Optional project report

1. Introduction

a. Model checking

Model checking is a way to verify the specifications or correctness of a system algorithmically. This is done by first converting the functionality of a software or hardware system into an abstract representation and verifying that the machine behaves as intended in every single state.

b. Applications

Model checking is especially useful for testing software or hardware which is too complex for testing manually. It is also almost essential in testing machines where failure can have a fatal negative consequence. Examples of such systems would be medical devices, self-driving cars, rockets or vulnerability testing for banks or ATMs.

2. Finite Automata

Finite automata are simple machines that can operate without any memory i.e. once an input is read, it is gone forever. A sequence of valid inputs on a finite automata are called strings and the set of all acceptable strings of a finite automata is the language of the finite automata.

a. Formal definition

The formal definition of a deterministic finite automaton includes the set of all its states, the input alphabet, the transition function which denotes the transition between states on a given alphabet, the initial or start state and all the accept or final states. It is formally written as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the set of ***accept states***.

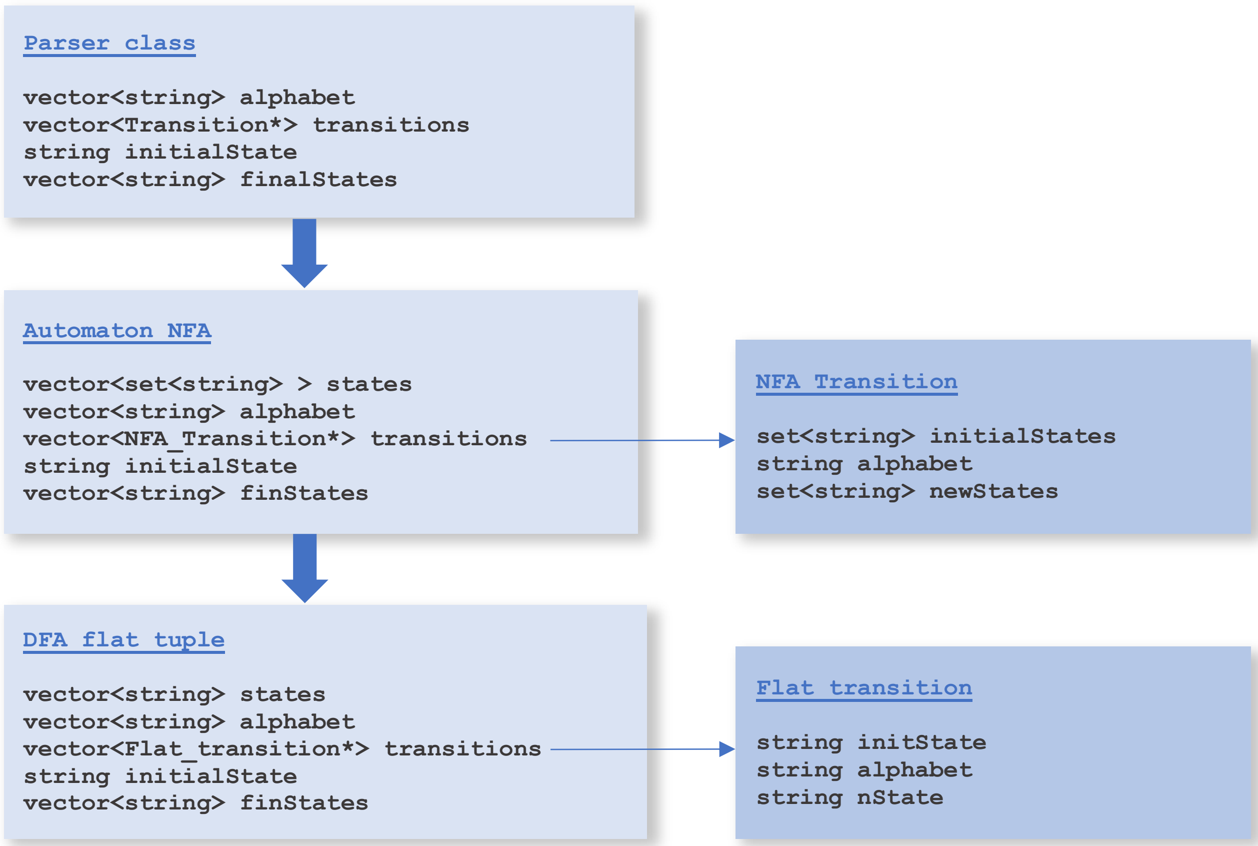
The main difference between nondeterministic finite automaton and deterministic finite automaton is the transition function. In an NFA, states do not have to have transitions for every alphabet; moreover, they can transition to multiple states on a single alphabet. NFA also allows transition over ε. Therefore, the transition for NFAs is defined as:

