DFA Reader/Tester

A Project by

Jonathan Z. Canto
Thad Benito Tangangco

Submitted to

Luisito L. Agustin
Instructor, CE 160

In Partial Fulfillment of the Requirements for the Course
CE 160: Automata and Formal Languages

Department of Electronics, Computer, and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

October 2003

ABSTRACT

A program to simulate DFAs which are read from an input text file was designed using the Visual C++ language. The input text file specifies the DFA title, alphabet, as well as the starting and accepting state/s, and is accessed by filename path. This input file has a specified format patterned after the DFA transition table that makes it easy to follow transitions through the input string. The program makes use of CStrings, which are strings defined in Visual C++, to store the transition table, which then allows determinination of transitions for that DFA based on the input string. It also includes a Trace function, which displays the current state of the DFA after consuming a portion of the input string. The program is able to function as a DFA reader/tester by reading from a text file and transitioning from state to state. Sample input text files have shown the program to be fully functional for single or multiple character state names.

ACKNOWLEDGEMENTS

We would like to thank our teacher Mr. Lui Agustin for giving us a great semester in CE160. We would have never gotten this far without his expert guidance. Credit must be given to both Sir Tris Calasanz and Sir Carlos Oppus for letting us use the instrumentation laboratory day in and day out. Our thesis adviser Sir Monje also deserves accolades for putting up with us for so long. Sir Amparo also deserves thanks for all the help and advice he has given us regarding this project and many others. Finally kudos to everyone else that's persevering through the 5th year for Computer Engineering. May God bless us all in our endeavors and may we all graduate happy and lively.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Deterministic Finite Automata

1.1.1 Definition

Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from Q $\times$ Σ to Q , let q0 be a state in Q and let A be a subset of Q. We call the elements of Q a state, δ the transition function, q0 the initial state and A the set of accepting states.
Then a deterministic finite automaton is a 5-tuple (Q , Σ, q0 , δ, A).

Notes on the definition:

1. The set Q in the above definition is simply a set with a finite number of elements. Its elements can, however, be interpreted as a state that the system (automaton) is in.

2. The transition function is also called so because the automaton moves into the state δ(q, a) if it receives the input symbol a while in state q. The transition function is denoted by the symbol δ. Thus for each state q of Q and for each symbol a of Σ, δ(q, a) must be specified.

3. The accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ceases to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted.

4. A deterministic finite automaton is also called simply a "finite automaton". Abbreviations such as FA and DFA are used to denote deterministic finite automaton.

1.2 Scope and Limitations

The software can accept the parameters of a DFA from an input text file. It recognizes the start and accepting states along with the corresponding truth table of this DFA based on the specified file format. State names can be of single or multiple character length.

One limitation of the software is that it can only read lines of text a maximum of 80 characters long, so if the text file containing your DFA has a line that exceeds 80 characters long , the software will not be able to read this line of text correctly and may exhibit erratic behavior or not function at all.

The text file containing the DFA must strictly follow the specified format in order for the program to work properly. Non-compliance with this format will automatically mean that the text file will be read differently and result in a misrepresentation of the DFA to the program.

CHAPTER 2

INPUT FILE FORMAT

The text file that the program accepts for input must follow a specified format so that the software can correctly read the information contained within. Having a specific format to follow allows the software to do without a complicated parser that will increase code length and software complexity. The text file containing the DFA's transition table must follow this format:

Title of the DFA
alphabet
startstate
accepting states
transition table %
transition table %
transition table %
…

The text file that will specify the DFA specified above is as follows:

Strings of 0,1 with at least one 0
01
q0
q2 &
q0 q2 q1 &
q1 q2 q1 &
q2 q2 q2 %

Note that there is a terminating character at the end of the lines that specify the accepting state(s) and the DFA's transitions. This can be any character that the user wishes to put, may it be a '&', '%' or even a number like '0'. The software is programmed to read the line of text only until before the last character that appears on that line. Since there is no parser in the program, the software will read the lines of text word by word, where each word is separated from the next by a single space.

CHAPTER 3

SOFTWARE DESIGN

The program was written purely in Visual C++ and is fully functional on any Microsoft Windows operating system from Windows 95 or later. No special functions, header files, or user-customized classes were used. Aside from the text file containing the DFA, no external files or DLL's are required for its usage. This is purely a stand-alone program that only requires a text file specifying a DFA as its primary input.

The actual code of the software is very simple and easy to code/debug because, as mentioned earlier, no user-customized classes were used. In fact, the software does not even use (or need) a parser for reading from the input text file. This simplifies the code and drastically shortens code length. Because no specialized classes were used for the software it is very easy to debug and modify.

