CPSC-406 Report

Marc Nathaniel Domingo
Chapman University

May 4, 2023

Abstract

The purpose of this course is to understand how the Theory of Algorithms coincide with the application of functions and algorithms in Software Engineering. In particular, this course places an emphasis on the concepts of Automata, Logic, and Concurrency in regard to algorithms in our eventual application of these topics to a final project.

Contents

1 Introduction

1.1 Week 1

During this week, the overall outline and "roadmap" of topics and project deadlines were covered during class, as well as an introduction to the concept of Definitive Finite Automata (DFA).

1.2 Week 2

Lectures covered Definitive Finite Automata (DFA) and Non-Definitive Finite Automata (NFA) in more detail, as well as how to convert NFA to DFA.

1.3 Week 3

During this week, the lectures covered the concepts of querying a database, as well as algorithms for Unification and Resolution.

1.4 Week 4

During this week, lectures covered the concept of Distributed Hash Tables (DHT) in the application of Decentralized Systems.

1.5 Week 5

During this week, the lectures covered the concepts of Turing Machines and their relation to Polynomial and Non-Polynomial Time Problems.

1.6 Week 6

During this week, the lectures reviewed Satisfiability as covered during our lectures on Non-Polynomial Time Problems in their application to a Sudoku Solver.

1.7 Week 7

During this week, the lectures covered the concept of Distributed Systems in terms of how to create a Logical Clock.

1.8 Week 8

During this week, the lectures covered the concept of Big O-complexity in terms of certain sorting algorithms such as bubble sort, in addition to an introduction to Complexity Theory.

1.9 Week 9

During this week, the lectures covered the concept of model checking in addition to how the Needham-Schroeder Key Protocol can be modeled through Spin to witness how messages can be intercepted.

1.10 Week 10

During this week, the lectures covered the concept of Temporal Logic and how it can be applied to solve Satisfiability problems.

1.11 Week 11

During this week, the lectures covered the concepts of Composing Automata in addition to writing Regular Expressions based off of Non-Definitive Automata (NFA).

1.12 Week 12

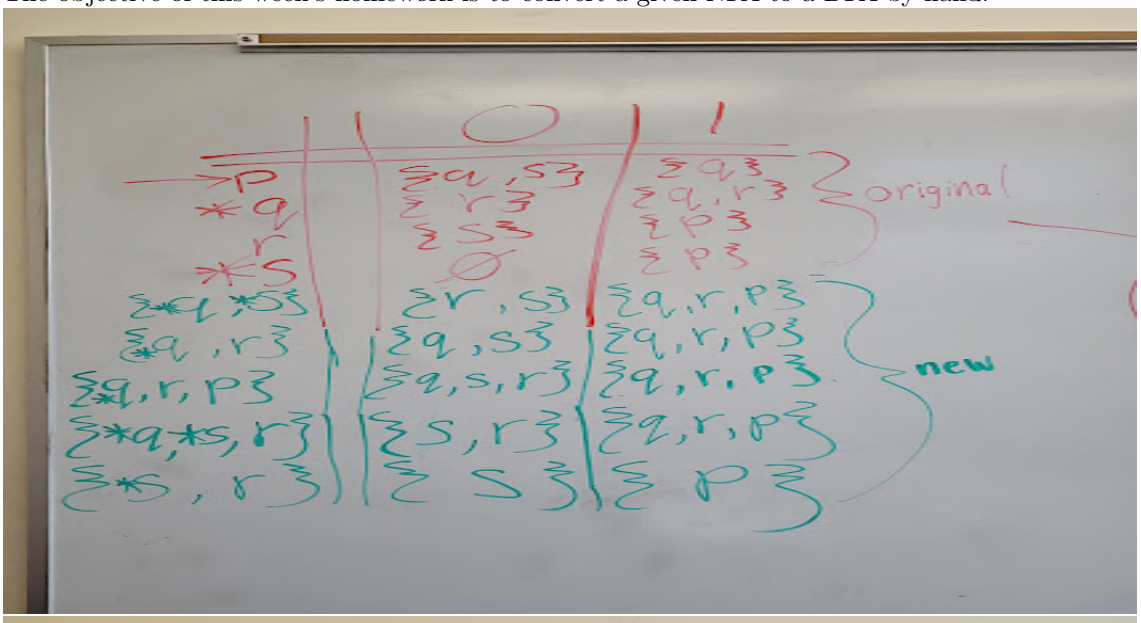
During this week, the lectures covered the concepts of Atomicity, Sequential Consistency, and Weak Memory in the context of a multi-threaded program in addition to what Safetiness, Liveliness, and Fairness are defined as within the context of a system.

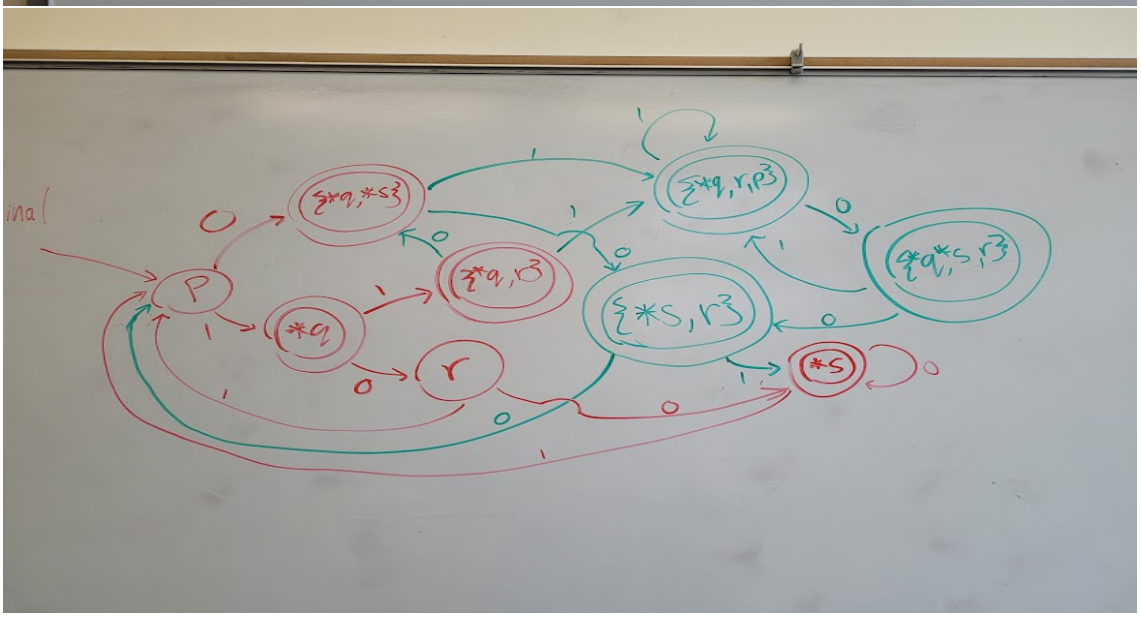
2 Homework

This section contains solutions to homework.

