meets the user demands. Fig. 2 summarizes the workflow of the translation process in a high-level of abstraction. First, the user interacts with the tool's interface. In turn, the interface passes the user's model specification in a file to be processed by the FSM parser. Then, an FSM object is created based on the details specified. Lastly, the code generator takes in the FSM object and generates source code that represents the FSM structure and attributes.

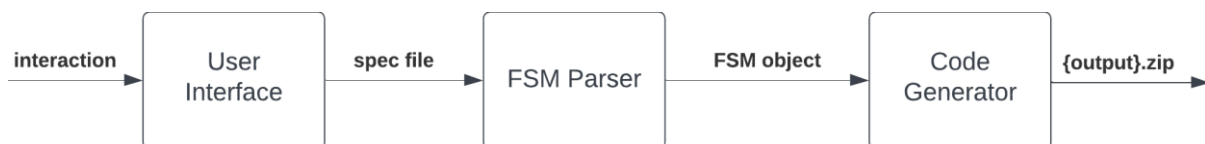


Figure 2 – FSMT Abstract Workflow

Fig. 3 shows a more accurate representation of the current state of the tool. It currently provides two interfaces to interact with. A web application where the user can draw statecharts and submit them, and a console application where the user enters a command with specific arguments. As for the parser, it can process model specification as either a JSON file and or an FSM4J file. The only available generator is the Java generator.

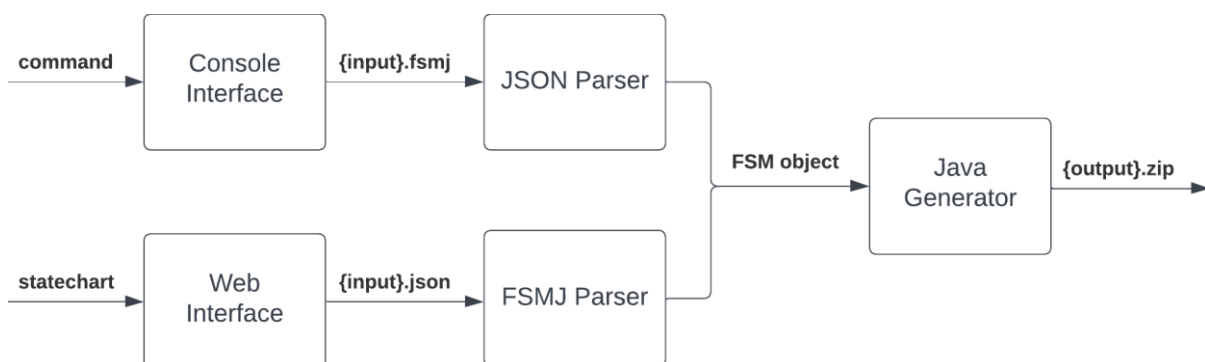


Figure 3 – FSMT Actual Workflow

3.2 Implementation Details

This tool is so straightforward and easy to use, that it is unnecessary to know anything about the code it generates in order to use it. But for the curious, the following subsections dive into the technical implementation details behind the curtain and describe the theory from a software engineering perspective.

3.2.1 FSM Logic and Structure

This section describes the logic behind each FSM model element and how they interact with each other. These elements are aggregated together into one another forming an aggregation stack that ends up with the created model. Fig. 4 illustrates these elements and their relationships using a general unified modelling language (UML) class diagram [18]. An FSM model consists of configuration options and a set of states.

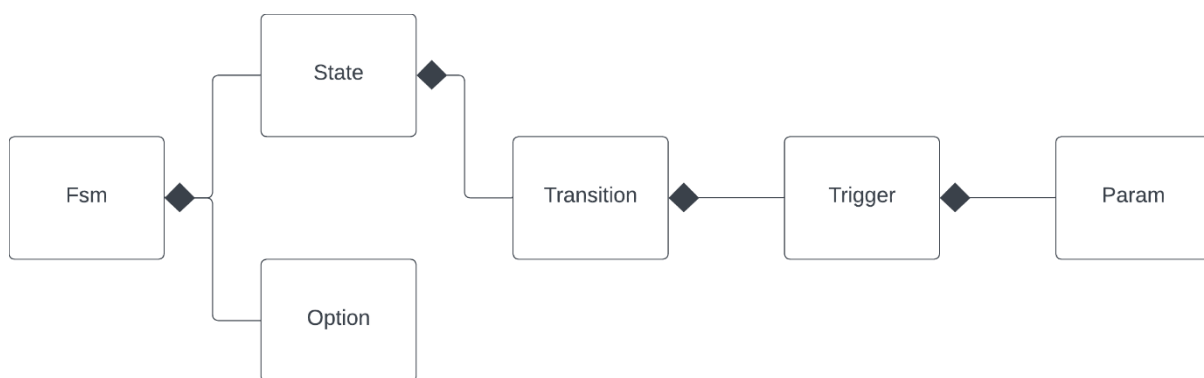


Figure 4 – FSM General Structure UML Diagram

- **Options**

Each option has a name and corresponding values. The currently available options for a valid FSM are:

- 1) Class: specifies the application class to which the FSM is associated.
- 2) Package: specifies to which class package this FSM belongs. This is the same package as the application class.
- 3) Initial state: specifies the FSM's start state.
- 4) Actions: couples between the FSM and the application class by defining a list of void-type methods that the generated application class implements.

Of the four options listed above, only the last one is optional, and the others are required. These options are defined in a constant list which allows new options to be added and existing options to be modified which increases the level of extensibility.

- **States**

As for states, they should mainly have a name. Each state can have its custom entry and exit code, which are executed when entering and leaving the state, respectively. This is useful for cross-cutting concerns, such as a state representing a client connecting to an online resource, where the enter code would initiate the connection, and the enter code would terminate it. Also, each state can have multiple transitions.

- **Transitions**

Each transition consists of the following attributes:

- 1) Trigger: contains the transition name and the list of associated parameters.
- 2) Guard: contains a condition that needs to be satisfied in order for the transition to take place.
- 3) Next state: the next state to transition to.
- 4) Code: statements to be executed on a successful transition. If an action is to be executed, the context variable `ctx` should be prefixed to call the action method.

- **Self-Transitions**

The transition that doesn't switch to a different state is called a loopback transition. There are two types of loopback transitions. An external loopback leaves the current state and comes back to it. This means that the state's exit code and entry code are executed. This transition can be made by setting the next state as the current state's name.

In contrast to an external loopback transition, an internal loopback causes the transition to remain in the current state and not leave it. This means that the state's exit and entry code are not executed. Using `null` as the next state makes this kind of transition.

- **Guards**

When using transition guards and transition parameters, multiple instances of the same transition must have the same argument list. Just like Java methods, for example, the methods `send(String message)` and `send()` are not the same. This is the same case for transitions. Failure to use the identical argument list when defining the same transition with multiple guards will result in incorrect code being generated.

If the guard's condition evaluates to true, then the transition is taken. If the guard condition evaluates to false, then one of the following occurs, ordered by precedence:

- 1) If the state has another guarded transition with the same name and arguments, that transition's guard is checked.
- 2) Failing that, If the state has another unguarded transition with the same name and argument list, that transition is taken.
- 3) If none of the above, then the base transition logic is followed.

A state may have multiple transitions with the same name and parameter list as long as they all have unique guards. When a state does have multiple transitions with the same name, care must

be taken when ordering them. The compiler will check the transitions in the same top-to-bottom order that is used except for the unguarded version. That will always be taken only if all the guarded versions fail. Guard ordering is only important if the guards are not mutually exclusive, meaning it is possible for multiple guards to evaluate to true for the same event.

An example of guards

```
SomeState {
  transition(x: int) [# isOdd(x) #] => StateX {# #},
  transition(x: int) [# isEven(x) #] => StateY {# #},
  transition(x: int) [# isPrime(x) #] => StateZ {# #},
  ...
}
```

- **Exit and Enter Code**

When a transition leaves a state, it executes the state's exit code before any of the transition code. When a transition enters a state, it executes the state's entry code after the transition code. So, the execution order upon a transition is as follows:

- 1) From state's exit code.
- 2) The transition code.
- 3) To state's entry code.

An example of enter and exit actions

```
Connected {
  __enter__ {#
    initiateRemoteConnection(ctx.getId());
    logger.log("Entered Connected State and initiated Connection");
  #}
  __exit__ {#
    terminateRemoteSession();
    logger.log("Terminated session and disconnected");
  #}
  ...
}
```

- **Base States**

What happens if a state receives a transition that is not defined in that state? FSMT provides a mechanism for handling that situation. Every model has a special state named **Base**. The

uppercase **B** is significant. It acts as a safe-guard or a default in a switch statement for states without a specific transition's implementation.

Like all other states, the **Base** state can contain transitions that may have guards and arguments. This means the **Base** state may contain multiple guarded and one unguarded definition for the same transition. The other states can't issue transitions to the **Base** state, otherwise it would ruin the purpose of having it.

An example of a base state

```
Base {  
    transition1 => StateX {#    #},  
    ...  
    transitionN => StateY {#    #}  
}
```

- **Transition's Precedence**

After taking **Base** states into consideration, transition definitions have the following precedence:

- 1) Guarded transition.
- 2) Unguarded transition.
- 3) The **Base** state's guarded definition.
- 4) The **Base** state's unguarded definition.

Since the tool does not force the user to specify a **Base** state, it is possible that there is no transition defined. If FSMT falls through this list, it will throw a runtime exception [19] with the message "Undefined transition".

- **Further Extension**

It is also worth mentioning that all the elements mentioned are represented as data classes in Java to make it easier to distinguish between different elements. All the elements, including the FSM model itself, implement a Java **Item** interface. This type of abstraction introduces some level of decoupling, allowing future contributors to expand the FSM logic easily without interfering with the other items. Fig. 5 summarizes the state machine items, their attributes and relationship using a UML diagram.

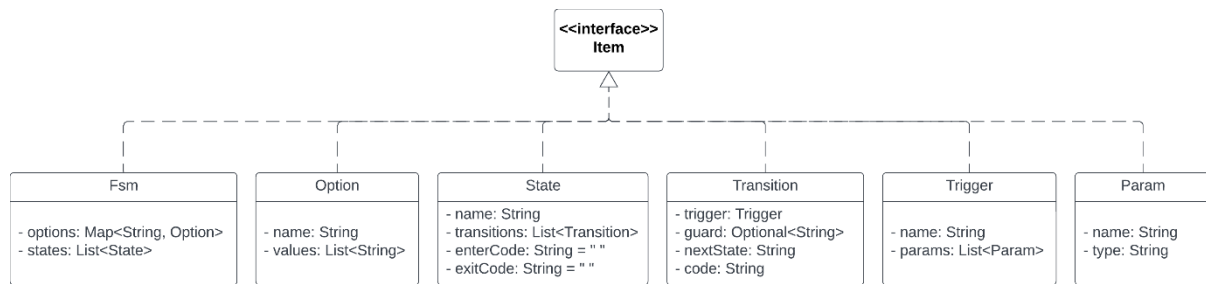


Figure 5 – FSM Specific UML Diagram

3.2.2. Web Application Interface

The web application is built using various tools that simplify the development process. First, the frontend GUI of the website is discussed. The following listings briefly describes each component used to build the GUI:

- 1) TypeScript [20]: the programming language used to write the frontend code. Its main advantage is improved type safety, which helps catch errors during development and enhances code reliability.
- 2) React [21]: serves as the core library for building the user interface of the web application. It provides a component-based architecture and a virtual Document Object Mode (DOM) for optimized rendering and updates.
- 3) Chakra UI [22]: a GUI component library for React. It provides a set of pre-designed, accessible, and customizable components that developers can use to create visually appealing and responsive user interfaces. Chakra UI simplifies the process of styling and layout.
- 4) Redux [23]: a state management library that complements React by providing a predictable state container. It allows developers to manage application state in a centralized manner, making it easier to track changes, implement complex data flows, and handle application-wide state.
- 5) JointJs [24]: a library for building interactive diagramming and visualization features. It can be used to create and manipulate diagrams, flowcharts, and other visual representations. It integrates well with React, allowing developers to incorporate interactive diagrams into their React components and enhance the user experience with visual modeling capabilities.

