The COBOL DFA Tool

Ronald Finkbine
rfinkbin@ius.edu
Computer Science Indiana University Southeast
New Albany, Indiana 47150, USA

Abstract

Common today in Computer Information Systems (CIS) education programs are two types of analysis for software development, top-down and object-oriented. Often overlooked is state transition analysis. In an effort to introduce students to the concept of deterministic finite automata, an advance programming in COBOL course completed a final project of building a DFA tool in COBOL.

Keywords: deterministic finite automata, DFA, FA, finite state machine, FSM

1. INTRODUCTION

Most courses in Computer Information Systems (CIS) academic programs use either top-down or object-oriented analysis as the main technique for instructing students in analyzing requirements and determining software architectures. Business-related programming textbooks generally utilize the COBOL programming language and common types of programming assignments such as generating reports and updating files. This type introduction might be realistic to applications from the real world of the past, but current technologies such as client/server require that CIS students be trained more like standard CS students.

There are many names for Deterministic Finite Automata (DFA); state machines, finite state machines, Moore machines and Mealy machines (Samek 2002). These variations of the DFA vary in minute characteristics but they are essentially equivalent, which is to specific rule-based behavior (O'Byrne 2003) (Weeks 1992). This machine concept has traditionally been useful for construction of small software systems with requirements for extremely high reliability (Harel 1997) such as vending machines. The concept of the DFA is to specify computational behavior of a system by determining the acceptance (yes or no) of an input string. A computer program always has a state, the specific set of values in all variables. A DFA specifies the progression of a program through its various states while processing an input string resulting in a yes/no answer.

The high-reliability characteristic of embedded systems would be a desirable quality to add to all types of software including business-related systems. The DFA concept comes very close to the concept of 'proof' as in "show that your program solves this problem and does nothing else". The use of the DFA concept allows the programmer to write programs that are *near-provable*. The definition of the term *near-provable* depends upon acceptance of diagrams as proofs, a current research area in the area of applied mathematics [7].

This paper discusses a DFA tool for COBOL, written in COBOL; that was assigned to a sophomore-level programming project in a CIS program.

2. ASSIGNMENT

The assignment is in two parts and part one to develop an elementary code generator

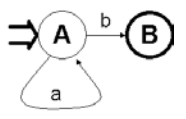

Figure 1: DFA

that accepts a text file specifying the behavior of a DFA. Part two is to write a DFA machine to solve a simple problem. Before discussing the specifics of the assignment the concept of the DFA needs to be explained.

3. DFA DEFINITION

Figure 1 displays the conceptual DFA and specifies the behavior of the computation and these notations require explanation. A DFA consists of a finite number of states (circles) (Nisley 2002) and transitions (arrows) (Harel 1987). The doublewide arrow entering from the left is the start symbol and every DFA must have a single start symbol to denote the beginning of computation. There are two states noted with capital letters and they represent a position in the computation. The state **A** can be described as the *seen-zero-or-more-letter-A-state* and the machine will stay in this state while the input string is processed one character at a time and each character is an **a**. The transitions in this DFA are label by two characters (**a** and **b**) and they are the alphabet. As the input string is processed all letters **a** are processed by the state **A**. When a letter **b** is encountered the DFA will move to the **B** state. Note that all input characters must be in the alphabet (no letters **c** or **d**) and there is no way to loop and accept multiple letters **b**. Each acceptable string will have only one **b** at the end of zero or more letters **a**. Examples of this type of acceptance/rejection behavior abound. A first example is a vending machine accepts a string of one quarter for a can of soda (a very inexpensive one!) and would also accept a string of five nickels. Both of these input strings would get to the acceptance state of *seen-25-cents* and issue a can of soda. A second example would be any type of password security system that depends on a Personal Identification Number.

In Figure 1, the state **B** is the *seen-one-B-state* and is also an acceptance state due to its dark border. In transitioning from state **A** to state **B** one input character was read from the input stream. Each DFA must have at least one acceptance state and, to reach this state, a DFA must complete its last state change while at the same time exhausting the input string. The set of strings (language) accepted by this DFA is: any number of **a**'s followed by one **b**. Each transition from state to state within the DFA must process one character from the input stream. The overall goal is to have the DFA in an accepting state at the time of exhaustion of the input string. Exhaustion of the input string while in a non-accepting state will cause the machine to produce an error.