2.1 Week 2

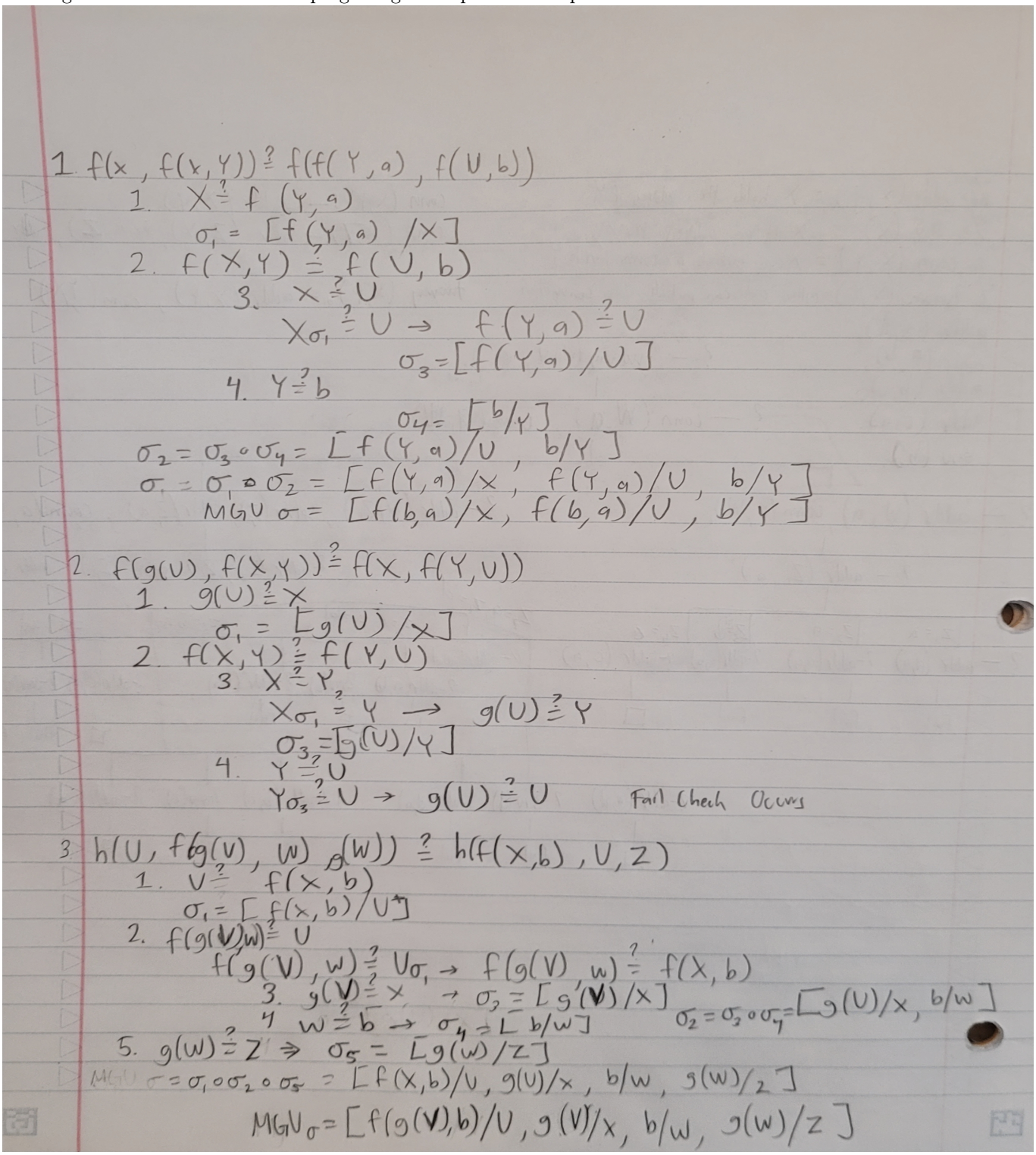
The objective of this week's homework is to convert a given NFA to a DFA by hand.