Fig. 4 represents the GUI users encounter when they visit the website. It has three main parts:

- 1) The workspace area where statechart diagrams can be drawn.
- 2) The properties panel on the righthand side, which allows configuring element settings.
- 3) The header bar on the top. It has a logo on the left, and the “Export” button on the right.



Figure 6 – Web Application GUI

Fig. 5 shows a simple diagram covering the possible elements that can be inserted into the workspace. **S0** and **S1** are states, and **T0** and **T1** are transitions. On the far left is the **Base** state. The little arrow on the upper left corner of **S1** indicates that the state is selected, and by clicking and dragging it, a new transition can be created.

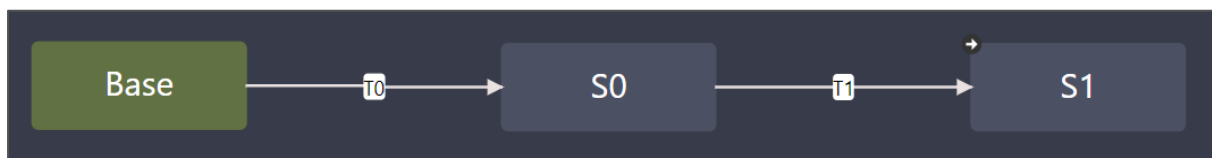


Figure 7 – Workspace Statechart Elements

Fig. 6 illustrates the available workspace options. The state adding options can be accessed by right-clicking on the workspace area. But when right-clicking on any element, the delete option is shown instead. Note that the “Add Base State” option is only shown when there is no current existing **Base** state, as only one **Base** state is allowed in the model.

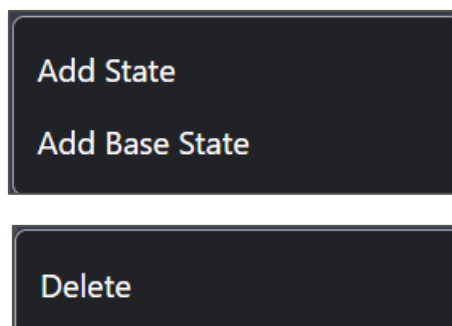


Figure 8 – Diagram Manipulation Options

Fig. 7 highlights the settings for each of the discussed elements. An element can be selected by left clicking on it. Doing that shows its configured settings on the properties panel. The state machine settings, shown on the left of the figure, becomes visible when no element is selected.

Figure 9 – Element Properties Panel

Fig. 8 is the code editing section. This section is revealed when clicking on one of the code editing options, or the condition editing option. The code written here should be written in Java, otherwise it would result in erroneous code in the generated classes.

Figure 10 – Code Editing Area

Fig. 9 is the “Export” button. When clicked, a JSON payload containing the FSM model specification is sent to the backend compiler engine to be processed.

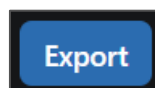


Figure 11 – Export Button

3.2.3. Integration with the Backend

The integration between the frontend GUI and the backend framework is established using a Spring Boot server application [25]. The backend framework refers to the group of the compiler engine components, which are the FSM parser and the code generator.

Spring Boot is an open-source Java-based framework that simplifies the process of building and deploying backend applications. It provides a set of opinionated defaults and auto-configurations, reducing the need for boilerplate code and manual configuration. It enables developers to build the backend logic and expose application programming interfaces (API) [26] that allow the frontend GUI to communicate with the backend.

Developers define the API endpoints. These endpoints act as bridges between the frontend and the backend, allowing data to be sent back and forth. By sending Hypertext Transfer Protocol (HTTP) requests to the specified API endpoints, the backend processes the requests received from the frontend and sends back responses containing the requested data or confirmation of the requested actions. The frontend GUI handles these responses and updates the user interface accordingly.

In this project, there is only one endpoint defined and it is the “/generate” endpoint. When clicking the “Export” button, an HTTP request [27] containing the FSM specification is sent to this endpoint. Upon receiving the request, the controller issues a command to the compiler engine to generate the required files from the specification extracted from the JSON payload. The generated files are then packaged into a zip file along with the runtime library [28] which is necessary for the files to run. Lastly, the controller sends the zip file back to be downloaded by the user’s browser.

3.2.4. Console Interface

Unlike the web application, the console interface allows the user to enter a command directly to the compiler engine. It is built using a Java library called `argparse4j` [29]. This library simplifies the process of handling command-line arguments. The main purpose of using it is to provide the ability to add new arguments easily in one place. This helps maintain a high level of cohesion in the code. Also, it offers valuable features such as automatic generation of help messages, support for positional and optional arguments, and validation of input values.

As mentioned earlier, the compiler application is a JAR file built using Maven. In order to be able to run it, Java has to be installed on the developer’s system. Also, the `java` command has to be accessible from the command line. The following command structure can be used to run the compiler:

```
java -jar /path/fsm4j.jar {spec} {options}
```

The only required argument is `spec`. It is a positional argument which represents the model specification file path. `options` represent a group of optional named arguments which represent the tool’s command line options. The following list shows the available options and their use cases:

- 1) `-f, --format`: specify the format of the specification file. The default value is `fsm4j`.
- 2) `-l, --language`: specify the language of the generated code. The default value is `java`.
- 3) `-o, --output`: specify the output directory of generated files. The default value is `output`.
- 4) `-h, --help`: shows an automatic usage help message and exits without running the tool, overriding any other entered arguments. There is no need to provide a value for this option.

3.2.5. JSON Specification Layout

The layout below shows a sample FSM specification written in JSON. The syntax is self-explanatory, and it can be easily observed that there is a direct correlation between the layout and the FSM data structure discussed previously.

```
{
  "options": {
    "package": ["com.sample"],
    "class": ["Syntax"],
    "initial-state": ["StartState"],
    "actions": ["action1", "action2"]
  },
  "states": [
    {
      "name": "StartState",
      "enterCode": "// StartState enter code executes",
      "exitCode": "// StartState enter code executes ",
      "transitions": [
        {
          "trigger": {
            "name": "normalTransition",
            "params": [
              {
                "name": "param",
                "type": "Type"
              }
            ]
          },
          "guard": "param.condition()",
          "nextState": "NextState",
          "code": "ctx.action1(); // executes action1 in transition code"
        },
        {
          "trigger": {
            "name": "innerLoopbackTransition",
            "params": []
          },
          "nextState": "null",
          "code": ""
        }
      ]
    },
    {
      "name": "NextState",
      "enterCode": "",
      "exitCode": "",
      "transitions": []
    }
  ]
}
```

```
]
}
```

The main drawback of using the JSON format for writing specification manually is that for more complex FSMs, the number of lines can grow to a point where it would become cumbersome to understand the model and connect the dots between the different elements.

On the other hand, the support for JSON files is particularly useful. This is due to the availability of the user-friendly GUI that enables the automatic generation and translation of the JSON file, which is less error-prone and can prove to be a better option for new users.

3.2.6. FSM4J Language Specification Layout

Shown below is the same sample used in the previous subsection, but here it is written using the FSM4J language. The number of lines is much lower using this language. This simple syntax proves to be a more preferable option for developers that like to write FSM specification manually and pass it as input directly to the compiler engine.

```
$package com.sample
$class Syntax
$initial-state StartState
$actions action1 action2

${{
    StartState {
        __enter__ {#
            // StartState enter code executes
        #}
        __exit__ {#
            // StartState exit code executes
        #}
        normalTransition(param:Type) [#param.condition()#] => NextState {#
            ctx.action1(); // executes action1 in transition code
        #},
        innerLoopbackTransition => null {##}
    },
    NextState {}
}}$
```

3.2.7. FSM4J Parser

Instead of building a parser for the FSM4J language from scratch, this process is simplified by using ANTLR. The syntax and rules of the custom language can be specified by writing an ANTLR grammar description. Using ANTLR's plugins, several files were generated, including the lexer and the parser and the default visitor classes. The visitor classes have methods that are called during the parsing process. These methods are overridden in a standalone visitor

class, where actions to be taken when the parser encounters different language constructs are defined. This is necessary for building the required FSM object. These steps are illustrated in Fig. 1.

Now that the setup is complete, these building blocks can be combined in order to implement the custom language parsing logic. First, the lexer and the parser objects are instantiated. The lexer converts the input character stream in tokens to be then parsed by the parser, generating a parse tree. Lastly, the defined visitor actions are performed on the parse tree to create and populate the FSM object. Fig. 12 summarizes the described process in a simplified manner.



Figure 12 – FSM4J Parser Flowchart

3.2.8. JSON Parser

Unlike the FSM4J parser, the implementation of the JSON parser is much simpler. It utilizes a popular open-source Java library for JSON processing called Jackson [30]. This library is commonly employed in web services, RESTful APIs, and other applications that involve JSON data manipulation.

Instead of manually defining a grammar description for JSON and using ANTLR to generate the parser, this task is delegated to Jackson's parsing APIs. The values can be easily extracted from the JSON and mapped to the corresponding Java object. The ObjectMapper class, which is provided by the Jackson library, is responsible for the JSON parsing and mapping process.

Fig. 13 shows a simple flowchart diagram summarizing the steps taken. First, an ObjectMapper object is instantiated. Then, the JSON input is parsed into a tree-like structure represented by a JsonNode object. This object represents the JSON data in a hierarchical format. By navigating through the structure, values can be easily extracted and mapped into a newly created FSM object without the need to define specific Java classes for mapping.



Figure 13 – JSON Parser Flowchart

3.2.9. The Parser Interface

One would think that it would be difficult to incorporate new types of parsers into the project, since other parts of the project would need to be modified to accommodate the addition. However, that's not the case, as the parsers are designed using the Strategy [31] design pattern. This pattern complies with the open-closed principle. It is an object-oriented programming principle which states that software entities should be open for extension but closed for modification.

Moreover, this pattern is particularly useful when there are a lot of similar classes that only differ in the way they execute some behavior. This applies for parsers as they only differ in their parsing behavior. Using the concept of polymorphism [32], this pattern isolates the implementation details of parsing from the code that uses it and makes it totally unnecessary for the compiler to know the type of parser to use prior to runtime. This enables it to switch from one parser to another during runtime by specifying the format in the command line options.

Fig. 14 shows a UML class diagram for the parsing behavior. It also demonstrates an example of adding a new extensible markup language (XML) [33] parser to the compiler. It can be observed that the parse method can be executed in the App class without knowing any implementation details of the parsers, thus doesn't get affected by the addition. Consequently, this enables future contributors to add new parsers without needing to modify other parts of the codebase.

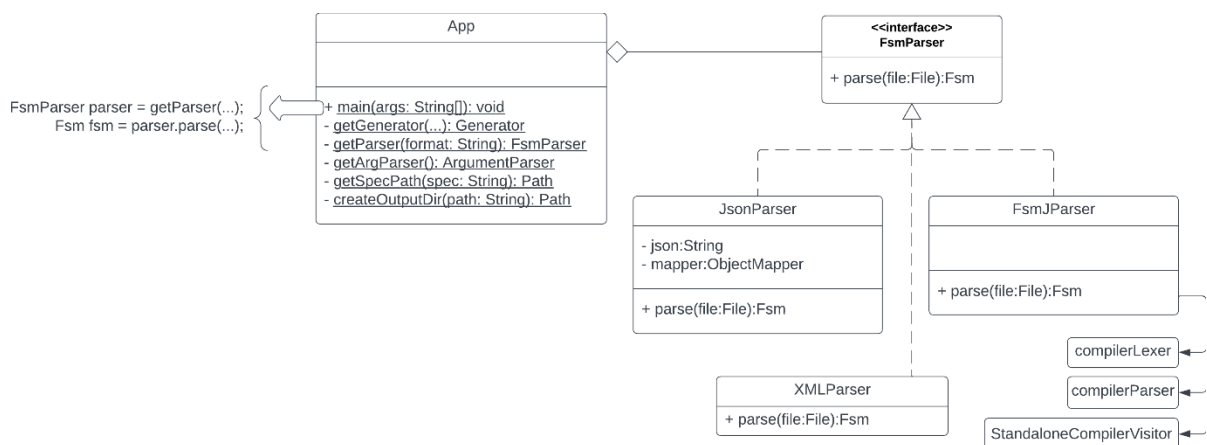


Figure 14 – FSM Parsers Design UML Diagram

3.2.10. Java Generator

The code generated using this generator follows a structure very similar to the one adopted by the State design pattern which was discussed previously. FSM4J deviates from the State pattern in several ways. Fig. 15 describes the relationships between the generated classes using a UML class diagram, assuming the application class name is set to App.