$$\delta : Q \times \Sigma_{\varepsilon} \rightarrow P(Q)$$

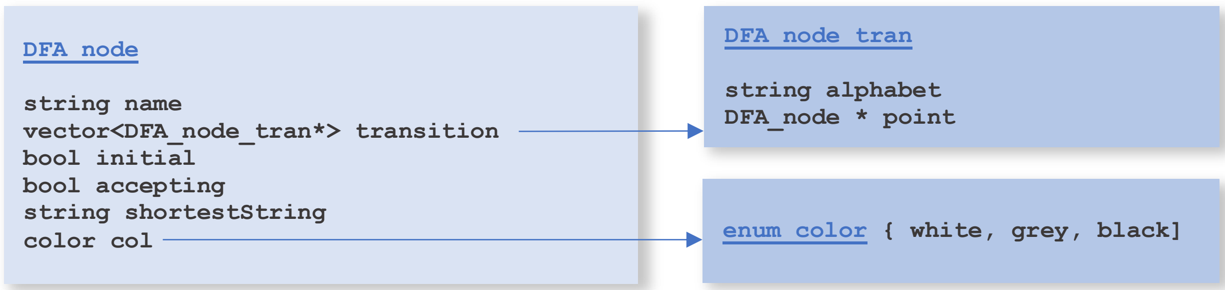
Where Q is the set of all states of the NFA and P(Q) is an element of the power set Q.

B. Data structure for representing finite automata

The Parser class saves the initial input since the input does not contain the information required to build an automaton. The Parser class is then used by the Automaton class to build a FA and convert it to a DFA from an NFA if required. Since NFAs and DFAs cannot be represented by the same structure, there is an intermediate structure where the states are elements of the power set of all states. The final output is a DFA tuple where all states are represented by strings for ease of use. This is the standard input for the rest of the functions and classes in the program.



`DFA_flat_tuple` is the structure that is used print a graph or to file. However the structure does not allow easy traversing which is needed to find the shortest string in the language. For generating graphs the following data structure is used in the program:



```
DFA_node

string name
vector<DFA_node_tran*> transition
bool initial
bool accepting
string shortestString
color col
```

```
DFA_node_tran

string alphabet
DFA_node * point
```

```
enum color { white, grey, black]
```

The `shortestString` represents the shortest string required to reach a given node and the colors are used to different status of breadth first search traversal.

3. Problems on Automata

a. Decidability of problems and their solution

The input to the program are two FA and since NFAs are equivalent to DFAs it does matter in this case what type of FA the input is. Since they are both FA we know that their language is regular. The next step is to generate the compliment to one FA. Since regular languages are closed the complement must also be regulat. After that we generate the intersection of the first FA with the compliment of the second FA and since regular languages are closed under intersection the resulting language is once again a regular language. The last step is to check if the language of the resulting intersected FA is empty and we know from Theorem 4.4 that E_{DFA} is a decidable language. Thus the whole problem is decidable.