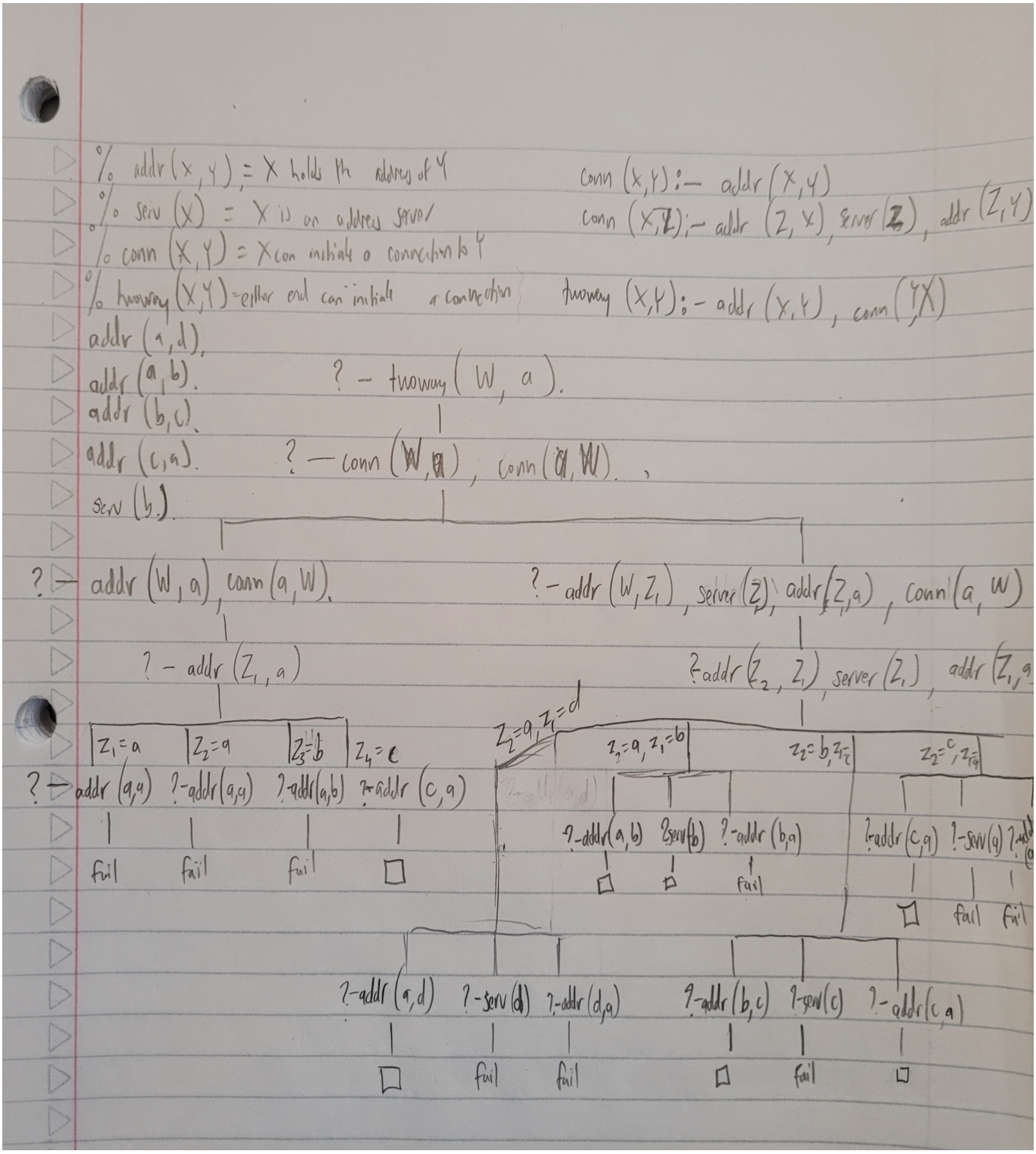




2.2 Week 3

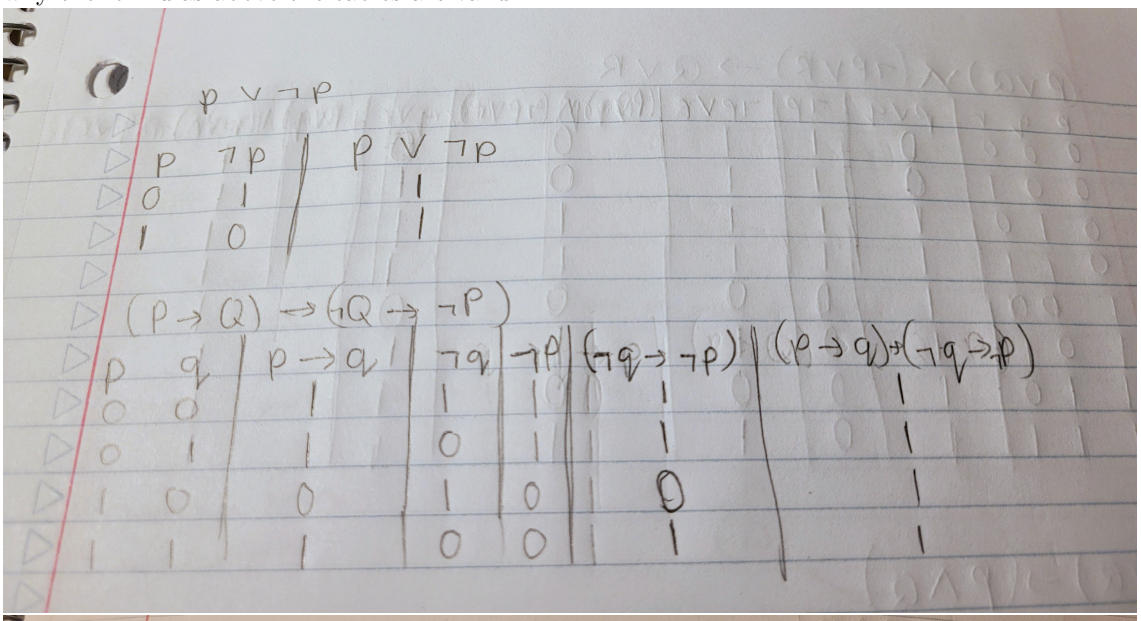
The objective of this week's homework is to solve Unification Algorithm problems in addition to practicing drawing an SLD-tree solution to a program given a particular input.