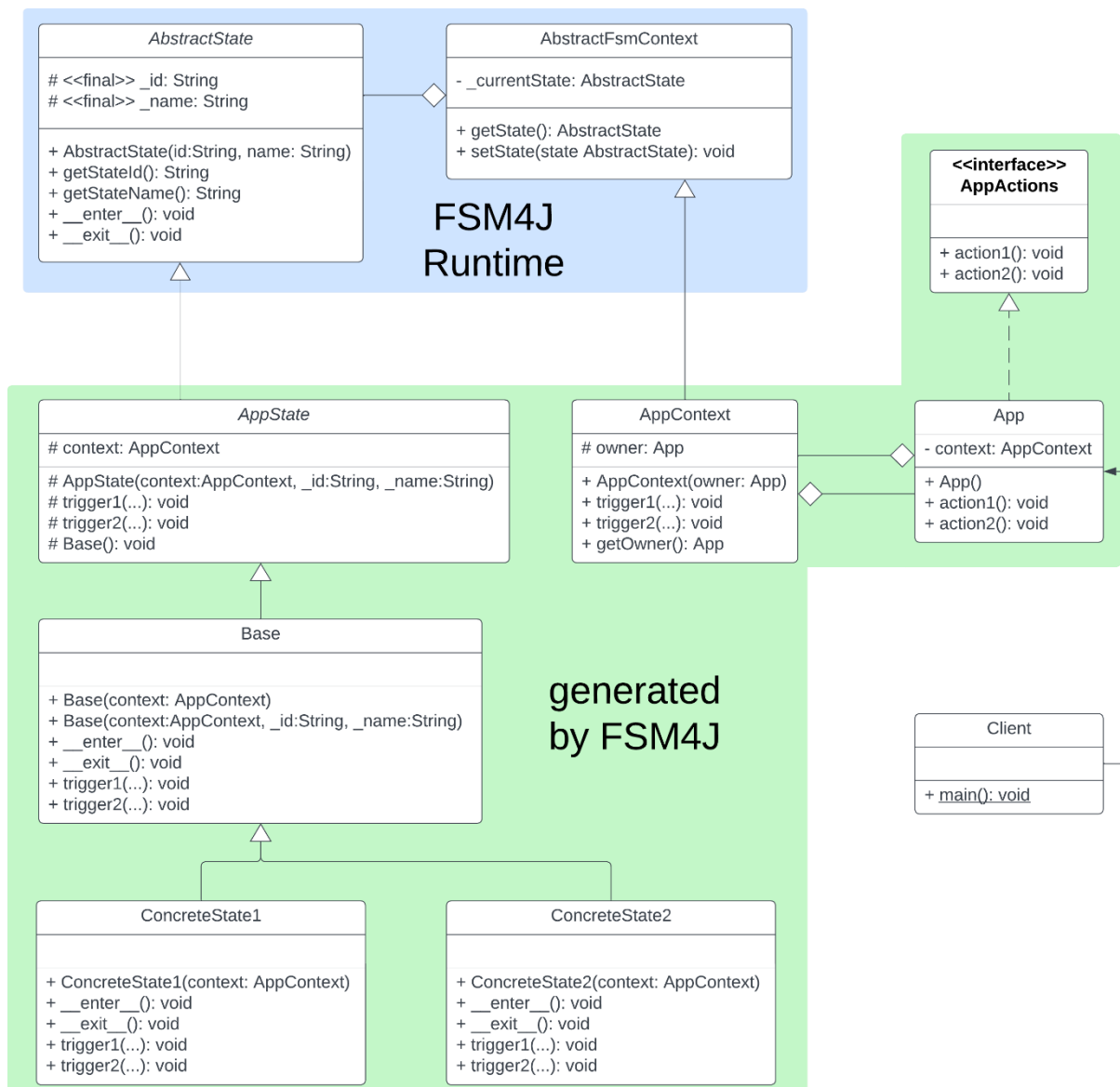


Figure 15 – Java Generated Code UML Diagram

The State pattern's Context class was broken into two classes: an **AbstractFSMContext** class, which is not generated but provided with the compiler, and **AppContext** class. **AbstractFSMContext** stores the current state. It also defines methods for setting and getting the state.

AppContext inherits from **AbstractFSMContext**, and downcasts the `getState` method to return the current state as an **AppState** object and not an **AbstractState** object - that is why this method is not in **AbstractFSMContext**. It also provides access to all transitions defined in all machines and maintains a reference back to its owner **App** object.

The state pattern has an abstract **State** class and concrete state classes which inherit from **State**. FSMT expands this hierarchy to four levels: **AbstractState**, **AppState**, the **Base** state, and concrete states.

This hierarchy is used to support FSMT's base transitions. **AppState** has a virtual method for each transition appearing in all state machines. These transition methods call **AppState**'s **Base** transition method. This global **Base** transition throws an "Undefined transition" runtime exception when called.

The **Base** state class contains the base state's transitions. Each concrete state is a class which inherits from the **Base** class. The concrete state methods implement state machine transitions.

While FSMT generates many classes, they take up little run time space. There is only one instance of each concrete state class. Only one **AppContext** class need be instantiated for each **App** class instance.

The FSMT State pattern logic is hidden from the **App** class. This class only has reference to the **AppContext** object of the FSM. It can be used to call the transition methods. A developer can define the methods he wants using the available transitions, without needing to know the design of the FSM. Finally, it's only a matter of instantiating the **App** class and calling the defined methods.

3.2.11. The Generator Interface

The generating behavior is design is identical to the parsing behavior and uses the Strategy design pattern. This enables contributors to easily implement the generator interface and add concrete generator classes for different programming languages. Fig. 16 shows a simplified UML of the Strategy pattern usage in code generation behavior. The figure also demonstrates an example of Python generating feature to the compiler.

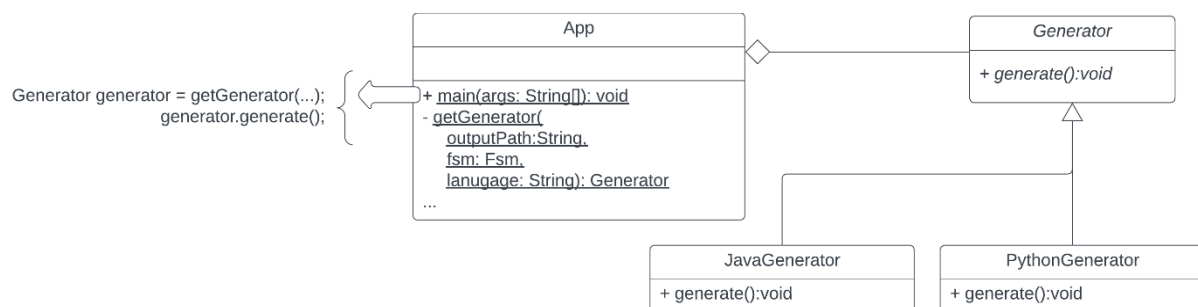


Figure 16 – Code Generators Simplified UML Diagram

CHAPTER 4

RESULTS AND DISCUSSION

As mentioned previously, the proposed tool has a lot of potential to be useful in many applications. This chapter presents the results of using the tool, validating the theory and implementation discussed in the previous chapter. It describes how to use it properly, and how to make use of the generated code in software projects.

In the following sections, the results of applying the translation procedure on three different experiments will be shown. All the examples are illustrated using the web application interface, whereas only one of them is demonstrated using the console interface for the purpose of reducing redundancy.

4.1. Run Scenarios

Opening the web application will create an empty workspace, as shown previously in Fig. 6. Fig. 17 shows the steps involved in the following guide. It walks through the process of drawing the mentioned diagram using the web interface. Then, the generated code is incorporated in a new project and running it using the input string written above.

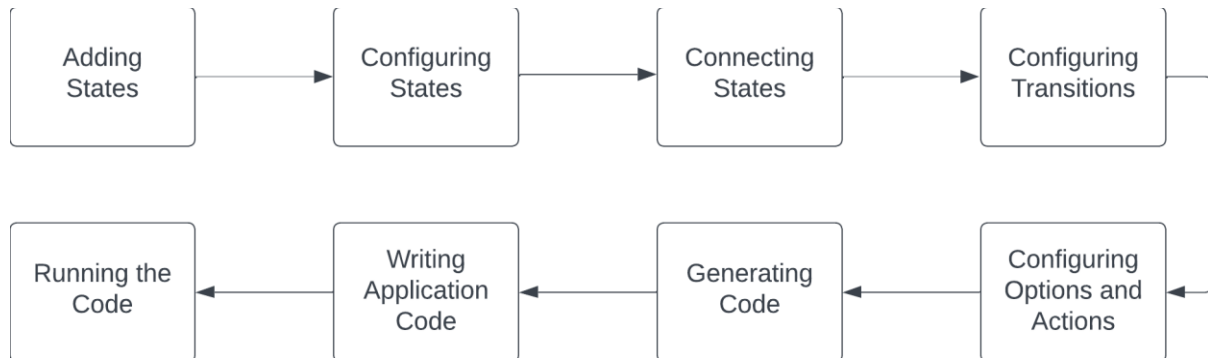


Figure 17 – Web Interface Run Scenarios Guide

4.1.1. Bit Sequence Detector

A Sequence detector [34] is a good example to describe FSMs. It produces a pulse output whenever it detects a predefined sequence. In this tutorial, a 4-bit sequence “1010” has been considered. Sequence detectors can be of two types: with overlapping and without overlapping. For example, consider the input sequence as “11010101011”. Then in the ‘without overlapping’ style, the output will be “00001000100”, and the output in the ‘with overlapping’ style will be “00001010100”. The ‘with overlapping’ style considers the overlapping sequences. The ‘without overlapping’ style is considered in the tutorial.

1) Adding States

By right clicking on the empty workspace, the options from Fig. 8 to add a new state appear. Fig. 17 shows the diagram after adding the states with their default name “State”. Four states are added, which is the same number of the sequence bits. Note that the diagram can be zoomed in and out by scrolling the mouse wheel.

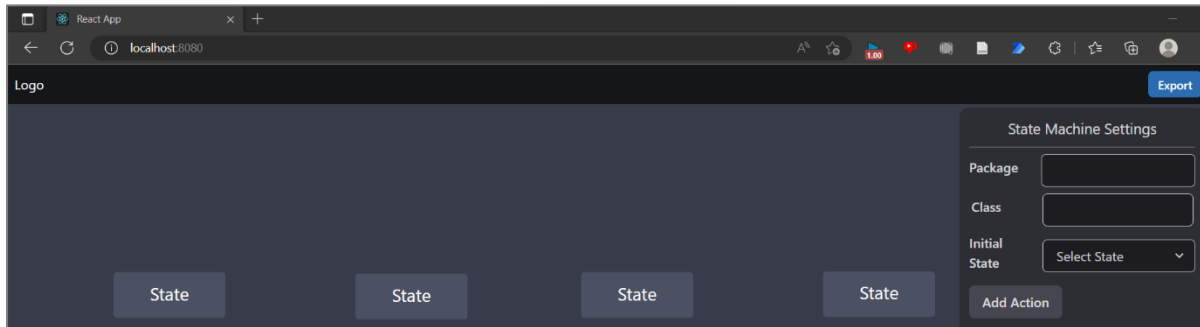


Figure 18 – Sequence Detector Example: Adding States

2) Naming States

The names of the states can be edited in the “State Settings” panel that is shown when left clicking on the state. Fig. 18 shows the diagram after this step is done. The states names are indicated by a number which represents the number of sequential bits matching the corresponding part of the sequence.