b. Algorithms to solve decidable problems

1. NFA to DFA conversion

For NFA to DFA conversion I used Gallier's algorithm for conversion which was perfect for this case since it does not require you to detect whether the input is a DFA or an NFA. In the case that the input is an NFA the output is a DFA and if the input is a DFA the output is still a DFA. Additionally, the Gallier algorithm works by finding reachable state from a given state which means it automatically gets rid of all the unreachable states. The conversion from Gallier pseudocode to C++ was almost 1 to 1 but for the sake of ease I represented the empty set with a singleton containing the element Dead-State. This saved a lot of time in the rest of my functions since I did not have to check for an empty set. The full code is given as follows.

```
    //Implementing Gallier's NFA to DFA pseudo code
    set<string> initialStateSet;
    initialStateSet.insert(parser->initialState);
    dfa->states.push_back(initialStateSet);

    int total = 1, marked = 0;

    while(marked < total){
        marked++;
        set<string> S = dfa->states.at(marked-1);
        for(vector<string>::size_type z = 0; z != this->dfa->alphabet.size(); z++) {
            set<string> U;
            set<string>::iterator s;

            for(s = S.begin(); s != S.end(); ++s) {
                string state = *s;
                set<string> nStates = stateAndAlphaToNewStates(state, this->dfa-
              >alphabet[z]);
                U.insert(nStates.begin(), nStates.end());
            }

            //This is an extra addition for c++, if there are no outgoing transitions
            //from current state with given alphabet then generate a dead state. This
            //also takes care of self loop on the dead state
            if(U.empty()){
                U.insert(this->deadStateName);
            }

            if (!(isItAnElement(U))) {
                total++;
                this->dfa->states.push_back(U);
            }
            addTransitionsToDfa(S, this->dfa->alphabet[z], U);

        } //end for
    } //end while
```

2. Generating the compliment
(Intersector.cpp: line 35) Generating the compliment of a given DFA was relatively trivial. Since my representation of the DFA in code was similar to the 5-tuple, I had a list of all states of the DFA. I generated the compliment by finding all the states that were not in the final states vector and replaced the old final states with the new filtered states.

3. Making the intersection
(Intersector.cpp: line 51) The intersection function was relatively simple . It required taking one state from one machine and combining it with all of the states of the other machine and repeating the process. However, I discovered an additional step was required before this could be executed. The issue was that the intersection was generating non-unique states.

Name of state in FA_1	Name of state in FA_2	Resulting state name
1	23	123
12	3	123

I solved this issue by adding SP# at the end of every state in the specifications DFA and adding #SYS at the beginning. While this was an excessively thorough solution and a simple '#' symbol between the states would have fixed the issue, having the names of the FAs helped debug errors in the final output since I could see exactly which part of the state was generated by which machine.

New name in FA_1	New name in FA_2	Resulting state name
1SP#	#SYS23	1SP##SYS23
12SP#	#SYS3	12SP##SYS3

4. Generating (human) readable output

The final issue of the program was that output of the program was not easily interpretable. Some of the machines generated up to a 100 states and while I could read the output file manually, I wanted a visual representation of the output so I could easily track paths and confirm the shortest string or empty language of the final output. I achieved this by writing a function that would turn `DFA_flat_tuple` into its JavaScript object representation. I then turned the JavaScript object into an intractable graph using vis.js library. Since all four of the DFAs the system, specification, compliment of specification and intersection were represented by the same structure I could visualize my results at each step.

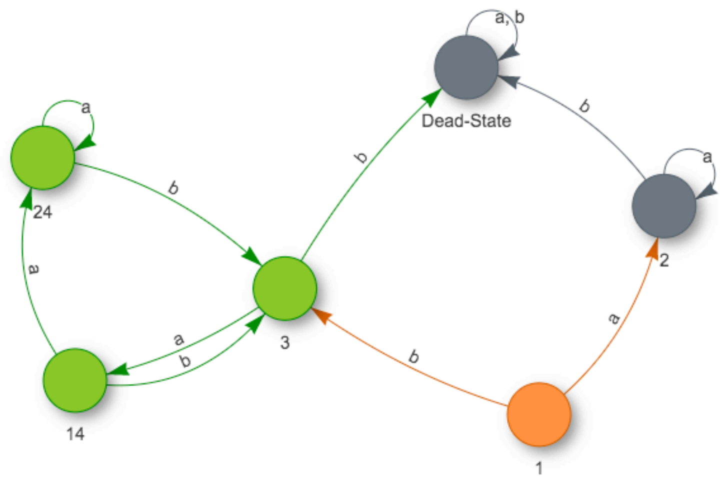


FIGURE 1. EXAMPLE OF A GRAPH GENERATED USING JAVASCRIPT

c. Complexity of algorithms

Most of the functions add, remove, search or update a vector of nodes thus run in a linear time $O(n)$. The other set of functions that take longer are as follows:

(Assuming both input FAs have equal size)
n *is the states of the given NFA*
d *is the states of the generated DFA*
m *is the set of alphabet*