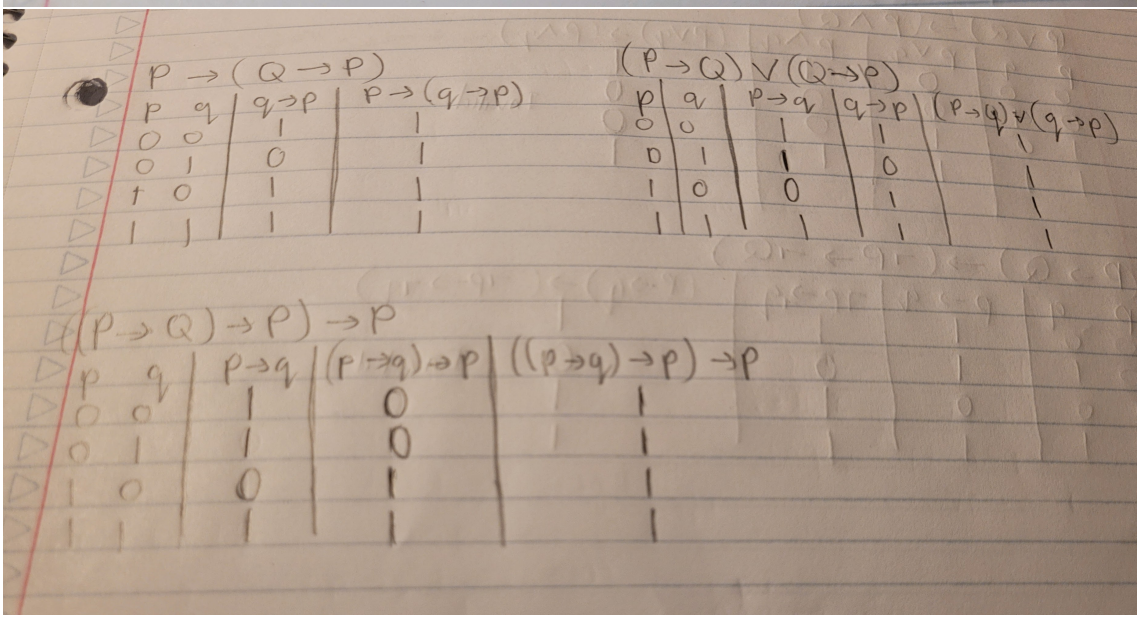


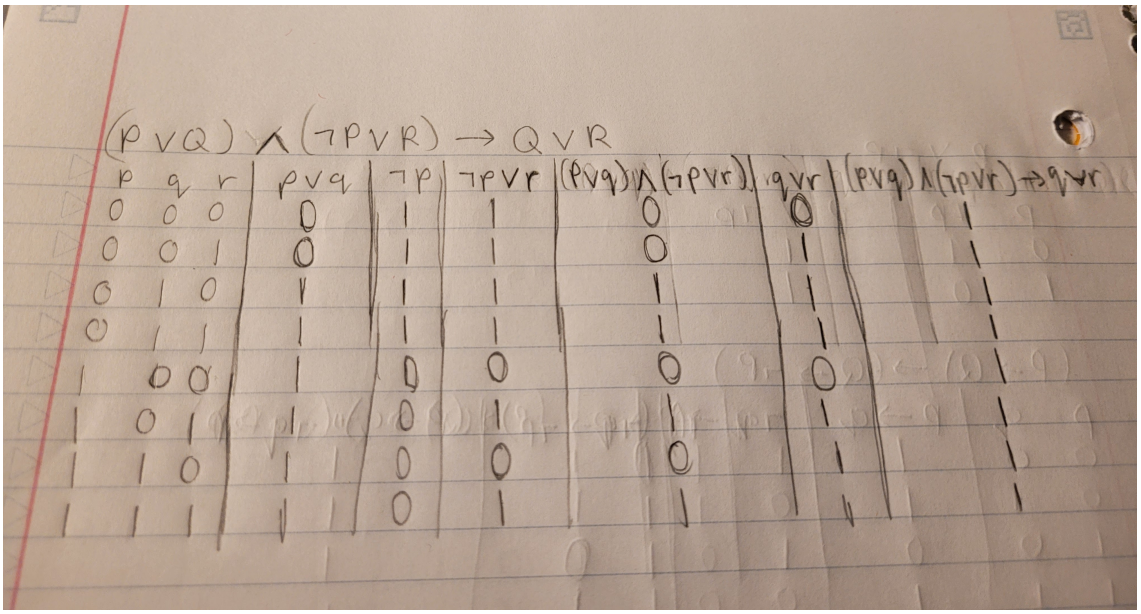

% addr(X,Y) = X holds the address of Y
% serv(X) = X is an address server
% conn(X,Y) = X can initiate a connection to Y
% twoway(X,Y) = either end can initiate a connection
addr(a,d).
addr(a,b).
addr(b,c).
addr(c,a).
serv(b).
conn(X,Y) :- addr(X,Y)
conn(X,Y) :- addr(Z,X), serv(Z), addr(Z,Y)
twoway(X,Y) :- addr(X,Y), conn(Y,X)
?- twoway(W,a).
?- conn(W,a), conn(a,W).
?- addr(W,a), conn(a,W).
?- addr(W,Z1), serv(Z1), addr(Z1,a), conn(a,W)
?- addr(Z1,a)
?- addr(Z2,Z1), serv(Z1), addr(Z1,
Z1=a
Z2=a
Z3=b
Z4=c
Z1=a, Z1=d
Z2=a, Z1=b
Z2=b, Z1=c
Z2=c, Z1=a
?- addr(a,a)
?- addr(a,a)
?- addr(a,b)
?- addr(c,a)
?- addr(a,b)
?- serv(b)
?- addr(b,a)
?- addr(c,a)
?- serv(a)
fail
fail
fail
fail
?- addr(a,d)
?- serv(d)
?- addr(d,a)
?- addr(b,c)
?- serv(c)
?- addr(c,a)
fail
fail
fail
fail