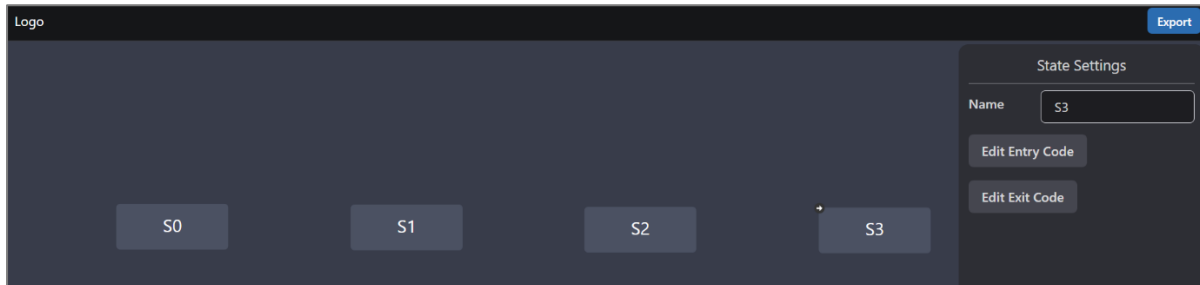


Figure 19 – Sequence Detector Example: Naming States

3) Connecting States

The logic of matching the bits with the sequence can be implemented by first connecting the states using transitions. The transition can be added by clicking and dragging the arrow from the top left corner of the state in the outward direction and connecting it to the desired next state. This arrow can be seen in Fig. 7 and is also shown in the previous figure where the state S3 is selected. Fig. 19 shows the diagram after connecting the states. Note that the trigger names are set to “Transition” by default.

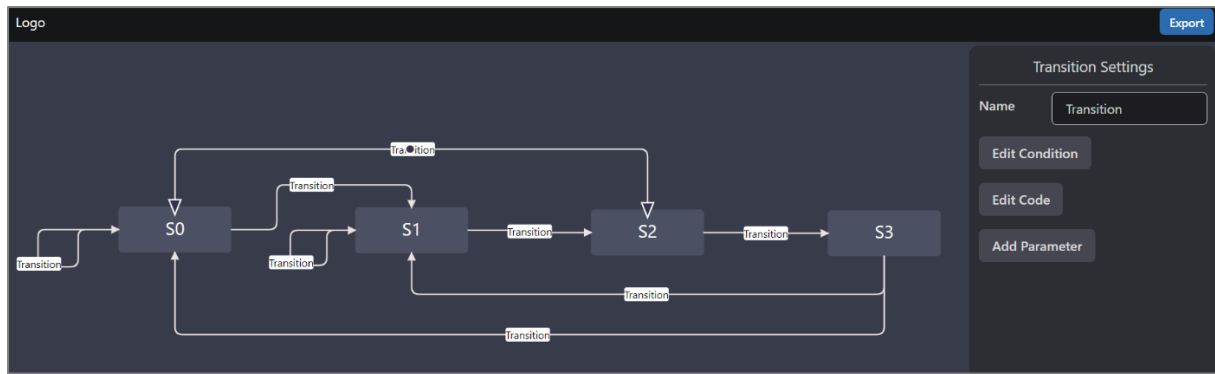


Figure 20 – Sequence Detector Example: Connecting States

4) Configuring Transitions

This step determines the input parameter the transition takes, and the condition that needs to be satisfied for it to take place. The transition code will be left for the fifth step. All of these properties can be configured in the “Transition Settings” panel shown when selecting a transition. Fig. 20 shows the diagram after specifying the transition properties. Each state has a transition called `readBit` that has two possible next states determined based on the input bit represented by an integer `b`. The detector only transitions forward if the input bit matches the corresponding sequence bit. Otherwise, it either stays in the same state or goes back to the closest previous state that has a valid sequence part that can be resumed.

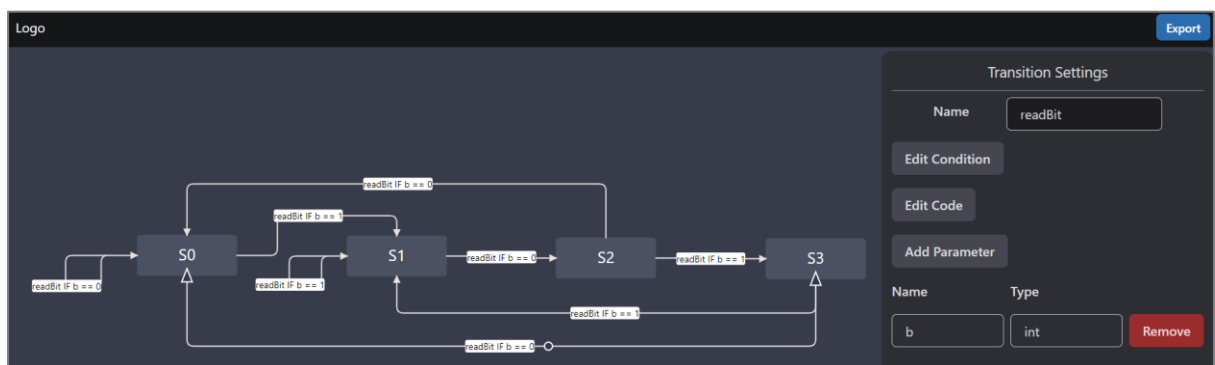


Figure 21 – Sequence Detector Example: Configuring Transitions

5) Configuring Options and Actions

The last step in the FSM design is to specify state options and actions. The initial state is set to S0 because there is no input to check yet, and the application class name is `SequenceDetector`. There are two actions defined: `match` and `mismatch`. The behavior of these actions will be implemented. These actions will get executed on each successful transition. Therefore, the transition code will be set to either `ctx.match();` if the sequence is detected, or `ctx.mismatch();` if the sequence is not detected. Fig. 21 shows the complete FSM diagram. Note that the `match` is only set to the transition from S3 to S0, so the detector starts over after each completion.

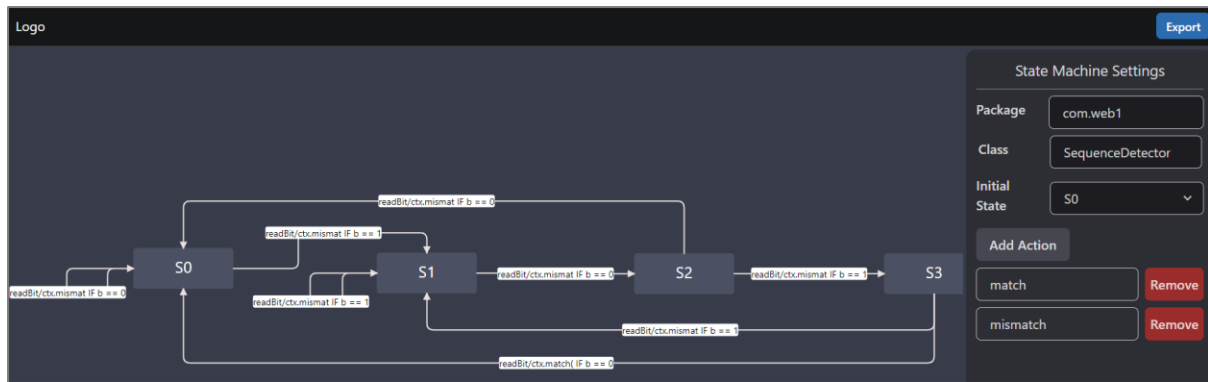


Figure 22 – Sequence Detector Example: Complete Diagram

6) Generating Code

Now that the diagram is complete, the code is ready to be downloaded. Upon clicking the blue “Export” button on the right top corner, a SequenceDetector.zip file downloaded, which the class files can be extracted from, including the runtime library as well. Fig. 22 illustrates the files in a new Java project created using Visual Studio Code (VSCoDe) [35]. The files are put in the same package “com.web1” specified earlier in the options.

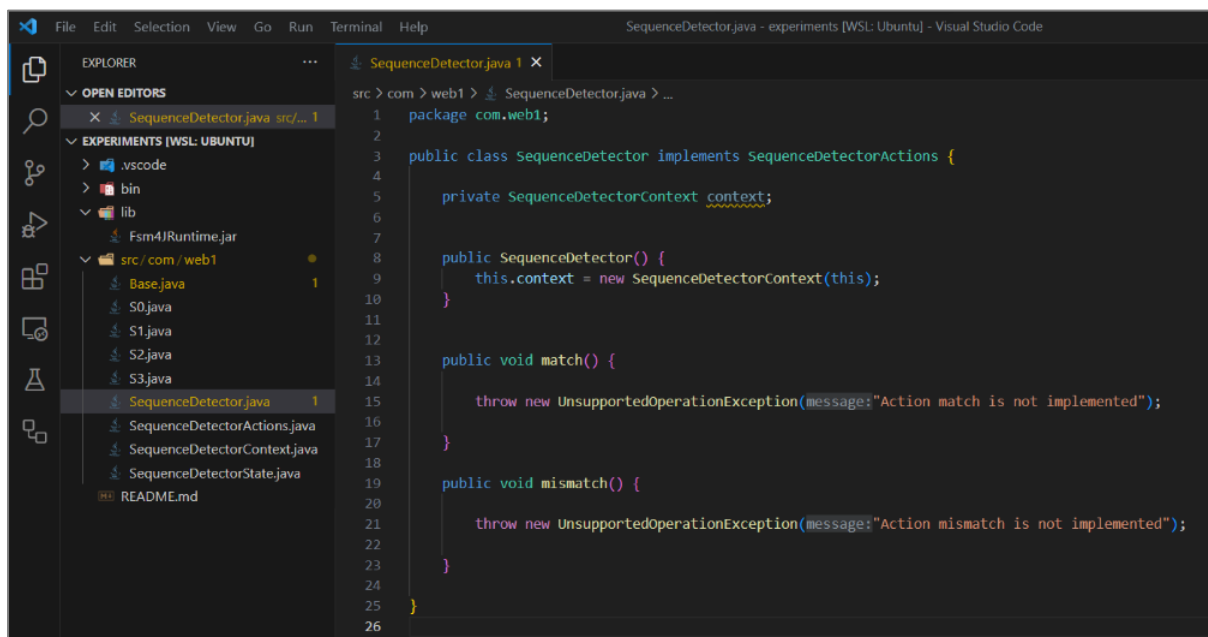
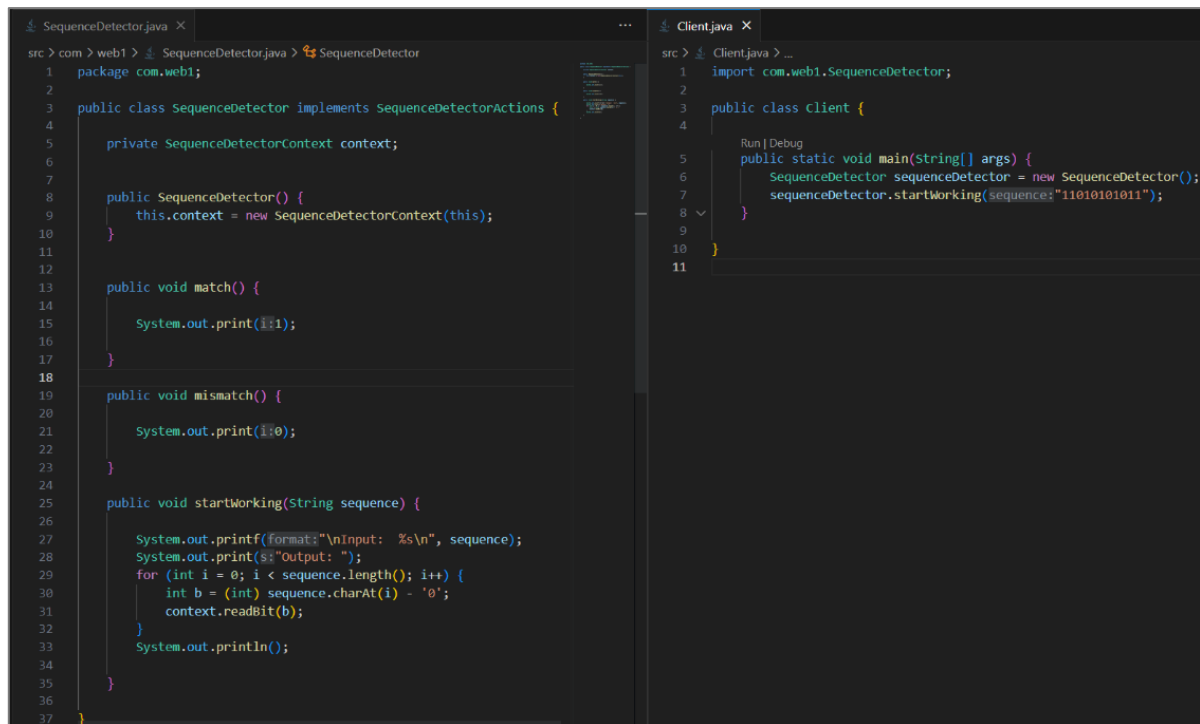


Figure 23 – Sequence Detector Example: Java Project Structure

7) Writing Application Code

It can be seen in the previous figure that the application class is opened in the editor. This class can be modified to include custom behavior implementation based on the user’s needs. This includes implementing the actions as well. Fig. 23 shows both the client code, represented in a new file “Client.java”, and the sequence detector behavior implementation, side by side. The “match” action is edited to print out the number one, and “mismatch” prints the number zero.