Function	Run time	Description
setFinalStates	$O(d * m)$	Iterates over all states and checks if it's an element of the set of final states.
makeDFATuple	$O(2^n)$	Generates a DFA from an NFA. The DFA could potentially have the size the power set of states of the NFA thus the upper bound of 2^n
giveTuple	$O(d)$	Converts from the structure Automaton NFA to DFA_flat_tuple
makeShortestString	$O(d+m)$	Does a BFS of the dfa. Since complexity of BFS is O(nodes+edges) the run time is d+m since each node has m edges.
makeIntersection	$O(m*d^2)$	Generates the intersection of two FAs. The part that generates the transition function is a triple nested loop since for each state in first FA it iterates over all states of the second FA for each of the alphabet.

4. Application: infusion pump verification

The goal of this application is to verify that an infusion pump's system model satisfies given specifications. In order to make figures more readable, I have simplified the state names and transitions and replaced them with the following:

Number	State
1	Idle
2	Rate set (1)
3	Pumping at rate 1
4	Warn user that there is no fluid
5	Waiting for user to acknowledge completion
6	Bell rung
7	Rate set (2)
8	Pumping at rate 2

Alphabet	Transition
a	const_rate_1
b	reset
c	pump_fluid
d	fluid_empty
e	ring_bell
f	fluid_full
g	const_rate_2

The Orange node in the figures denote initial state and green state denotes accepting state. Orange node with green outgoing arrows and green outline means the state is both initial and accepting.

1. Does the pump P satisfy the specification S_1?

The specifications S_1 states the following:

$$S_1 = (a \cup c \cup b \cup e \cup f)^* (\varepsilon \cup de (a \cup b \cup c \cup e \cup f \cup d)^*)$$

This defines that as long as fluid is not empty then all states are acceptable but if fluid is empty then it has to immediately be followed by a ringing of the bell and which can then be followed by any sequence of states.

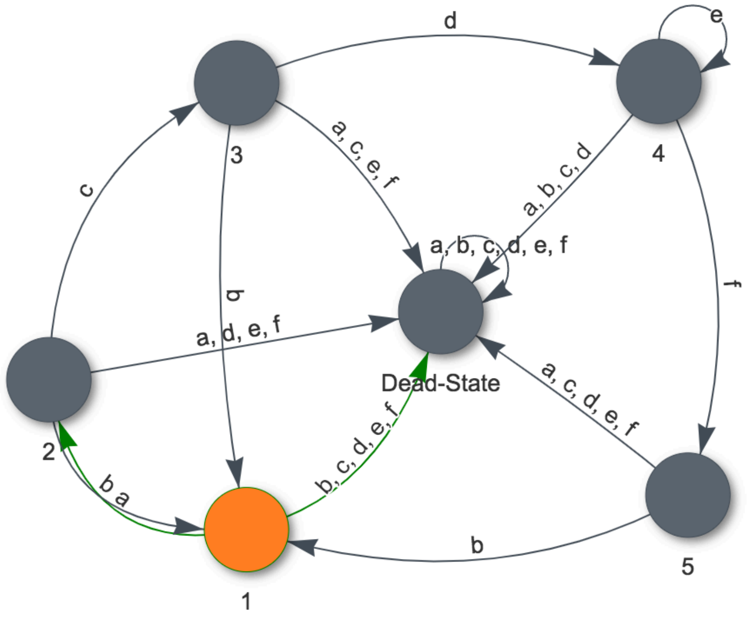


FIGURE 2. SHOWING THAT D DOES NOT HAVE TO BE FOLLOWED BY E WHICH IS AGAINST SPECIFICATIONS

a. Running the specifications and system model on code

It is obvious that there is a route from the initial state <Idle> to the accepting state that includes <fluid_empty> but not <ring_bell>. Running the specifications and system model yielded expected results which was that acdfb was in the language when it should not be.