2.3 Week 6

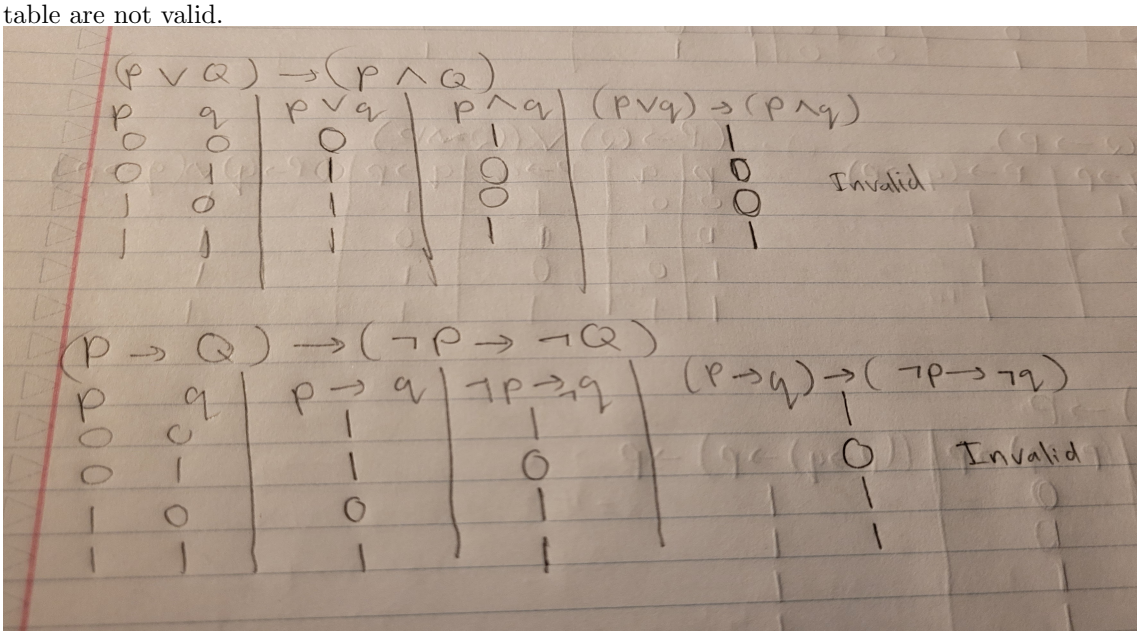
The objective of this week's homework was to create Indirect Truth Tables to understand Satisfiability in terms of Propositional Logic. For the following tables, we used the Indirect Truth Table method to prove why the formulas above the tables are valid.







For the following tables, the Indirect Truth Table method was used to prove why the formulas above the table are not valid.



2.4 Week 9

The objective of this week's homework is to explore exercises relating to the Needham-Schroeder Public-Key Protocol using Spin models.

One such example is to go through the program **ns.pml** and define what the meaning behind the propositions *success*, *AliceBob*, and *BobAlice*.

- **success** is a propositional variable that is true if and only if the first protocol (Alice) has successfully sent a reply to the second protocol (Bob), and if the second protocol (Bob) had sent a reply to the first protocol's (Alice) initial message.

- **aliceBob** is a propositional variable that is true if and only if the partner of the first protocol, Alice, is Bob.
- **bobAlice** is a propositional variable that is true if and only if the partner of the second protocol, Bob, is Alice.

The next exercise required determining which of the two formulas at the bottom of the code would violate the protocol when executed, as well as to explain the nature of the protocol based on that violation.

- The violation was caused by

$$ltl\{[](success\&\&aliceBob \rightarrow aliceBob)\}$$

- As the only difference between the two lines is the placement of **aliceBob** and **bobAlice** are swapped, this implies that the protocol is hard-coded such that the protocol does not get violated if Alice replies to Bob's reply, but in the event that the roles are reversed, the protocol is susceptible to intruders.

The next exercise of the homework involved looking through the execution sequence output from the protocol violation.

- The execution sequence ended up being 83 lines long, which matched up with the protocol violation message, with an additional line representing the attempt at the next check.

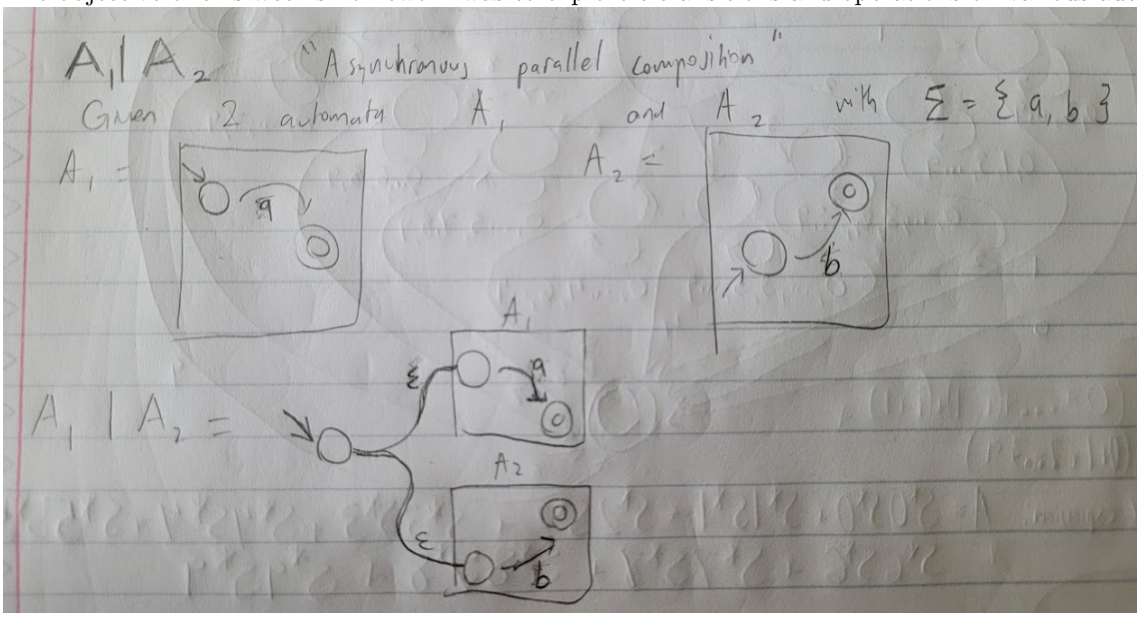
The last exercise involved creating a Message Sequence Chart (MSC) that represents a successful attack on the protocol.

```
6:   proc  0 (Alice:1) ns.pml:77 (state 12)  [network!msg1,partnerA,data.key,data.d1,data.d2]
 7:   proc  2 (Intruder:1) ns.pml:144 (state 1)      [network?msg,_,data.key,data.d1,data.d2]
23:   proc  2 (Intruder:1) ns.pml:201 (state 64)
[network!msg,rcpt,data.key,data.d1,data.d2]
24:   proc  1 (Bob:1) ns.pml:104 (state 1)   [network?msg1,bob,data.key,data.d1,data.d2]
34:   proc  1 (Bob:1) ns.pml:123 (state 17)
[network!msg2,partnerB,data.key,data.d1,data.d2]
35:   proc  2 (Intruder:1) ns.pml:144 (state 1)      [network?msg,_,data.key,data.d1,data.d2]
45:   proc  2 (Intruder:1) ns.pml:180 (state 37)
[network!msg,rcpt,intercepted.key,intercepted.d1,intercepted.d2]
46:   proc  0 (Alice:1) ns.pml:80 (state 13)  [network?msg2,alice,data.key,data.d1,data.d2]
52:   proc  0 (Alice:1) ns.pml:93 (state 20)
[network!msg3,partnerA,data.key,data.d1,data.d2]
53:   proc  2 (Intruder:1) ns.pml:144 (state 1)      [network?msg,_,data.key,data.d1,data.d2]
69:   proc  2 (Intruder:1) ns.pml:201 (state 64)
[network!msg,rcpt,data.key,data.d1,data.d2]
70:   proc  1 (Bob:1) ns.pml:126 (state 18)   [network?msg3,bob,data.key,data.d1,data.d2]
```

