Simulations of Finite-State Automata Using Java 6.0
Part Two: Advanced Topics

Nicholas J. DeLillo

Nicholas J. De Lillo is Professor of Mathematics and Computer Science at Manhattan College where he has taught courses in computer science, computer engineering, and software engineering at both the undergraduate and graduate levels for over thirty years. In addition, Professor De Lillo regularly teaches courses in the masters program in computer science here at Pace. He is on also on the Editorial Board of Technical Reports.

Professor De Lillo is the author of numerous research papers and textbooks in mathematics and computer science. His textbooks include Advanced Calculus with Applications (1982); Computability with Pascal, co-authored with John S. Mallozzi (1984); A First Course in Computer Science with Ada (1993); Data Structures with C++, co-authored with John S. Mallozzi (1997); Object-Oriented Design in C++ Using the Standard Template Library (2002); Object-Oriented Design in Java Using java.util (2004); and Data Structures Using Java 5.0 (2006).

Professor De Lillo holds a B.S. in mathematics from Manhattan College, an M.A. in mathematics from Fordham University, and the Ph.D. in mathematics from New York University, where he was a student of Martin Davis's.
SIMULATIONS OF FINITE-STATE AUTOMATA USING JAVA 6.0
Part Two: Advanced Topics

Nicholas J. DeLillo
Professor of Mathematics and Computer Science
Manhattan College

1. Abstract.

In [1], we simulated the actions of a single deterministic finite-state automaton (DFA) using Java 6.0. Specifically, we applied the idea of enumerated types to simulate the alphabet, states, and transitions of the underlying DFA. In the present paper, we use more advanced features of Java 6.0 to first produce a more flexible version of the DFA simulation given in [1], and then generalize this construction to simulate the activity of any DFA. In each case, we will provide an interactive version of each such simulator, leaving versions whose specifications appear in predefined text files as topics for further study.

2. A Generalization of the DFA of [1].

We assume all of the preliminary definitions and terminology of [1]. In particular, we assume the specifications of the DFA named M. More precisely, we assume the definitions of the enumerated types

```java
enum States {INITIAL, DIGIT, HYPHEN, NOGOOD};
```

and

```java
enum Alphabet {SLASH, DASH, OTHER};
```

to represent the set of states and the alphabet, respectively, of M.

Since we cannot input programmer-defined enumerated type values directly, we instead interactively input the `char` values 'I', 'D', 'H', and 'N' to represent `INITIAL, DIGIT, HYPHEN, and NOGOOD`, respectively. We may then use the method

```java
public static States TheState(char ch)
{
    States st = null;
    switch(ch)
    {
    case 'I': st = States.INITIAL;
        break;
    case 'D': st = States.DIGIT;
        break;
    case 'H': st = States.HYPHEN;
        break;
    }
    return st;
}
```

As before, boldfaced text are used to denote Java keywords.
break;
case 'N': st = States.NOGOOD;
} // terminates text of switch
return st;
} // terminates text of TheState

Informally, we may view TheState as a function of a single char-valued variable that converts the values 'I', 'D', 'H', and 'N' into the respective values INITIAL, DIGIT, HYPHEN, and NOGOOD of the enumerated type States. This follows the pattern first discussed in [2] using Pascal.

The next task is to represent the transition table by interactively inputting each row of the table as a character string, and then converting each character of the string into the corresponding member of States, using TheState. Each such character string will terminate with '$'. As an example, the first row of the transition table for the DFA described in [1] will be input as

DIN$

This will be converted to States.DIGIT, States.INITIAL, and States.NOGOOD, respectively.

As in [1], we use the two-dimensional array Transition to simulate the set of transitions of the underlying DFA. The first subscript again ranges over the members of States, and the second subscript ranges over the members of Alphabet. However, in this generalization, we do not employ the auxiliary method ordinal() to describe the subscripts. Instead, we apply the corresponding int values of the subscripts, in the order of appearance in the definitions of each of the enumerated types States and Alphabet.

More precisely, we define the dimensions of Transition using the declarations

final int AlphabetSize = 3;
final int StatesSize = 4;

and then declare Transition as

States[][] Transition = new States[StatesSize][AlphabetSize];

This allows us to alter the set of states and the alphabet of the DFA using very simple editorial changes. In addition, introducing the method TheState allows us to interactively input the respective rows of the transition table as character strings, each such row terminating with '$'. The following code sequence accomplishes this:

Scanner scan = new Scanner(System.in);
for(int index1 = 0; index1 < StatesSize; ++index1)
{
    System.out.println("Input the next row of the transition table");
    System.out.println("as a character string, terminating with ".");
    String strValue = scan.next();
int index2 = 0;
char chValue = strValue.charAt(index2);
Alphabet symbol;
while (index2 < AlphabetSize)
{
  Transition[index1][index2] = TheState(chValue);
++index2;
  chValue = strValue.charAt(index2);
} // terminates text of while-loop
} // terminates text of for-loop

The assignment

Transition[index1][index2] = TheState(chValue);

first converts the character value stored in chValue into its States equivalent, and then copies this converted value into its proper position in Transition. We must emphasize. However, that the user must input the respective rows of the transition table interactively in the proper order, as described by ordinal(). For example, if we use the transitions of the DFA of [1], the proper form of the input must be given by

DIN$
\>
DHNS
\>
DNNS
\>
NNS$

The next design issue involves the determination of the final states of the DFA. We wish to input these states interactively as members of a character string. These characters will then have to be converted into a form that is compatible with the members of States. Thus, we must input these characters as members of a String value, and then convert each such character to its corresponding States equivalent.