3.1 How data is stored internally

The key to the program's simplicity is its way of storing data internally. Instead of using classes or even the standard tree structure which is the widely accepted means of implementing DFAs, the software stores the DFA's transition table inside a two-dimensional array of CStrings. This is more efficient than using a recursive tree structure and custom classes because the CString class is already a well-defined and easy-to-use class structure that comes complete with help files and technical support from Microsoft via through their MSDN library. Using an array of strings precludes having to use complicated recursive functions that are very CPU-intensive and relatively hard to debug.

The motivation behind using a two-dimensional array is not just its simplicity, but its very obvious correspondence to an actual transition table. A transition table contains rows and columns that have as contents of their cells the names of the different states, the alphabet (or different inputs possible), and finally the states that each state will transition to given a specific input. The two-dimensional array of strings is actually also a table in that it has rows and columns of strings (or cells). This means that we can store all the information described in a DFA's transition table inside this array of strings.

The program uses the array **state [x] [y]** to store the transition table of the DFA it reads from the text file. The row is specified by the first number **x** while the column is specified by the value **y**. Each state and its transitions are stored in a single row. So state **q0** that transitions to **q2** on an input 0 and **q1** on an input 1 will have the following entry in the array:

[**q0**] [**q2**] [**q1**]

Here **state [0] [0]** holds the state's name while **state [0] [1]** holds **q2** which is the state **q0** will transition to on the first input in the alphabet. The alphabet itself is stored as a string which is itself an array of individual characters. The string is **alphabet [z]** where z is used to specify a specific character inside the string. Each of the possible inputs of the DFA's alphabet is stored as a character inside the **alphabet** string. The start state is stored in its own string **start** while the accepting states are stored inside a one-dimensional array of strings called **accept [a]**.

	0	**1**
->q0	q2	q1
q1	q2	q1
***q2**	q2	q2

Now for a DFA with the above transition table, the program will store this information in the following manner:

alphabet:	**“01”**		
start:	**“q0”**		
accepting[z] :	**[q2]**		
state [x] [y]:	**[q0]**	**[q2]**	**[q1]**
	[q1]	**[q2]**	**[q1]**
	[q2]	**[q2]**	**[q2]**

Having all the data stored as strings (or arrays of strings) is very convenient because we can use all the common and very useful functions of the CString class provided with Microsoft Visual C++. We can easily compare data using the string compare functions as well as add/replace entries inside the array. It is also very easy to lookup data inside the string arrays because we can easily perform a search through each row or column with just a single loop.

3.2 The Transition Function

The actual code for the transition function (along with comments) is as follows:

```
CString Transition(CString inpts ,CString  alph )

//This is the actual transition function that for our DFA reader/tester.
//      This function accepts as arguments the current character or input and
//      the string that contains the alphabet of the DFA. It will return the next
//      state the DFA must transition to.
{
      int templ = alph.GetLength();                  // store the length of the alphabet in
templ
      int cx=0;                        //       \
      int      dx;                     //        \
      int index = 0;                   //                these variables are all used as
counters
      int      bx;                     //        /
      int ctr=0;                       //       /
      int found=0;
      CString rets;                    // the return value of the function will be stored
                                       // here in rets

      // this loop will look at the current state (which is a universal variable) and look //
      // up the two-dimensional state array to find its row there
```

```
while (ctr<255)
        {
                if (state[ctr][0]==current)
                {
                        index=ctr;
                        found=1;
                        break;
                }
                ctr++;
        }

        // this loop will then match the input character against the alphabet to find out
        // which column of the two-dimensional state array the function should look up

        while (cx<templ)
        {
                if (alph[cx]==inpts[0])
                {
                        dx=cx;
                }
                cx++;
        }
        bx=dx+1;

// The function will now use the column and row addresses to lookup the state array
to // // find  out what is the next state. Once this next state is what will be returned by
the //
// function.
        rets=state[index][bx];
        return rets;
}
```

Basically this transition function first takes the input character then matches it with a character inside the **alphabet** string. Once it matches the input with a character in the alphabet it knows now what column it should look under inside the **state** array. The current state the DFA is in determines what row in the **state** array contains the destination state. Since the software now knows the current state and the input, it simple has to look up **state [row] [column]** to find out what state it should transition to. The string specified inside **state [row] [column]** is then returned by the function.

CHAPTER 4

HOW TO USE THE SOFTWARE

4.1 The Graphical User Interface

Programming the DFA Reader and Tester in Visual C++ allowed the program to use window-based forms to provide an intuitive and user-friendly interface. There is only one main form window within which the user may access all the functions and features of the program.