4. COBOL DFA TOOL

This section describes a COBOL DFA generator. The input to this program is a text file similar to Figure 2 and its output is a text file that contains a COBOL program that is syntactically correct and will implement the state machine as described in the DFA text file. This produced COBOL program will implement the DFA described. Figure 2 describes a DFA whose start state is **A**, the acceptance state is **B**, the name of the to-be-produced program is DFA.COB, the name of the file to be input to DFA.COB is IN.TXT, and the name of the file to be produced by the execution of DFA.COB is OUT.TXT.

The DFACODE section is for any additional COBOL code that needs to be included to complete the program such as addition data structures or procedures. The DFASTATE section specifies the behavior of the DFA and

```
DFACONTROL
        START A
        ACCEPT B
        OUTPUT-COBOL "DFA.COB"
        INPUT-DATA "IN.TXT"
        OUTPUT-DATA "OUT.TXT"
DFACODE
F-BEGIN.
        DISPLAY "IN F_BEGIN"
F-END.
        DISPLAY "IN F_END"
DFASTATE
        A       A       "a"     F-Begin
        A       B       "b"     F-End
```

Figure 2: Sample DFA file

Figure 2 implements the DFA from Figure 1. Go from state **A** to state **A** when the input character is an **a** and execute function / procedure / paragraph **F_begin**. Go from state **A** to state **B** when the input character is a **b** and execute function / procedure / paragraph **F_end**.

```
F-BEGIN.
          DISPLAY 'IN F-BEGIN'
F-END.
          DISPLAY 'IN F-END'
MAIN-PARAGRAPH.
          MOVE 'A' TO CURRENT-STATE
          MOVE 'NO ' TO INPUT-EMPTY
          READ INPUT-CHAR AT END MOVE 'YES' TO INPUT-EMPTY
          PERFORM UNTIL INPUT-EMPTY = 'YES'
                    IF CURRENT-STATE = 'A' AND INPUT-CHAR = 'a' THEN
                              MOVE 'A' TO CURRENT-STATE
                              PERFORM F-BEGIN
                    ELSE IF CURRENT-STATE = 'A' AND INPUT-CHAR = 'b' THEN
                              MOVE 'B' TO CURRENT-STATE
                              PERFORM F-END
                    ELSE IF NOT ((INPUT-CHAR = 'a') OR  (INPUT-CHAR = 'b')) THEN
                              MOVE 'ERROR' TO CURRENT-STATE
                              DISPLAY "BAD INPUT CHARACTER"
                    ELSE
                              MOVE 'ERROR' TO CURRENT-STATE
                              DISPLAY "UNSPECIFIED STATE/TRANSITION"
                    ENDIF
                    READ INPUT-CHAR AT END MOVE 'YES' TO INPUT-EMPTY
          END-PERFORM
          IF CURRENT-STATE = 'B' THEN
                    DISPLAY 'ACCEPTED'
          ELSE
                    DISPLAY 'REJECTED'
          ENDIF
```

Figure 3: Sample Generated COBOL Program

The produced-DFA-COBOL program will only allow input characters of the alphabet **a** and **b**. Any other characters input will produce an answer of not accepted.

The produced-DFA-COBOL program will execute until the input string is exhausted. This will cause the program to complete its input phase and then to check the current state. If the program is in an acceptance state then the input string is accepted (yes) else the input string is rejected (no). Figure 3 displays the COBOL code that is produced by the processing of the Figure 2 DFA file. Note the beginning code handles the start state of the DFA. The ending code processes the acceptance state as specified in Figure 2. There can be more than one acceptance states. The code for both the input and output files is straightforward and omitted from Figure 3. The main perform loop executes until the end of the input is reached and inside the loop must be an if-then execution path for every state-transition combination from the DFA file.

Appendix A of this paper includes the DFA and a detailed explanation for the fox, chicken and grain problem.

5. SUMMARY

The results of this project were very good in that all of the students were able to complete the group project and were able to answer simple questions regarding the concept of the DFA on the final exam. A small number of assignments were completed (in class and take home) using the state transition method such as: the fox, chicken and grain problem; the vending machine design problem; and the infix to postfix notation conversion problem.

This tool shows to the CIS student that there are alternatives to the top-down and object-oriented methods of system analysis. The most challenging concept to convey to the student is a COBOL source file is just data that another executable program (the compiler) can read, transform and execute.

Work is continuing on this tool with the intent to make it available to the educational community. Related research is ongoing in an effort to determine the computability of transforming non-DFA-based programs into DFA programs.

6. REFERENCES

Anderson, Michael, (2003), "Diagrammatic Reasoning Website," http://zeus.cs.hartford.edu/~anderson/

Harel, David, (1987), *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, North-Holland.