A new method is added which takes the input sequence and executes a transition on a bit-by-bit basis. The client code simply instantiates the sequence detector and runs the method.



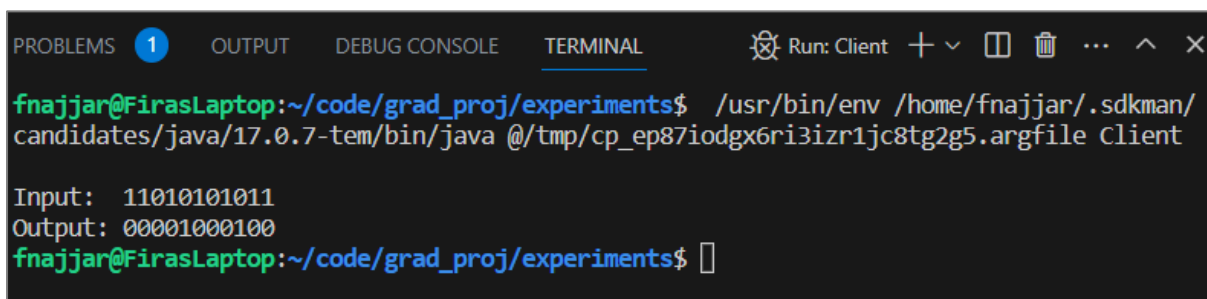
```
SequenceDetector.java
src > com > web1 > SequenceDetector.java > SequenceDetector
1 package com.web1;
2
3 public class SequenceDetector implements SequenceDetectorActions {
4
5     private SequenceDetectorContext context;
6
7
8     public SequenceDetector() {
9         this.context = new SequenceDetectorContext(this);
10    }
11
12
13    public void match() {
14
15        System.out.print(1);
16    }
17
18
19    public void mismatch() {
20
21        System.out.print(0);
22    }
23
24
25    public void startWorking(String sequence) {
26
27        System.out.printf(format: "%s\n", sequence);
28        System.out.print("$"output: ");
29        for (int i = 0; i < sequence.length(); i++) {
30            int b = (int) sequence.charAt(i) - '0';
31            context.readBit(b);
32        }
33        System.out.println();
34    }
35
36
37 }

Client.java
src > Client.java > ...
1 import com.web1.SequenceDetector;
2
3 public class Client {
4
5     Run | Debug
6     public static void main(String[] args) {
7         SequenceDetector sequenceDetector = new SequenceDetector();
8         sequenceDetector.startWorking(sequence; "11010101011");
9     }
10
11 }
```

Figure 24 – Sequence Detector Example: Writing Application Code

8) Running the Code

The code can be run in VSCode by clicking on the Run option above the main method in the Client class. Fig. 24 shows the resulting output in the integrated terminal. The same input sequence that is mentioned in the introduction of this subsection is used in code. The outputs of both are the exact same which indicates that the generated State pattern code works as intended.



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL Run: Client
fnajjar@FirasLaptop:~/code/grad_proj/experiments$ /usr/bin/env /home/fnajjar/.sdkman/candidates/java/17.0.7-tem/bin/java @/tmp/cp_ep87iodgx6ri3izr1jc8tg2g5.argfile Client
Input: 11010101011
Output: 00001000100
fnajjar@FirasLaptop:~/code/grad_proj/experiments$
```

Figure 25 – Sequence Detector Example: Running the Code

4.1.2. DNA Sequence Analyzer

Deoxyribonucleic acid (DNA) [36] is a molecule that carries the genetic instructions for the development, functioning, growth, and reproduction of all living organisms. It consists of a sequence of nucleotides, which are the building blocks of the DNA molecule. The DNA sequence refers to the specific arrangement of these nucleotides along the DNA molecule. The

four types of nucleotides found in DNA are adenine (A), thymine (T), cytosine (C), and guanine (G). These nucleotides form pairs—adenine with thymine, and cytosine with guanine—through hydrogen bonds. The sequence of these nucleotide pairs constitutes the genetic code that carries the instructions for the synthesis of proteins and the regulation of various biological processes.

In bioinformatics, DNA sequence analysis [37] is a process of studying and interpreting the information encoded within a DNA sequence. It involves various computational and statistical techniques to extract meaningful insights from the genetic code. The analysis can include tasks such as identifying genes, predicting protein structures, studying evolutionary relationships, and discovering potential genetic variations.

Fig. 24 shows a complete FSM design for a sequence analyzer for identifying genes. The sequence to be analyzed is “TAGC”. It looks really similar to the sequence detector in design but differs in the application field. Starting from the “Start” state, each transition will check for a match in the DNA sequence. For each nucleotide in the subject DNA that matches the corresponding nucleotide in the specified sequence, a transition into the next state takes place. Any mismatch would cause a return into the “Start” state. For a complete match, the “found” action will execute.

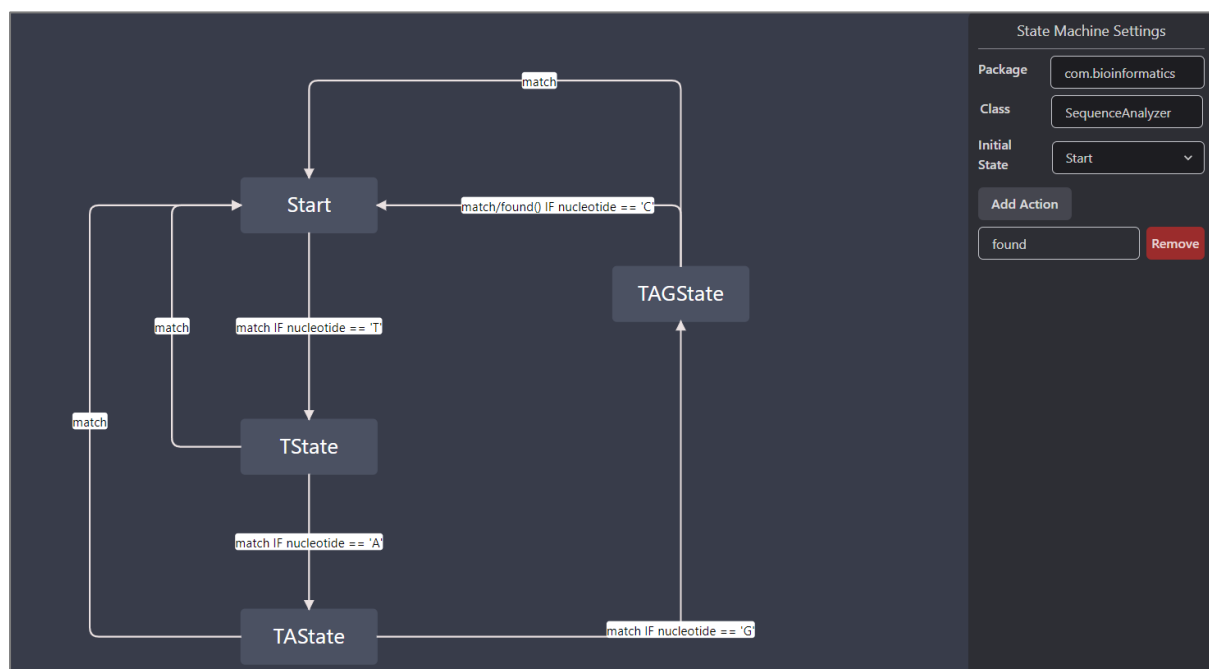


Figure 26 – Sequence Analyzer Example: Complete Diagram

Then after the design is complete, the files are downloaded and are incorporated into a project. Fig. 26 shows the generated files, the main code, and the resulting output. It can be noticed that only one sequence is detected, which starts at the 6th nucleotide.

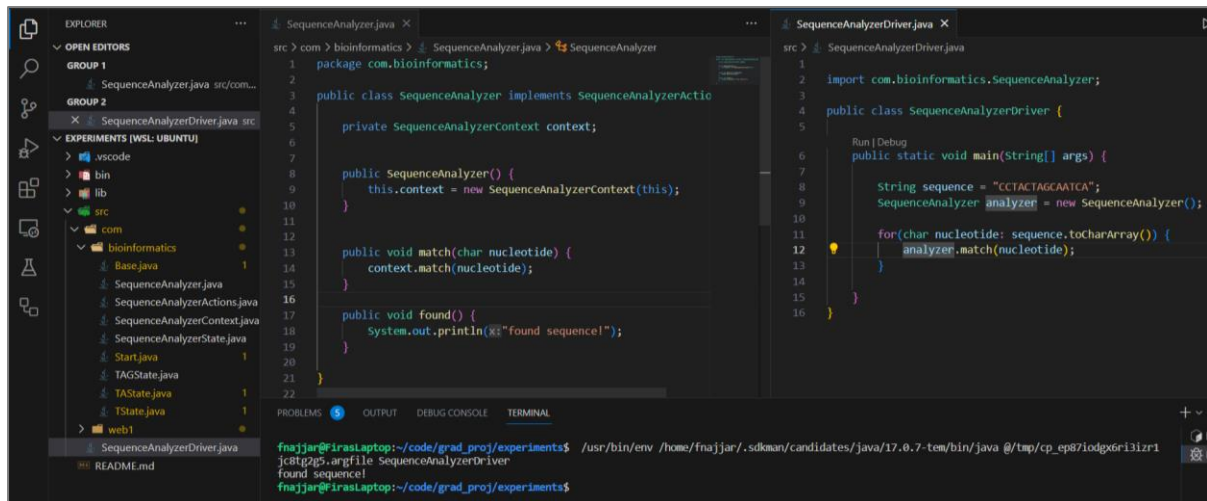


Figure 27 – Sequence Analyzer Example: Running the Code

4.1.3. Transmission Control Protocol (TCP)

TCP [38] is a core communication protocol of the Internet Protocol (IP) suite used in the field of computer networking. It is responsible for establishing reliable, connection-oriented communication between devices over IP networks. It handles data flow control, error detection, and recovery mechanisms to guarantee the reliable delivery of data packets. TCP plays a vital role in supporting various applications, such as web browsing, email, file transfer, and any other application that requires reliable data transmission.

Fig. 27 is a complete FSM diagram that describes the process that TCP undertakes in a simplified manner. The steps involved can be grouped into two phases indicated by numbers on the figure, and can be summarized as the following:

- 1) Connection establishment: The client initiates a connection by sending a SYN segment. The server responds with a SYN-ACK segment, acknowledging the request. The client then sends an ACK segment to acknowledge the server's response, establishing the connection. After this point, data transfer and reception can happen. The details of which are not discussed in this example.
- 2) Connection termination: When data transfer is complete, the connection is terminated. The initiating device sends a FIN segment to request termination. The receiving device acknowledges the request with an ACK segment. If the receiving device has additional data to transmit, it can send its own FIN segment. Finally, the initiating device acknowledges the FIN segment, and the connection is closed.

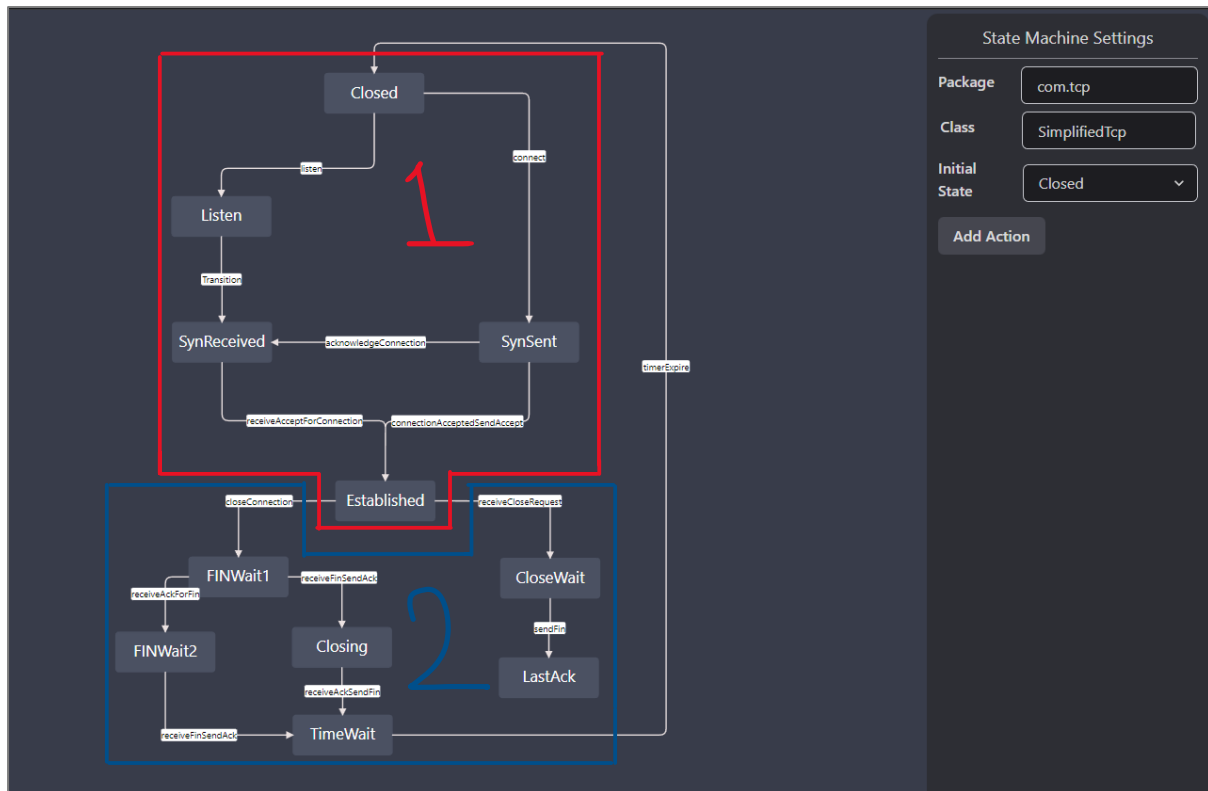


Figure 28 – TCP Example: Complete Diagram

Lastly, code generation is proceeded, and the classes are incorporated into a new project the same way it was done in the previous two examples. It is not illustrated in this documentation to avoid repetition, and it also requires a more complex implementation code for the process to be replicated in a comprehensive manner. The generated code is provided in the GitHub repo, along with the code for the other two examples.

4.2. Command Line Run Scenario

Shown below is the specification file for the same sequence detector example discussed previously. The specification is written using the custom language since it is easier to implement the FSM in it.