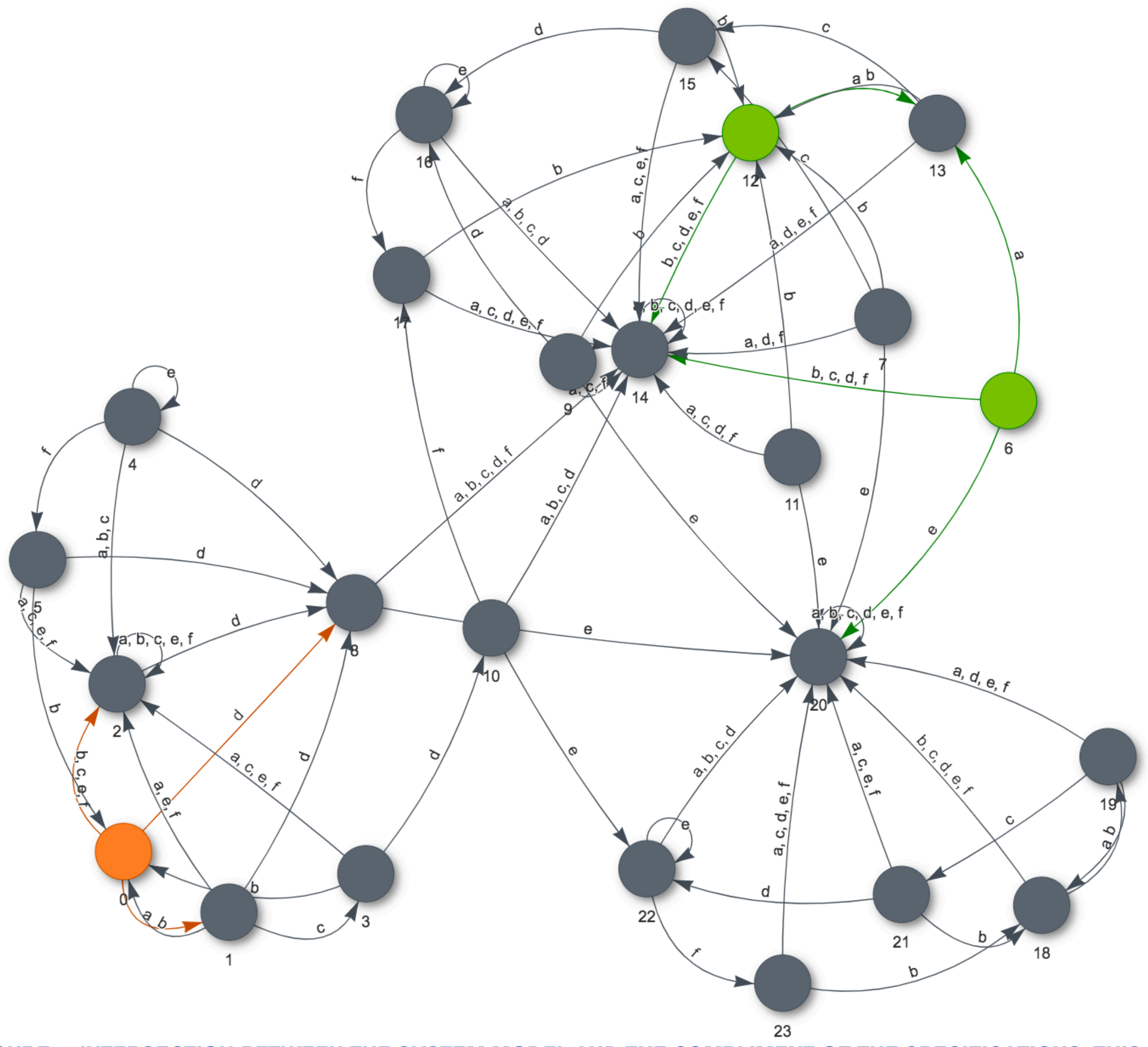


FIGURE 3. INTERSECTION BETWEEN THE SYSTEM MODEL AND THE COMPLIMENT OF THE SPECIFICATIONS. THIS LANGUAGE IS NOT EMPTY AND ACCEPTS THE STRING ACDFB.

b. Satisfying the specifications

The system does not satisfy specifications because <ring_bell> transition does not lead to a new state which makes the transition optional. To fix the system I added a new state after <fluid_empty> called <bell_rung> and <fluid_empty> only has one transition out of it going towards the <bell_rung>. This means that the only event that can be followed by <fluid_empty> is <bell_rung> and this is demonstrated by figure 4. Running the new system model on my code yielded an empty language which means that the model satisfies the specifications as also demonstrated by figure 5.