Harel, David and Eran Gery, (1997), *Executable Object Modeling with Statecharts*, IEEE Computer , July.

Nisley, Ed, (2002), *State of the Machine*, Dr. Dobb's Journal, December.

O'Byrne, Brian, (2003), *State Machines and User Interfaces*, Dr. Dobbs Journal, January.

Samek, Miro, (2002), *Practical Statecharts in C/C++*, CMP Books, 1-57820-110-1.

Weeks, Kevin, (1992), *States and the Art*, PC Techniques, Oct/Nov.

APPENDIX A

This appendix details the fox, chicken and grain problem mentioned in the text of this paper and is stated in Figure 4. A DFA to solve this problem is displayed in Figure 5 with *state-1* as the start state and *state-8* and *state-error* as accepting states (bold border).

A man is crossing a river on the way to market with a chicken, a bag of grain and a fox. If left unattended the fox will eat the chicken, and the chicken will eat the grain. The boat will only hold the man and one of these at a time. Your task is to work out a sequence of crossings that will affect a safe transfer of the man, the fox, the chicken and the grain safely across the river.

Figure 4: Text of Fox, Chicken and Grain Problem

Each legal, reachable, safe state is numbered 1 through 10. Reading the notation on *state-1* shows the man, fox, chicken and grain on the left side of the river (see legend) and this is a safe state since the man is there with the remainder of the items.

The transitions between states show the necessary inputs to solve the problem. The setup is: you are to write a computer program that will read an input string from a file or the keyboard that will propose a sequence of moves, your program with either accept or reject this string.

To read Figure 5, the DFA can transition from *state-1* to *state-2* if the man takes the chicken across the river. Both *state-1* and *state-2* are safe states because the fox is never alone with the chicken (the man is on the same side of the river) and the chicken is never alone with the grain.

Note that the man can move the chicken from *state-2* to *state-1*, back across the river and still remain in a safe state. The man is not always required to make progress in solving the problem for the DFA to remain in a safe state.

From *state-2* to *state-3* the man leaves the chicken on the other side of the river and rows back alone.

From *state-1*, the DFA allows transitions for either an illegal character or a legal character that will put the DFA into an unsafe state into the *error-state*. This same transition should be on every other state (numbered 2 to 10) but is not actually put on the DFA figure in an effort to not clutter the picture.

If you use the letter *e* to represent the trip across the river when the man takes nothing with him, a legal input string to solve the problem would be: *CeFCGeC*.

Translating into English, the input string represents:

1. man and chicken cross river
2. man returns with empty boat
3. man and fox cross river
4. man returns with chicken
5. man and grain cross river
6. man returns with empty boat
7. main and chicken cross river

This is one of the two shortest strings that will be accepted by this DFA.

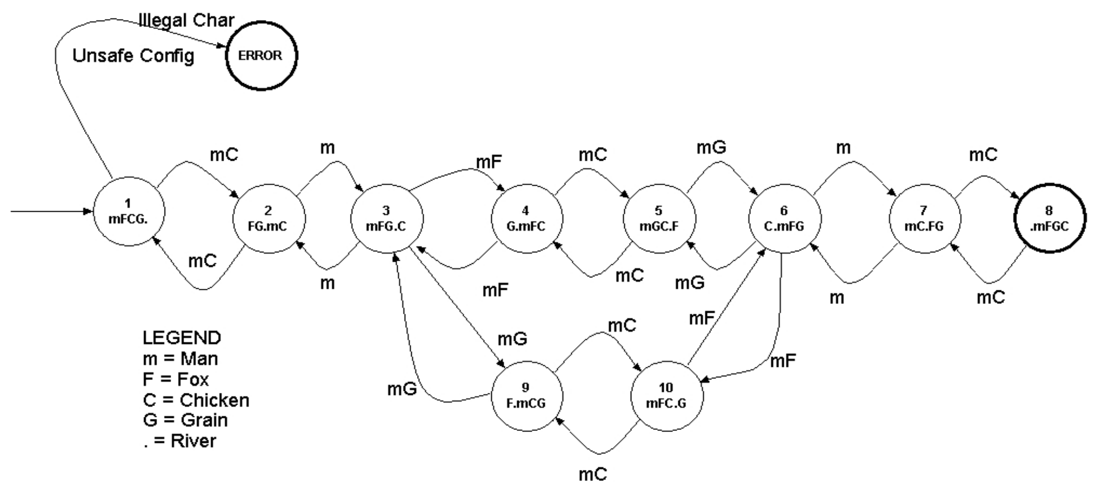

Figure 5: DFA for Fox, Chicken and Grain Problem