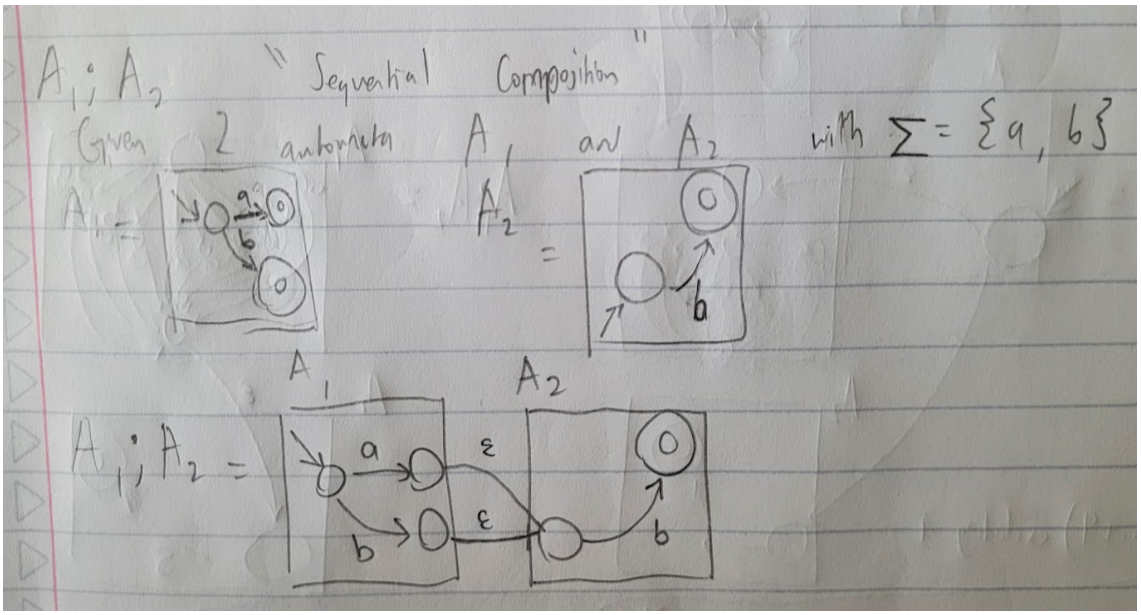
- The MSC represents a successful attack on the protocol as the intruder has managed to message both Alice and Bob before the other is able to receive and reply to the message from a proper protocol.

2.5 Week 11

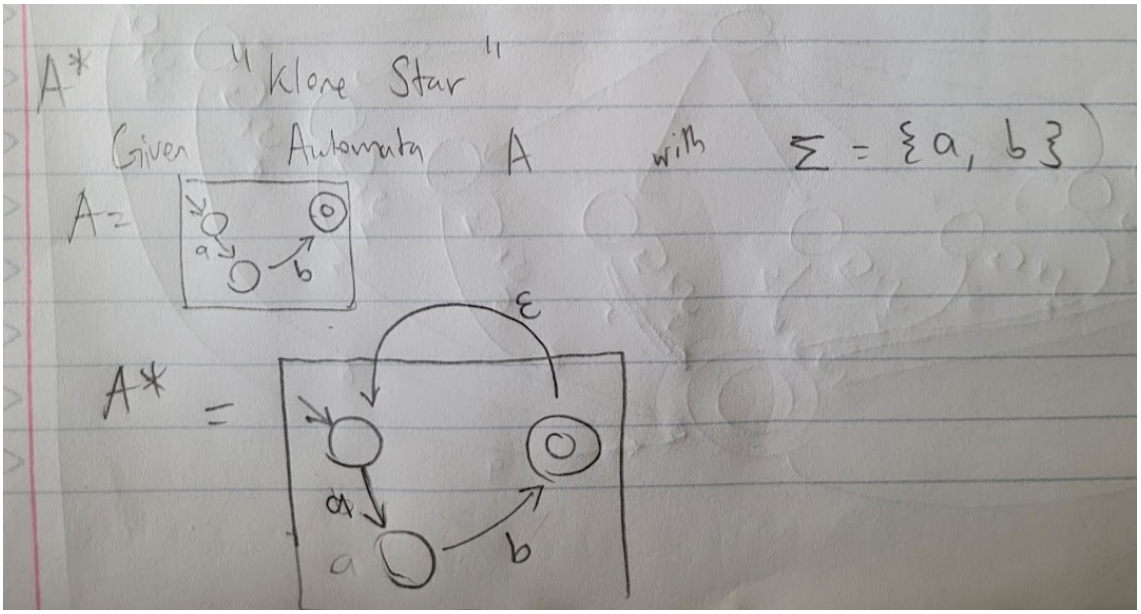
The objective of this week's homework was to explore ϵ-transitions and operations on various automata.