```
$package com.console

$class SequenceDetector

$initial-state S0

$actions
    match
    mismatch

${{
    S0 {
        readBit(b:int) [#b == 0#] => S0 {#
```

```

        ctx.mismatch();
    #},
    readBit(b:int) [#b == 1#] => S1 {#
        ctx.mismatch();
    #}
},

S1 {
    readBit(b:int) [#b == 0#] => S2 {#
        ctx.mismatch();
    #},
    readBit(b:int) [#b == 1#] => S1 {#
        ctx.mismatch();
    #}
},

S2 {
    readBit(b:int) [#b == 0#] => S0 {#
        ctx.mismatch();
    #},
    readBit(b:int) [#b == 1#] => S3 {#
        ctx.mismatch();
    #}
},

S3 {
    readBit(b:int) [#b == 0#] => S0 {#
        ctx.match();
    #},
    readBit(b:int) [#b == 1#] => S1 {#
        ctx.mismatch();
    #}
}
}}$

```

Fig. 28 shows a project directory the needed file to generate the code: specification file and the compiler JAR file in the root folder, and the runtime library in the “lib” folder.

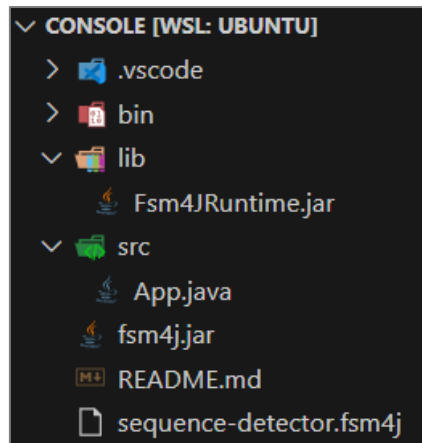


Figure 29 – Console Interface Example: Project Setup

Executing the following command in the terminal will generate the files and put them in the specified package in the source folder.