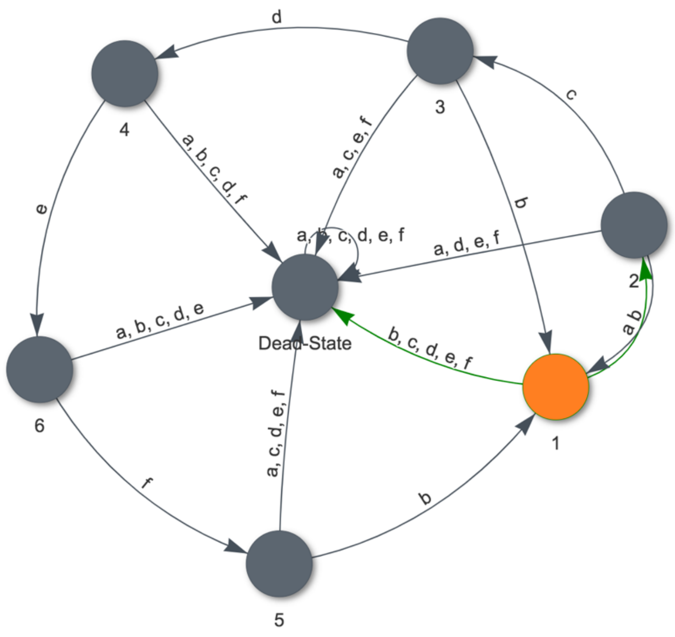


FIGURE 4. THE UPDATED SYSTEM MODEL DOES NOT HAVE A SELF LOOP ON STATE 4 ON ALPHABET E AND INSTEAD E LEADS TO A NEW STATE 6.

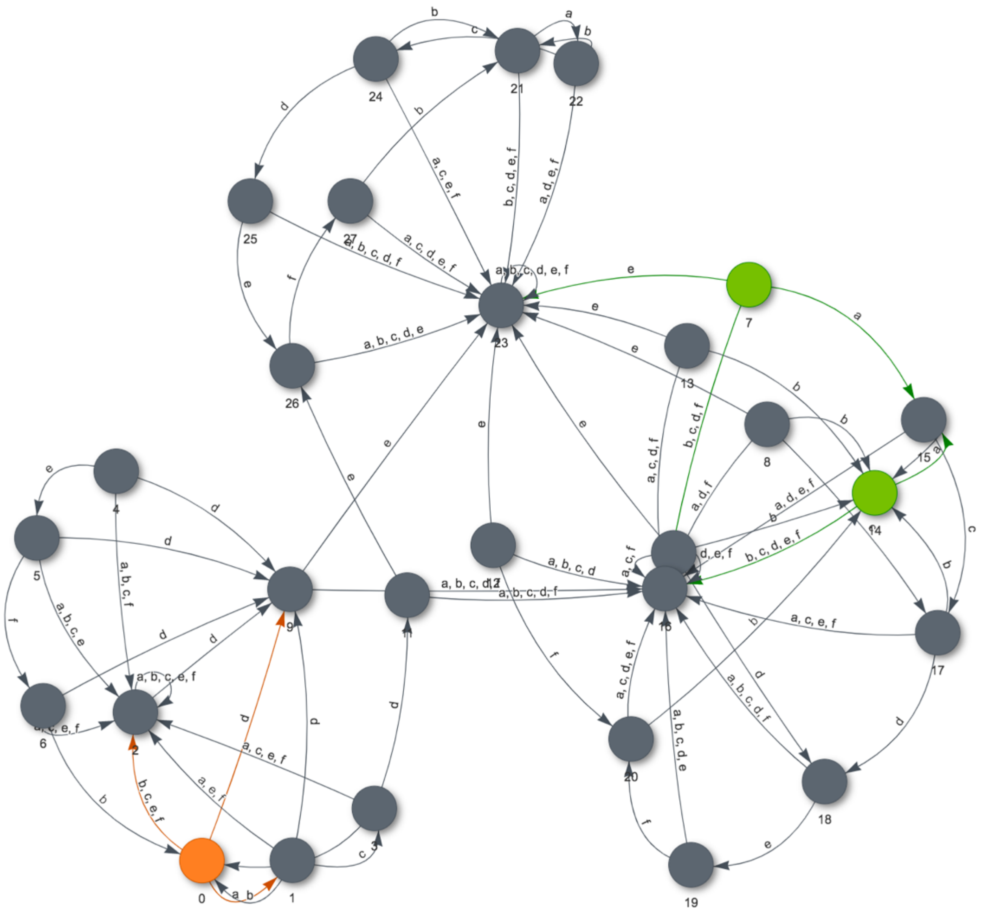


FIGURE 5. THE INTERSECTION WITH THE UPDATED SYSTEM MODEL SHOWS THAT ALL THE ACCEPTING STATES ARE UNREACHABLE WHICH MEANS THE SYSTEM SATISFIES THE SPECIFICATIONS