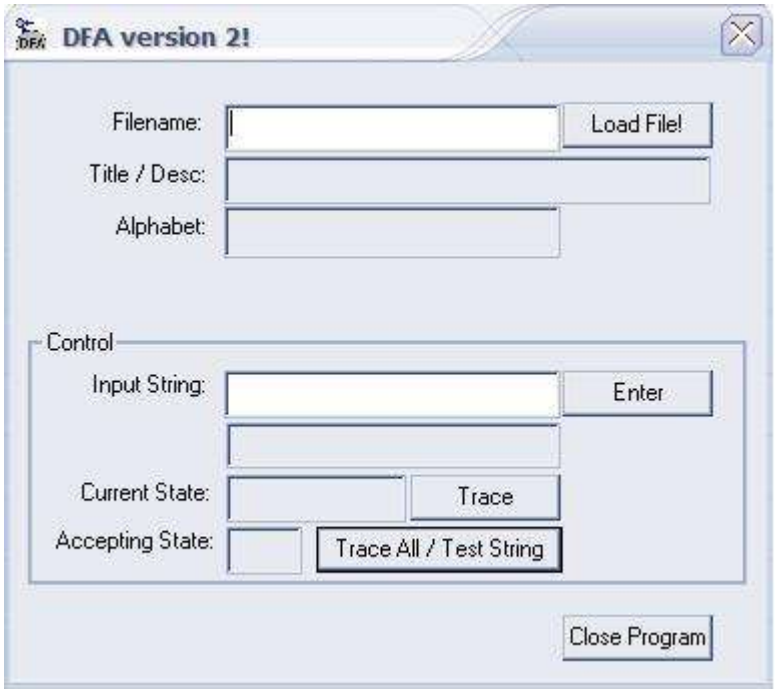

Figure 1. GUI Main Window

From this window the user may load a DFA from a text file using the provided **Filename** edit box. The user may also enter input strings inside the **Input String** field and then either trace through the DFA using the **Trace** button or test the entire string using the **Trace All / Test String** button. Tracing through the input string consumes one

input at a time, and the consumed inputs are shown in the field just below the **Input String** field. If the current state is an accepting state, then the **Accepting State** field will be marked “Yes”. If not, then it will be marked “No” instead. Testing the entire string will make the program trace through the input string. If the final state is accepting or not accepting it will notify the user with appropriate error message.

Figure 2. Accept field “Yes”Window

Figure 3. Accept field “No” Window

For the users quick reference, a brief guide on the software’s operation is featured in the **About** window of the software. The user can access this by selecting “About DFA” inside the software window’s dropdown menu.

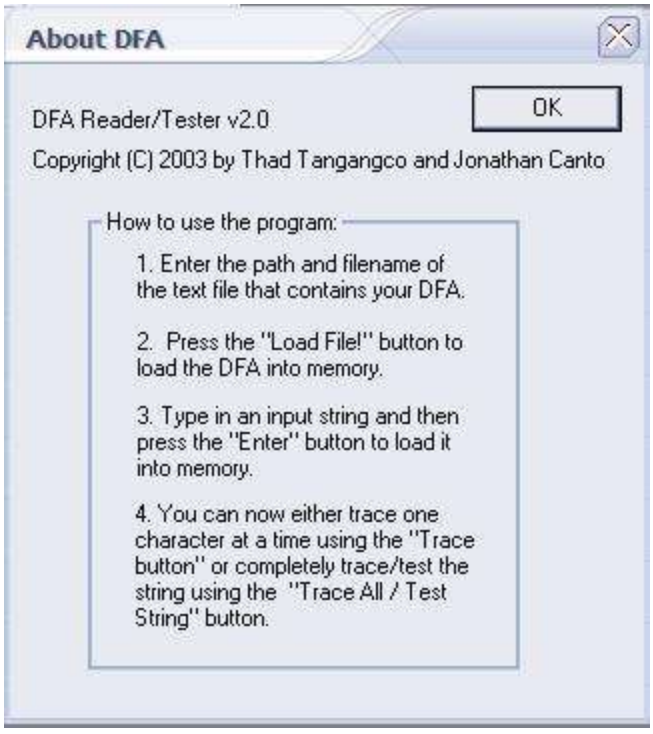

Figure 4. Software About Window

BIBLIOGRAPHY

1. Definition of Deterministic Finite Automata, College of Sciences, Old Dominion University, http://www.cs.odu.edu/~toida/nerzic/390teched/regular/fa/dfa-definitions.html