```
java -jar fsm4j.jar sequence-detector.fsm4j -o src/com/console
```

Fig. 29 shows the whole project window after executing the command and running the code. The SequenceDetector class is modified the same way in the web interface example, and the main class is the same as well.

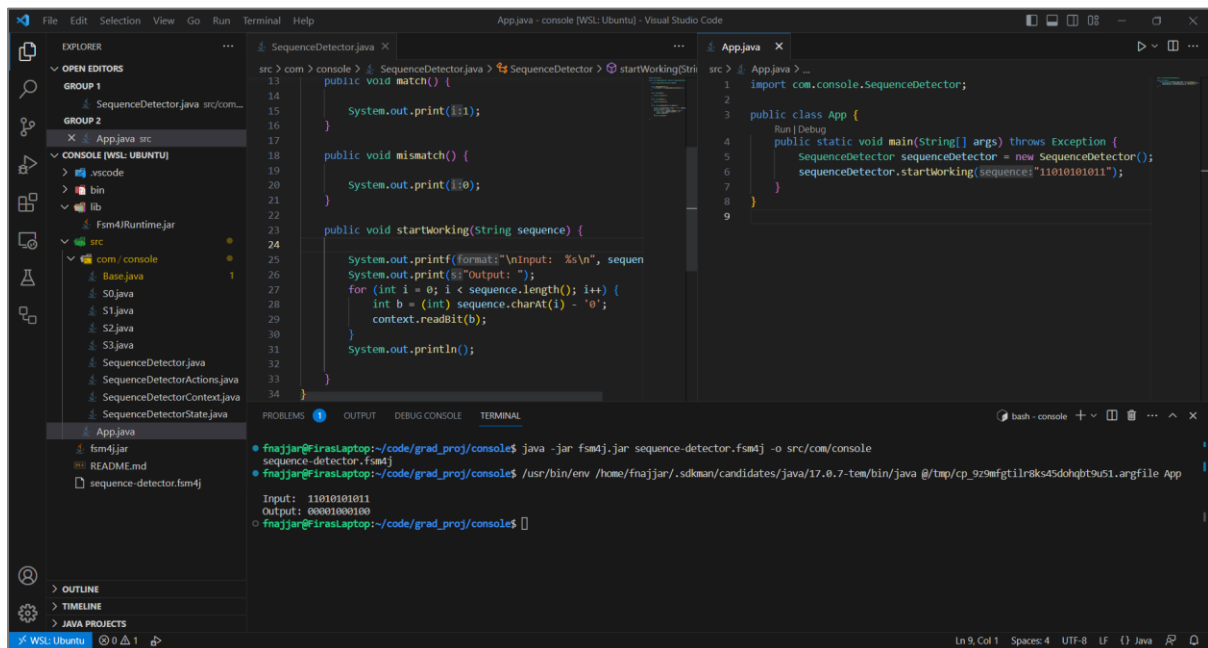


Figure 30 – Console Interface Example: Running Code

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

Through careful planning and a focus on usability, this project has successfully delivered an FSM design automation tool that offers two interfaces to cater to the different preferences and needs of developers. The combination of a visual diagramming interface and a command-based console interface, powered by a robust backend framework, provides a comprehensive solution for designing and generating code using the State design pattern. This significantly reduces the manual effort and potential for errors in hand-coding the State pattern implementation.

The project's main objective was to provide developers with a flexible and intuitive solution for designing and generating code based on FSM models. The web interface serves as a visual diagramming tool, allowing users to create and manipulate FSM models through an interactive and user-friendly interface. This interface provides a graphical representation of the FSM, enabling users to define states, transitions, and associated properties effortlessly.

Complementing the web interface, the console interface offers a command-based approach that allows users to interact with the tool through a command-line interface. This interface accepts user commands and passes them to the compiler. Developers who prefer a text-based workflow can leverage the console interface to define FSM models, specify code generation options, and trigger the generation process directly from the command line.

The compiler engine forms the core of the tool, facilitating the translation of user-defined FSM models into source code. It handles the interpretation of user input from both the web interface and the console interface, ensuring consistency and accuracy in the generated code. The framework incorporates robust error handling mechanisms to provide informative error messages, guiding users in rectifying any issues during the code generation process.

From a software engineering perspective, the tool's architecture has been thoughtfully designed to ensure flexibility and extensibility. The implementation is based on modular components, allowing for easy integration of new features and enhancements. The code generation process is well-abstracted, promoting code reuse and enabling developers to customize and extend the tool to accommodate specific project requirements.

This tool finds applications in various domains, including engineering and bioinformatics. In engineering, the tool can aid in the design and analysis of complex systems such as control systems and communication protocols. In bioinformatics, the tool facilitates the representation and analysis of biological processes, such as gene regulatory networks and protein signaling pathways. The tool's versatility and adaptability across these domains contribute to enhanced efficiency, productivity, and system understanding for users.

Looking ahead, there is potential for the tool to expand its capabilities and support additional programming languages and design patterns. By embracing industry standards and

incorporating user feedback, the tool can evolve to address the changing needs of developers and keep pace with advancements in software engineering practices.

5.2. Future Work

By offering insights into future directions, this section aims to stimulate discussions and provide guidance for future contributors. The following subsections highlight areas that need further investigation and extension, and suggest possible improvements based on the limitations of the current project.

5.2.1. Visual Simulations

One of the main goals of this project is to prove to be useful to the developers. This can be done by expanding the functionality of the tool and opening new opportunities for new use cases. Including visual simulations in the web application's toolbox can add so much value to the tool and tremendously increase its usefulness.

This feature [39] provides the ability to actively execute the FSM model during design and simulate it via dynamic state triggers. It allows the users to interact with the FSM and observe its response in real time, which helps in understanding and tracking its behavior. Visual simulations can be used for complex system analysis, software testing and educational purposes.

The simulations can be represented in many ways. For example, the current state of the FSM can be visually highlighted or indicated. By invoking triggers and conditions, transitions can be initiated based on the defined rules. These can be activated through buttons, user input or other options that users can select. During state transitions, the visual simulation may highlight or animate the transition path, making it easier for users to track the flow and sequence of state changes. Then, the displayed state is updated to reflect the current active state. This dynamic visualization provides real-time feedback on the FSM's behavior and state transitions.

5.2.2. FSM Validation and Error Handling

Errors in simulations can occur when an unexpected or invalid state transition or event is encountered. These errors can happen due to various reasons, such as incorrect input, violations of constraints and requirements, or inconsistencies in the model. FSM validation and error handling can be considered as a separate expansion to visual simulations, which helps ensure the validity, integrity, and reliability of the FSM model and its behavior by providing clear feedback and appropriate error handling mechanisms.

Many approaches can be used to handle errors. One approach is displaying live error messages to inform the user about the nature of the error. These messages can describe the specific error encountered, provide guidance on how to resolve it, and suggest corrective actions. For example, the tool may suggest an alternative valid transition or provide an option to rollback the FSM to a previous valid state. In some cases, the FSM simulation tool may attempt to recover from the error automatically. Visual cues can also be used to highlight the specific elements or transitions in the FSM diagram that caused the error. The erroneous transition may

be highlighted in red or marked with an error symbol to draw the user's attention to the problematic area.

5.2.3. Session Saving

By implementing session saving [40] effectively, applications can enhance user experience, reduce friction, and improve overall user satisfaction by providing continuity and eliminating the need for users to repeat their actions or re-enter information during subsequent visits or sessions. This allows users to resume their activities or restore their previous state when they revisit the application or website.

In the context of this project's web interface, when users are creating or modifying FSM diagrams, session saving can store the state of the diagram. This includes the states, transitions, and associated properties that users have defined. Users can return to the application and resume their FSM editing process without losing their progress.

Session saving can also be used to store simulation scenarios or test cases. This includes the inputs, expected outputs, and any intermediate results generated during the simulation. Moreover, it can support collaboration features. Multiple users can work on the same FSM diagram simultaneously, ensuring that each user's changes and edits are preserved and can be synchronized with others.

By leveraging session saving in FSM diagram web applications, users can have a seamless experience in designing, analyzing, and simulating FSMs. They can work on their FSMs across multiple sessions, experiment with different scenarios, and collaborate with others effectively. Session saving ensures that users can resume their work, maintain a history of changes, and perform analysis without losing progress or manually recreating the FSM diagrams.

5.2.4. Asynchronous States

Asynchronous states [41] refer to states that do not transition based on explicit input triggers. In an asynchronous FSM, the transition between states is determined by internal conditions. This can be particularly useful in representing states that perform background tasks such as periodically updating variables. Adding this element as part of the FSM structure gives more chance for customization and variety within models.

One common example of an asynchronous state is a "wait" or "idle" state. The FSM remains in this state until the specified condition is met or the timer expires, at which point it transitions to the next state based on the internal logic of the FSM.

It is important to note that asynchronous states can introduce complexities in FSM design and analysis. Since these states are not directly tied to external events, their behavior may not be easily predictable or controllable. Careful consideration should be given to ensure proper synchronization and coordination with other parts of the system to prevent unintended behaviors or race conditions.

Asynchronous states can be represented using a self-transition or a loop with an attached symbol on it. The symbol indicates the duration or condition that must be satisfied before

transitioning to the next state. This visual representation helps in understanding the flow of control and the presence of asynchronous behavior within the FSM.

5.2.5. New Formats and Languages Support

Many of the features in this project have been designed with consideration for extension. The FSM parsers and code generators are built using the Strategy design pattern. This allows contributors to easily extend and implement their own parsers and generators. The full implementation details behind this are discussed in Chapter 3.

Adding new target programming languages can be very useful based on the use case of that language. For example, generating State pattern code in JavaScript that can manage the behavior of user interfaces in response to user interactions or application events. For example, a form component can have different states such as "editing," "validating," or "submitted," each requiring specific behavior and validation rules.

As for parsers, adding support for new input format also adds to the usability of the tool. For example, parsing YAML Ain't Markup Language (YAML) [42] can be very useful. YAML is a human-readable data serialization format that provides a simple and concise way to represent structured data. It is often used for automation scripts, and it can be leveraged to represent FSMs in scripts that involve state-based decision making or control flow. Moreover, YAML's simplicity and readability make it suitable for defining FSM configurations, especially in scenarios where human-readable and editable configuration files are desired.

REFERENCES

- [1] “Object-oriented programming,” *Wikipedia*. May 27, 2023. Accessed: Jun. 04, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=1157313739
- [2] “State.” <https://refactoring.guru/design-patterns/state> (accessed Jun. 04, 2023).
- [3] S. Kathikeyan, “The Impact of Poor Software Quality in Business (Infographic) - APEX,” *APEX Global*, Aug. 26, 2016. <https://www.apexgloballearning.com/impact-poor-software-quality-business-infographic/> (accessed Mar. 08, 2023).
- [4] *The lost art of software design by Simon Brown*, (Oct. 13, 2022). Accessed: Mar. 09, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=36OTe7LNd6M>
- [5] G. Tomassetti, “The ANTLR Mega Tutorial,” *Strumenta*, Mar. 08, 2017. <https://tomassetti.me/antlr-mega-tutorial/> (accessed Jun. 04, 2023).
- [6] “SMC: The State Machine Compiler.” <https://smc.sourceforge.net/> (accessed Mar. 09, 2023).
- [7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition. Reading, Mass: Addison-Wesley Professional, 1994.
- [8] “JSON.” <https://www.json.org/json-en.html> (accessed Jun. 04, 2023).
- [9] “GitHub.com Help Documentation,” *GitHub Docs*. <https://ghdocs-prod.azurewebsites.net/en> (accessed Jun. 04, 2023).
- [10] T. Donohue, “How to create a JAR file with Maven in IntelliJ,” *Tutorial Works*, May 09, 2022. <https://www.tutorialworks.com/intellij-maven-create-jar/> (accessed Jun. 04, 2023).
- [11] “61+ Workflow Automation Statistics and Forecast in 2023,” Nov. 15, 2022. <https://quixy.com/blog/workflow-automation-statistics-and-forecasts/> (accessed Mar. 09, 2023).

- [12] C. Cohenour, “Teaching Finite State Machines (FSMs) as Part of a Programmable Logic Control (PLC) Course,” in *2017 ASEE Annual Conference & Exposition Proceedings*, Columbus, Ohio: ASEE Conferences, Jun. 2017, p. 28917. doi: 10.18260/1-2--28917.
- [13] R. Balogh and D. Obdržálek, “Using Finite State Machines in Introductory Robotics: Methods and Applications for Teaching and Learning,” 2019, pp. 85–91. doi: 10.1007/978-3-319-97085-1_9.
- [14] S. Hussain, J. Keung, and A. A. Khan, “The Effect of Gang-of-Four Design Patterns Usage on Design Quality Attributes,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic: IEEE, Jul. 2017, pp. 263–273. doi: 10.1109/QRS.2017.37.
- [15] “SMC -- The State-Machine Compiler,” in *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition. Reading, Mass: Addison-Wesley Professional, 1994, pp. 429–442.
- [16] “XState - JavaScript State Machines and Statecharts.” <https://xstate.js.org/> (accessed Mar. 09, 2023).
- [17] “Blockly,” *Google for Developers*. <https://developers.google.com/blockly> (accessed Jun. 04, 2023).
- [18] “Unified Modeling Language,” *Wikipedia*. May 09, 2023. Accessed: Jun. 04, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=1154002355
- [19] “RuntimeException (Java Platform SE 8).” <https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html> (accessed Jun. 04, 2023).
- [20] “TypeScript: JavaScript With Syntax For Types.” <https://www.typescriptlang.org/> (accessed Jun. 04, 2023).
- [21] “React.” <https://react.dev/> (accessed Jun. 04, 2023).
- [22] “Chakra UI - A simple, modular and accessible component library that gives you the building blocks you need to build your React applications,” *Chakra UI: Simple, Modular and*

Accessible UI Components for your React Applications. <https://chakra-ui.com> (accessed Jun. 04, 2023).

[23] “Redux - A predictable state container for JavaScript apps. | Redux.” <https://redux.js.org/> (accessed Jun. 04, 2023).

[24] “JavaScript diagramming library for interactive UIs – JointJS.” <https://www.jointjs.com/> (accessed Jun. 04, 2023).

[25] “Spring Boot,” *Spring Boot*. <https://spring.io/projects/spring-boot> (accessed Jun. 04, 2023).

[26] “API,” *Wikipedia*. May 22, 2023. Accessed: Jun. 04, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=API&oldid=1156413408>

[27] “HTTP | MDN,” May 24, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP> (accessed Jun. 04, 2023).

[28] “Runtime library,” *Wikipedia*. Jul. 12, 2021. Accessed: Jun. 04, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Runtime_library&oldid=1033305968

[29] “Argparse4j - The Java command-line argument parser library — argparse4j 0.9.0 documentation.” <https://argparse4j.github.io/> (accessed Jun. 04, 2023).

[30] “Jackson Project Home @github.” FasterXML, LLC, Jun. 03, 2023. Accessed: Jun. 04, 2023. [Online]. Available: <https://github.com/FasterXML/jackson>

[31] “Strategy.” <https://refactoring.guru/design-patterns/strategy> (accessed Jun. 04, 2023).

[32] BillWagner, “Polymorphism,” Jan. 31, 2023. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism> (accessed Jun. 04, 2023).

[33] “XML,” *Wikipedia*. May 19, 2023. Accessed: Jun. 04, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=XML&oldid=1155641749>

[34] D. S. ROY, “FSM design,” *Digital System Design*, Jun. 05, 2018. <https://digitalsystemdesign.in/fsm-design/> (accessed Jun. 03, 2023).

[35] “Visual Studio Code - Code Editing. Redefined.” <https://code.visualstudio.com/> (accessed Jun. 04, 2023).

- [36] “DNA,” *Wikipedia*. Jun. 02, 2023. Accessed: Jun. 04, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=DNA&oldid=1158157352>
- [37] “Sequence analysis,” *Wikipedia*. Dec. 14, 2022. Accessed: Jun. 04, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sequence_analysis&oldid=1127458093
- [38] “The TCP/IP Guide - TCP Operational Overview and the TCP Finite State Machine (FSM).” http://tcpipguide.com/free/t_TCPOperationalOverviewandtheTCPFiniteStateMachineF-2.htm# (accessed Jun. 03, 2023).
- [39] “What is Visual Simulation? (with picture).” <https://www.easytechjunkie.com/what-is-visual-simulation.htm> (accessed Jun. 04, 2023).
- [40] “What does save session do? - Linux Mint Forums.” <https://forums.linuxmint.com/viewtopic.php?t=218814> (accessed Jun. 04, 2023).
- [41] “Asynchronous Finite State Machines,” in *Digital Electronics 3*, John Wiley & Sons, Ltd, 2016, pp. 213–286. doi: 10.1002/9781119371083.ch3.
- [42] “The Official YAML Web Site.” <https://yaml.org/> (accessed Jun. 04, 2023).

APPENDIX A

USER MANUAL

The following user manual is provided on the GitHub repository the project is hosted on.

FSMT

FSMT is a tool that can generate working code from finite state machine specifications

The tool provide to modes to provide FSM specifications

- Textual mode using a custom language
- Graphical web application to provide the specification visually

How to run

Running the project locally requires

Running the compiler only

To run the compiler commandline tool

- Clone the project into a folder on your local machine using `git clone https://github.com/ammam-abu-yaman/FSM2Java.git`
- Run `mvnw clean install` to compile the project
- The compiler can be found in `src/main/compiler/target/compiler-jar-with-dependencies.jar`

The resulting Jar file contains all the necessary dependencies to run the project compiler

Running the application

The application contains the web UI and a server that run the compiler on your behalf. To compile and run the application follow these steps

- Clone the project into a folder on your local machine using `git clone https://github.com/ammam-abu-yaman/FSM2Java.git`
- Run `mvnw clean install` to compile the project
- The application can be found in `target/server-0.0.1-SNAPSHOT.jar`

Now you have two choices to run the application. You run it directly on your local machine using `java -jar target/server-0.0.1-SNAPSHOT.jar` and the web application will boot up on `localhost:8080`

You can also run it using docker after building an image from the compiled project using the provided Dockerfile.

User Manual

There are two ways to use the project, one is to use the compiler directly after writing your FSM description using the fsm4j language or use the web ui to build a visual description of your FSM.

Fsm4j language spec

Meta data

Meta data provides information to the compiler about about your application specifics to be used in code generation.

The syntax for a meta data directive is as follow

```
$property-name property-value [property-value]*
```

The supported properties are:

package: used to specify the application's package to be included in the generated code (only relevant for Java code generation)

class: specify the application name that is used to name all the generated classes

initial-state: used to specify the starting state of the FSM

actions: multivalue property that is used to specify the actions that should be present in the application class code.

States specification

The state specficiation comes after the the meta data section. This section describes the states, state transitions, action code and guards that composes the finite state machine

the section starts with `{{` and ends with `}}`

State specification syntax

States are generally defined as follow

```
state-name {  
    [__enter__ {# .. #}]  
    [__exit__ {# .. #}]  
  
    transition-one,  
    transition-two,  
    transition-three  
    ..  
}
```

starting with the state name then optionally defining enter code and exit code that will be executed every time that state is entered or exited respectively, followed by the state's transitions' definitions.

Transition specification syntax

Transitions are generally defined as follow