- Asynchronous Parallel Composition accepts languages that are capable of operating in parallel to each other, utilizing a set of strings that consists of the combined alphabet that each language accepts.
- The above example would have a regular expression of $A_1|A_2 = \epsilon a + \epsilon b$



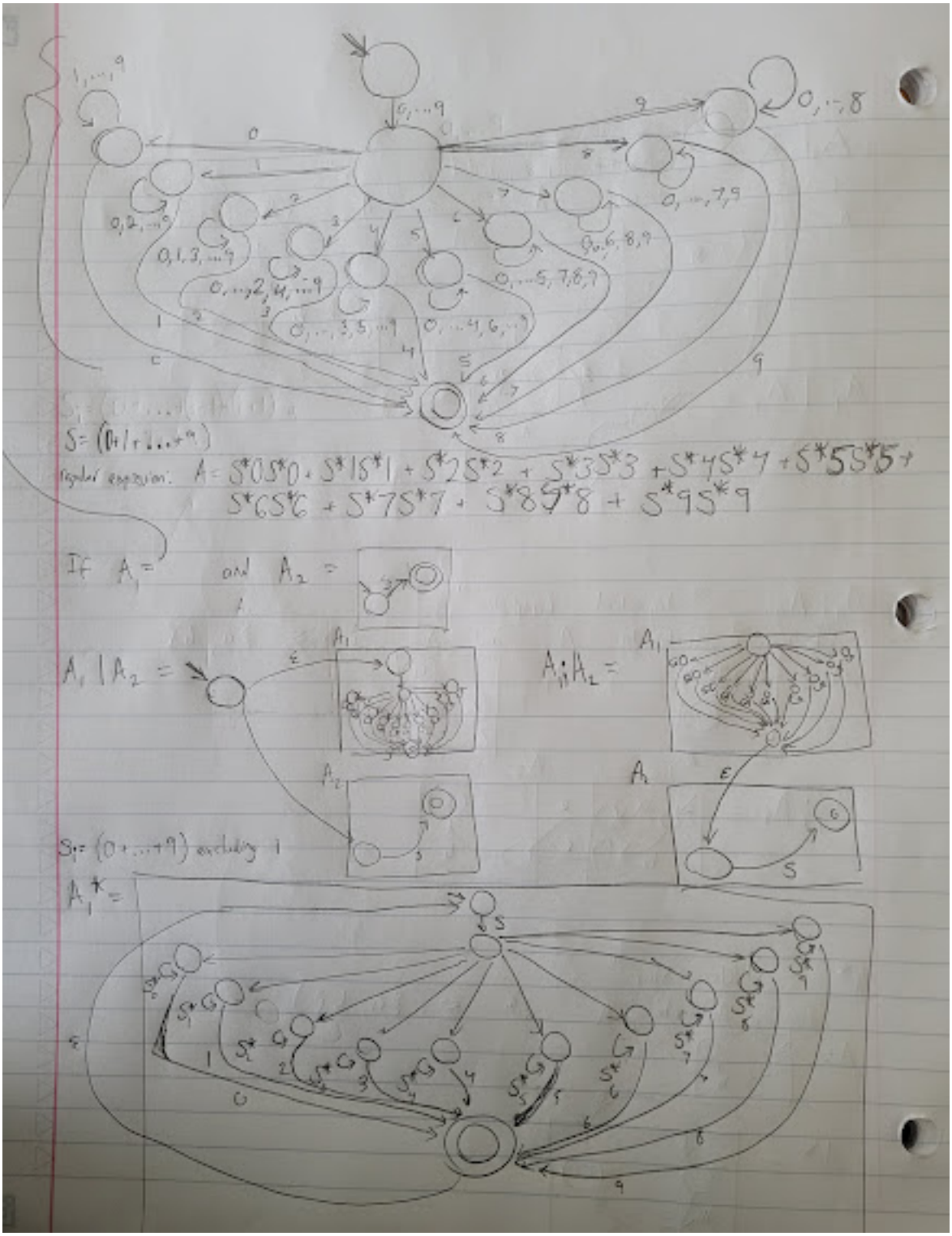
- Sequential Composition accepts languages that are capable of running in a sequential sequence, that is the output of one language is used as the starting point of another language, utilizing a set of strings that both languages share.
- The above example would have a regular expression of $A_1; A_2 = (a + b)\epsilon b$



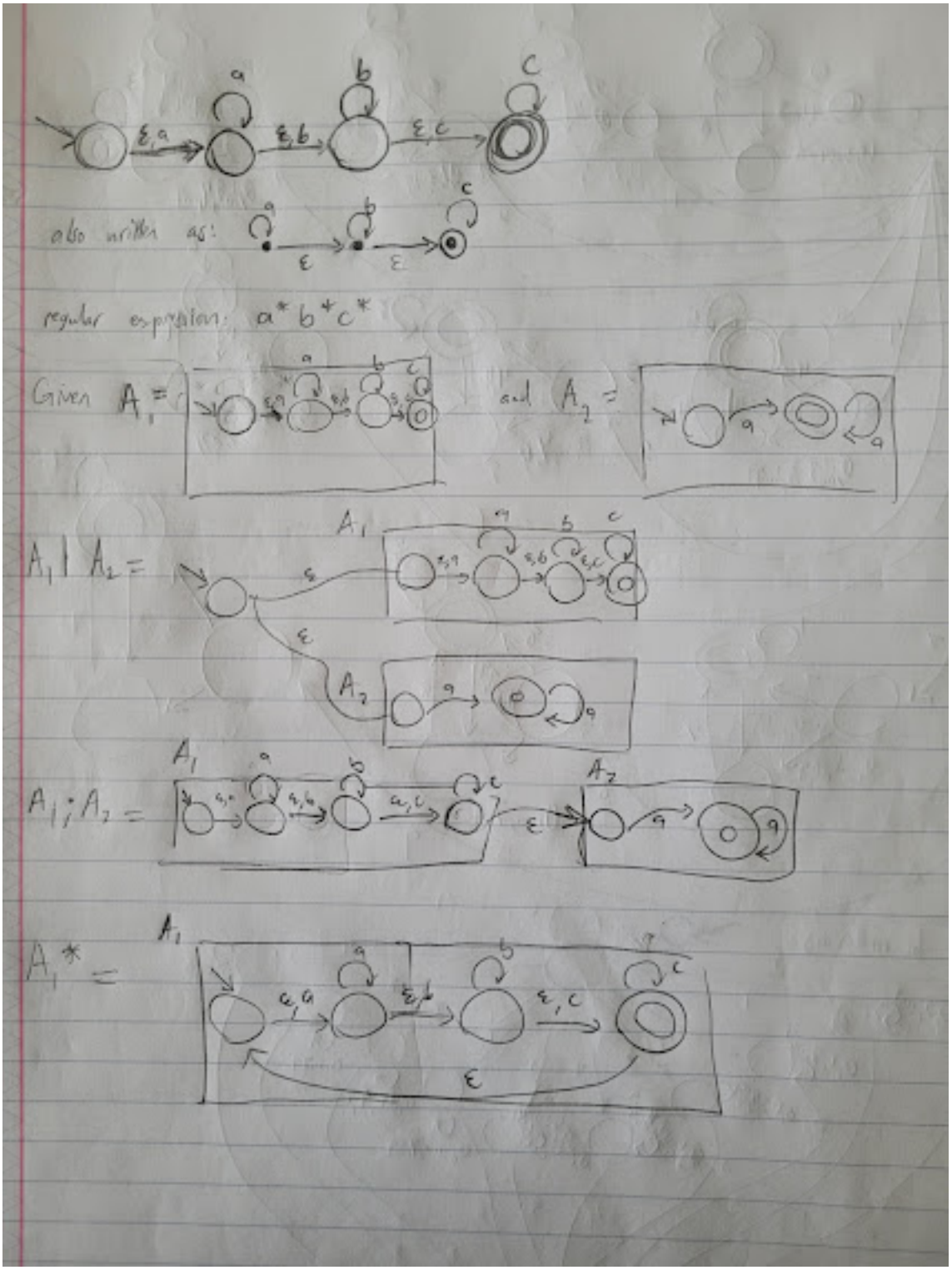
- Kleene Star accepts languages that are capable of infinitely repeating, starting back at the machine's initial state.
- The above example would have a regular expression of $A^* = ab\epsilon^*$