2. Extension of model to allow one more fluid flow rate

a. Modified model P`.

For the new modified P` I added two additional states <Rate_set> and <Pumping at rate 2> and a new alphabet <const_rate_2>. Since the states do not state what is the rate of the flruid that is being pumped after <fluid_empty> occurs, I did not add any more additional states to the machine.

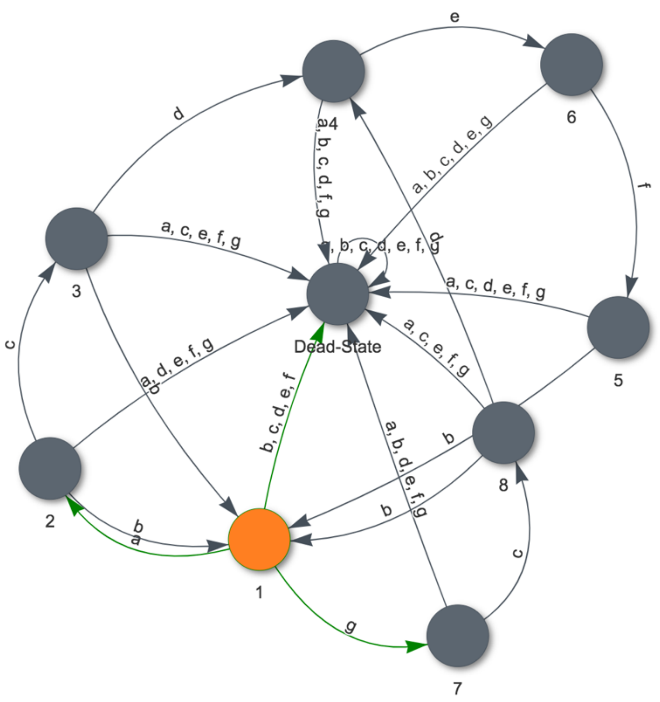


FIGURE 6. THE SYSTEM MODEL SHOWS P` WITH TWO NEW TRANSITIONS AND ONE NEW ALPHABET.

b. Updated specifications

The new updated specifications S_2 state that if a <const_rate_1> or <const_rate_2> occurs then eventually <reset> occurs which is captured by the following regular expression and visualized in figure 7.

$$S_2 = (b \cup c \cup d \cup e \cup f)^* \, (\varepsilon \cup ((a \cup g)(a \cup c \cup d \cup e \cup f \cup g)^* b(a \cup b \cup c \cup d \cup e \cup f \cup g)^*))$$

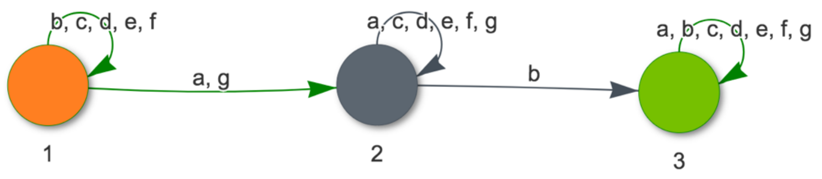


FIGURE 7. SHOWS THE UPDATED SPECIFICATION AUTOMATON.

c. Confirming that P` satisfies S_2

My code indeed confirmed that the new system model P` satisfies S_2. The empty language of the new intersection generated by my code is visualized in figure 8.