```
transition-name [[# guard-condition #]] => next-state {# .. #}
```

starting with the transition name followed by an optional guard condition then the next state followed by the code that will be executed upon hitting the transition.

Base State

You can also define a Base state that act as a failsafe mechanism that contains default implementation of the FSM transitions, it can be defined as any other State but the name of the state must be `Base`.

APPENDIX B

PROJECT TIME CHART

S1: Ammar Abu Yaman				
S2: Firas AlNajjar				
Project Start:			Mon, 10/10/2022	
TASK	ASSIGNED TO	PROGRESS	START	END
Research & Discovery				
Finite State Machines	S1+S2	100%	10/10/22	10/17/22
Compilers Design	S1+S2	100%	10/17/22	10/24/22
Front End design: React, Bootstrap	S2	100%	10/24/22	10/31/22
Drag & Drop Front End Technologies	S2	100%	10/31/22	11/7/22
Backend Development & APIs	S2	100%	11/7/22	11/14/22
Technical Documentation Writing	S1	100%	11/14/22	11/21/22
Presentation Directing	S1+S2	100%	11/21/22	11/28/22
Compiler Implementation				
Design of Compiler's input format	S1	100%	11/28/22	12/5/22
Scanner & Parser	S1	100%	12/5/22	12/12/22
AST semantic passes	S1	100%	12/12/22	12/19/22
Error and warning messages	S2	100%	12/19/22	12/26/22
AST to Java Translation Code	S2	100%	12/26/22	1/2/23
Organizing Java code into classes	S2	100%	1/2/23	1/9/23
Web Tool Development				
User Interface Design	S2	100%	1/9/23	1/14/23
Project Workspace	S1+S2	100%	1/14/23	1/19/23
FSM Visual Design Tools	S1	100%	1/19/23	1/24/23
FSM Validation	S1	100%	1/24/23	1/29/23
Graphical FSM to Portable format conversion	S1+S2	100%	1/29/23	2/3/23
Design and implementation of backend APIs	S1+S2	100%	2/3/23	2/8/23
Integration of the backend with the compiler	S1+S2	100%	2/8/23	2/13/23
Integration & Testing & Deployment				
Integration of the front end visual tool with th	S1	100%	2/13/23	2/18/23
Testing the system on local machine	S1	100%	2/18/23	2/23/23
Fixing integration issues and optimizing the sys	S2	100%	2/23/23	2/28/23
Deploying the system on a server for others tc	S1+S2	100%	2/28/23	3/5/23
Writing the user manual for the System	S1+S2	100%	3/5/23	3/10/23

APPENDIX C

PRESENTATION SLIDES

- Fsm4J Project Presentation

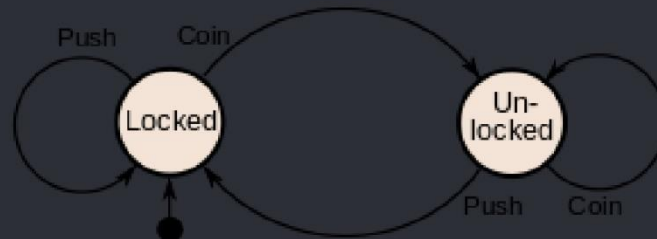
1

Introduction

2

- Finite state machines (FSMs)

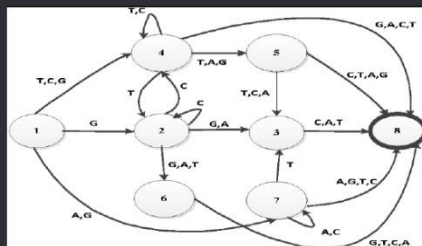
Finite state machines are powerful tool that is used to describe models, algorithms and workflows of systems. However implementing them in code isn't straightforward.



- Applications of finite state machines

Bioinformatics

FSMs are important tool in bioinformatics, where they are used to model and analyze biological sequences, perform sequence alignment, and predict structural and functional properties of biomolecules.

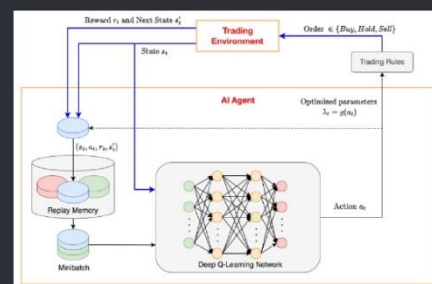


Finance

FSMs can be greatly utilized in finance and can be used to model and analyze complex financial systems, automate trading strategies, detect patterns, and manage risk.

Digital circuit design

FSMs are used to model and control the behavior of digital circuits, such as control units in CPUs, memory controllers, and communication protocols.



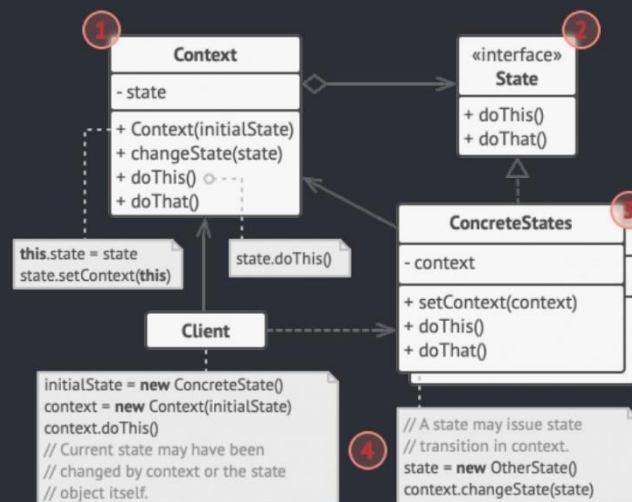
- Classical approach to implement FSMs

Typical implementation of FSMs uses conditional logic and state ids to manage state transitions. This approach has many problems including:

- Convoluted conditional logic is hard to understand and error prone.
- Difficult to reason about state transitioning.
- Violates the OCD principle where any change requires modifying existing code.

5

- Let's instead use State Design pattern inspired by the work of Robert Martin aka Uncle Bob



6

• State Pattern

○ Why use State design pattern

State pattern comes from the work of uncle Bob, it uses classes to represent states and methods on these classes to represent transitions, some of the ways state pattern help implementing state machines:

Allow for clear mapping from design to code.

reduces coupling between logic and implementation rather than relying on convoluted conditional logic it uses polymorphism instead.

Adhere to the OCD principle where functionality can be added by adding new state classes.

Disadvantages of using state pattern

State pattern has few disadvantages:

Control logic is spread across multiple files making it difficult to visualize.

Tedious to implement.

Error prone especially with handling transition logic.

7

2 A Better Alternative

8



Building on State Pattern

Our approach uses the state pattern as a starting point.

9

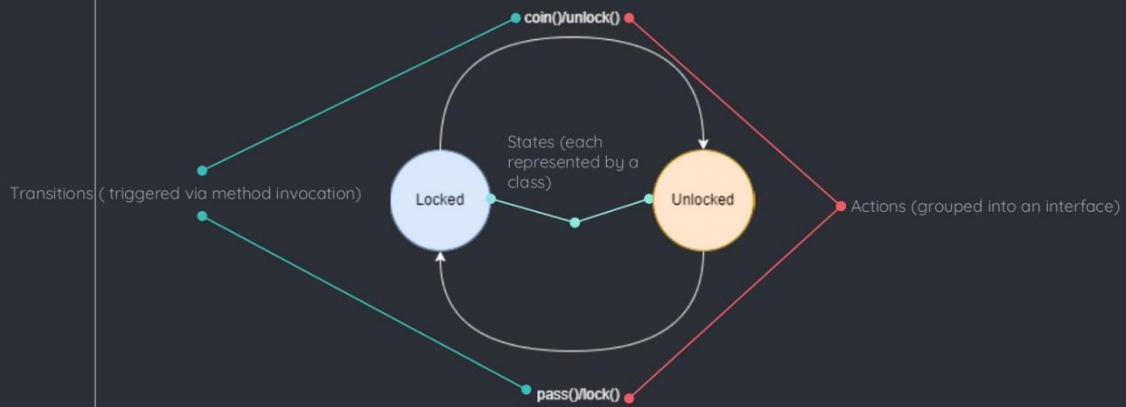
- Our approach

- Fsm4J a tool that merges the representation of the FSM with the state pattern implementation using:

- Building easy to understand visual/textual representation of FSM.
- Automatic code generation of state machine code
- Extending state pattern and refining the original design.

10

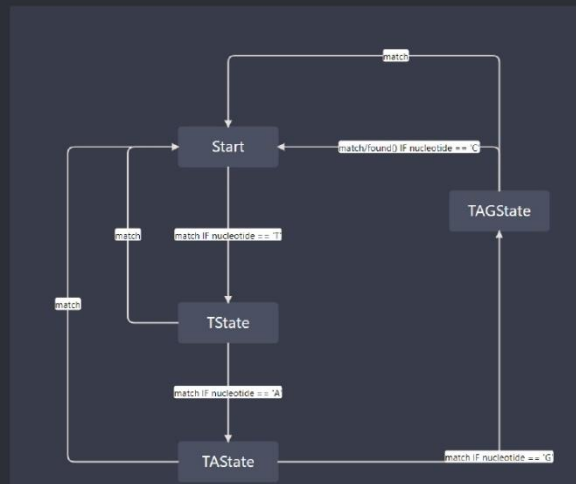
- Modularising state pattern into different components



11

Graphical diagramming method

Graphical tool providing convenient drag and drop method, easy to prototype and visualizing the design.



12

Textual method

Write the description of the state machine in a domain specific language.

```
$package com.turnstile
$class Turnstile
$initial-state Locked
$actions lock unlock alarm thankyou

${{
  Locked {
    coin => Unlocked {# ctx.unlock(); #},
    pass => null {# ctx.alarm(); #}
  },
  Unlocked {
    pass => Locked {# ctx.lock(); #},
    coin => nil {# ctx.thankyou(); #}
  }
}}$
```

13

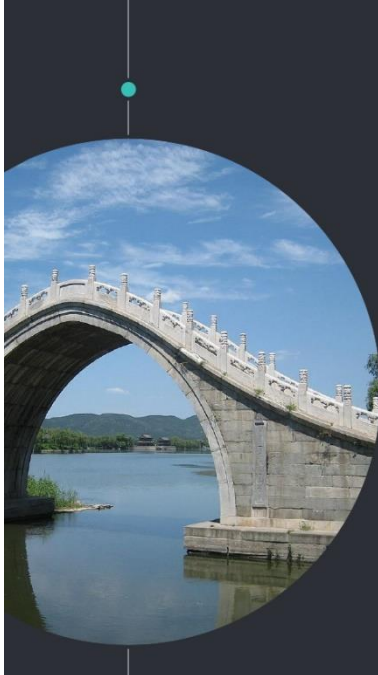
Automatic code generation

State classes, transitions and actions methods are generated automatically using our compiler.

TURNSTILE

- Base.java
- Locked.java
- Turnstile.java
- TurnstileActions.java
- TurnstileContext.java
- TurnstileState.java
- Unlocked.java

14



Our tool help bridge the design with the implementation and allows efficient and easy mapping between the description and code.

15



Provides a single point of change where change in design can immediately be reflected in code rather than be recoded manually.

16

3

Additional Features

17



Improving state pattern

We extended the original state pattern with additional features to increase it's generality

18

- Default States

Base states act as a safe guard or a **default** in a **switch statement** for states without a specific transition's implementation.

```
Base {  
    transition1 => StateX {# ... #},  
    ...  
    transitionN => StateY {# ... #}  
}
```

19

- Enter & Exit actions

Exit and enter actions are methods that are invoked when Abon entering or exiting a specific state. Useful for cross-cutting concerns.

```
Connected {  
    __enter__ {#  
        initiateRemoteConnection(ctx.getId());  
        logger.log("Entered Connected State and initiated Connection");  
    #}  
    __exit__ {#  
        terminateRemoteSession();  
        logger.log("Terminated session and disconnected");  
    #}  
    ...  
}
```

20

- Guards

- Guards allow us to make a transition conditional and only trigger if a certain condition met

```
SomeState {  
    transition(x: int) [# isOdd(x) #] => StateX {# ... #},  
    transition(x: int) [# isEven(x) #] => StateY {# ... #},  
    transition(x: int) [# isPrime(x) #] => StateZ {# ... #},  
    ...  
}
```

21

- A modular design for extension

- **Support for additional input formats**

The compiler frontend support both JSON and a custom domain specific language as input, with the ability to easily add support for additional languages (e.g. XML).

- **Support for multi language code generation**

While the current implementation only support generating code for Java, it can be easily extended to support other languages such as C++ which is also part of the future work of the project.

22

4

Live demo

Let's start with the first set of slides

23

Thanks

ANY QUESTIONS?

24

APPENDIX D

USB SOFT COPY

The attached USB flash drive below contains the following files:

- Project documentation
- Project proposal
- Project Gantt chart
- Project codebase (instructions in README.md file)