The next section of the homework consisted of finding the DFA and regular expressions of examples from the textbook, as well as finding the ϵ-NFA if the above operations were applied to the examples.

- The set of strings over the alphabet { 0, 1, ... 9 } such that the digit has appeared before.



- The set of strings consisting of zero or more a's followed by zero or more b's, followed by zero or more c's.


a
b
c
ε, a
ε, b
ε, c
also written as:
ε
ε
regular expression: a* b* c*
Given A1 =
and A2 =
A1 | A2 =
A1
A2
A1 ; A2 =
A1*

2.6 Week 12

The objective of this week's homework was to explore the concepts of Mutual Exclusion with and without Atomicity in addition to the concepts of Sequential Consistency in Weak Memory through the use of online multi-threading Java Programs.

Homework Question 1: Have a look at peterson.py. What program behavior do you expect? Is your expectation confirmed when you run the program?

- Based on the program itself, I expect the program to attempt to run both threads a total of 10000 times. The execution of each thread would iterate the "counter" of the program each time the thread is run, so I would expect the print statement at the end of the program to note that the program's counter ended at 20000. This expectation was confirmed after running the program.

Homework Question 2: Analyse the Java program peterson in the same way as you analysed the Python program in the previous exercise. Make sure to run the Java program on your local machine. What observations do you make?

- Based on the program itself, I expect that the program, similar to the python program from the previous question, would attempt to run both threads a total of 10000 times as both the python and java programs share a similar code for what is defined as the "critical_section" in the python file and what is defined as a "process" in the java file. During the execution of each thread, the program would iterate the "counter" variable after each thread has completed their process, and because both threads are running the "process" function that iterates 10000 times, I would expect the print statement at the end of the program to note that the counter ended at 10000. This expectation was confirmed after running the program.
- In terms of the program's behavior when executed, the java threading program appeared to run slower than the python program.

Homework Question 3: Explain why the outcome $\mathbf{a} = \mathbf{0}$, $\mathbf{b} = \mathbf{0}$ is not sequentially consistent, but the other three outcomes are.

- The outcome $\mathbf{a} = \mathbf{0}$, $\mathbf{b} = \mathbf{0}$ is not sequentially consistent within the context of the threading model as defined by memoryModel as there exists no currently existing iteration within the program that would compute and read both **a** and **b** as **0** before they are overwritten by the values of **x** and **y**. Currently, getting both **a** and **b** as **0** could be possible without the resetting of all variables at the end of *runTest()*.

Homework Question 4: Report the results you get from running memoryModelWithStats on your local machine. Include the specs of your processor, in particular the number of cores. If you can find out something about the caches, add this as well.

- Processor Information:
  - Name: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz
  - Base Speed: 2.71 GHz
  - Number of Sockets: 1
  - Number of Cores: 2
  - L1 Cache: 128 KB
  - L2 Cache: 512 KB
  - L3 Cache: 3.0 MB
- Results from memoryModelWithStats :

- (0, 0): 0
- (0, 1): 538
- (1, 0): 461
- (1, 1): 1

Homework Question 5: You can force sequential consistency of memoryModelWithStats by declaring certain variables volatile. In general, declaring variables as volatile comes at cost in execution time, so we want to use this sparingly. Which variables must be declare as volatile to ensure sequential consistency? Measure and report the effect that volatile has on your run time (I use java Main — gnomon).

- In the memoryModelWithStats, the variables that needed to be declared as volatile in order to ensure sequential consistency were the x and y variables, as those variables are shared, used, and updated across both created threads. By declaring them as volatile, this helps ensure sequential consistency as declaring them as volatile applies the "lock" that the previous programs were missing, protecting them during the "critical section" of each respective thread.
- Setting the variables x and y had affected the runtime of the program as the original program had an initial runtime of 9.875962 seconds, whereas the modified code had a runtime of 11.0930762 seconds. As I had difficulty installing gnomon, I used the Instant and Duration Java packages in order to record the respective runtimes of the program.

3 Paper

...

4 Conclusions

(approx 400 words) A critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of software engineering? What did you find most interesting or useful? What improvements would you suggest?

References

[ALG] Algorithm Analysis, Chapman University, 2023.