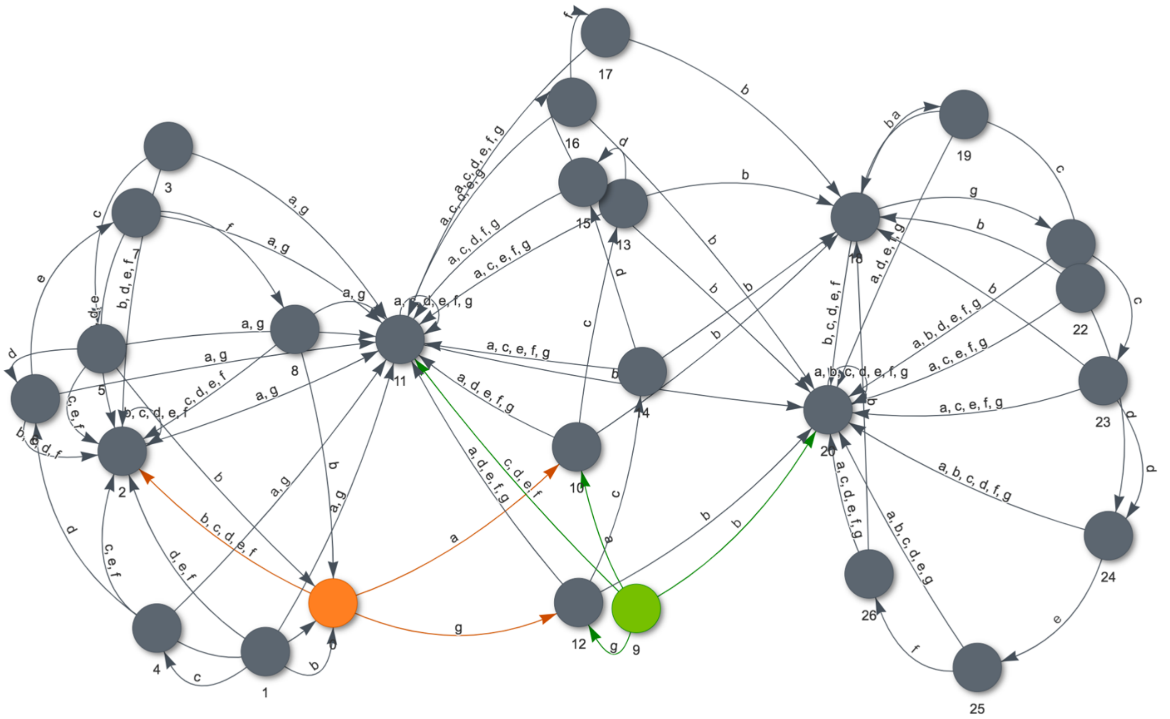


FIGURE 8. SHOWS THE LANGUAGE THAT IS THE RESULT OF P` INTERSECTED WITH COMPLIMENT OF S_2 IS EMPTY SINCE THERE ARE NO INCOMING EDGES ON THE ACCEPTING(GREEN) NODE.

5. Conclusions

Even though converting a system model to an abstract state representation can be a daunting task especially for larger models, it can be very rewarding. The Grammar-based whitebox fuzzing paper highlights this fact by using this type of model verification and increasing code coverage from 53% to 81% while using three times fewer tests.

Finite automata is a relatively older concept yet it is still very relevant and new changes are regularly proposed in order to solve modern problems with it. For example, new type of automata TBA has been proposed to model the behavior of finite-state asynchronous real-time systems. These models accept languages in which each event has an associated real-valued time occurrence.

In today's age where software grows overtime at an exponential rate it has become too difficult to manually write test cases that have a high code coverage. Even then, the common techniques for testing software do not always work with multi-threaded software. It is essential to have a good software verification model in order to write software for new applications like self-driving cars and this type of model can not only help us verify the software but also help generate code templates so the software has a robust foundation.

References

Aarts, F., Ruiter, J. D., & Poll, E. (2013). Formal Models of Bank Cards for Free. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. doi:10.1109/icstw.2013.60

Alur, R., & Dill, D. (n.d.). Automata for modeling real-time systems. *Automata, Languages and Programming Lecture Notes in Computer Science,* 322-335. doi:10.1007/bfb0032042

Center for Devices and Radiological Health. (n.d.). Infusion Pumps - Infusion Pump Software Safety Research at FDA. Retrieved from https://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm

Godefroid, P., Kiezun, A., & Levin, M. Y. (2008). Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices, 43*(6), 206. doi:10.1145/1379022.1375607