There is another important related consideration. Suppose we wish to place these values in a Set object (as the set of all final states of the DFA). We must then use the predefined Set data type, and then construct a Set object whose members are just those final states.

We will proceed by executing the code sequence

Scanner scan = new Scanner(System.in);
System.out.println("Please input the final states of the DFA");
System.out.println("as members of a character string, terminating with ").
// Construct an initially empty final state Set object.
Set<States> finalStates = new HashSet<States>();
String strValue = scan.next();
// Place holder for character in strValue
int s = 0;
char chValue = strValue.charAt(s);
while (chValue != '\$')

2 We could have just as easily defined finalStates as a TreeSet object.
States state = TheState(chValue);
// Add state to current set of final states
finalStates.add(state);
++s;
chValue = strValue.charAt(s);
} // terminates text of while-loop

This code sequence first constructs an initially empty set finalStates of final states of the DFA. The while-loop then fills finalStates with all of the final states of the DFA.

The remaining part of the simulator involves running the DFA with some specific input string of characters, testing whether that input string will be accepted or rejected. Here, as in [1], we first prompt the user for some input string and again invoke the static method Obtain to convert the individual char values of the input string into members of the enumerated set Alphabet. Recall that the coding of Obtain is as follows:

public static Alphabet Obtain(char ch)
{
    Alphabet sy = null;
    switch(ch)
    {
        case '/': sy = Alphabet.SLASH;
        break;
        case '-': sy = Alphabet.DASH;
        break;
        case 'o': sy = Alphabet.OTHER;
        case 'o': sy = Alphabet.OTHER;
    } // terminates text of switch
    return sy;
} // terminates text of Obtain method

There is one remaining consideration. Since the set of final states of the DFA have been formed interactively, the decision as to whether to accept or reject the input string depends upon whether the current state of the DFA after all of the characters of the input string have been scanned is in the Set object finalStates. Thus, the remaining code sequence of the main method may be designed as follows:

// Prompt user for input string
System.out.println("Please input any finite string of /, -, 0.");
System.out.println("Terminate string with $.");
String strValue1 = scan.next();
// Pointer to current character in the input string.
int ptr = 0;
char chValue1 = strValue1.charAt(ptr);
// Run machine
States currentState = States.INITIAL;
Alphabet symbol;
while(chValue1 != '$')
{
    symbol = Obtain(chValue1);
    currentState = Transition[currentState.ordinal()][symbol.ordinal()];
+++ptr;
    chValue1 = strValue1.charAt(ptr);
}  // terminates text of while-loop
if(finalStates.contains(currentState))
    System.out.println("Accepted");
else
    System.out.println("Rejected");

This terminates the text of the main method.

3. Summary and Further Generalizations:

This version of the simulator requires the user to provide interactive versions of three forms of input, in the order:

(1) the rows of the transition table of the DFA;
(2) the final states of the DFA;
(3) the input string to be tested.

The program as described simulates a specific DFA, but it is evident that the design we have described can be used to simulate other automata. If we were planning to simulate a number of automata using the above design, we would concede that each such DFA would involve the same enumerated types Alphabet and States. Further, with Alphabet and States fixed, each distinct simulation would require its own program.

We seek further flexibility. The ideal situation would involve a single program that would simulate any DFA, regardless of the alphabet and set of states. It would then be necessary for the user to provide the characters of the alphabet interactively, as well as the states of the machine as a subrange of int. In addition, the definition and construction of the Transition array would no longer involve entries from some specific enumerated type of states, but would again require int-valued entries. We will produce these generalizations in a subsequent paper.


The Seidenberg School of Computer Science and Information Systems
Pace University

Technical Report Series

EDITORIAL BOARD

Editor:
Allen Stix, Computer Science, Pace--Westchester

Associate Editors:
Constance A. Knapp, Interim Dean, The Seidenberg School of Computer Science and Information Systems -- Pace University
Susan M. Merritt, Computer Science, Pace--Westchester

Members:
Howard S. Blum, Computer Science, Pace--New York
Mary F. Courtney, Computer Science, Pace--Westchester
Nicholas J. De Lillo, Mathematics and Computer Science, Manhattan College
Daniel Farkas, Information Systems, Pace--Westchester
Fred Grossman, Information Systems; Doctor of Professional Studies, Pace--New York and White Plains
Fran Goertzel Gustavson, Information Systems, Pace--Westchester
Joseph F. Malerba, Computer Science, Pace--Westchester
John S. Mallozzi, Computer Information Sciences, Iona College
John C. Melluzzo, Information Systems, Pace--New York
Pauline Mosley, Technology Systems, Pace--New York
Narayan S. Murthy, Computer Science, Pace--New York
Catherine Ricardo, Computer Information Sciences, Iona College
Judith E. Sullivan, CSIS Alumna, MS in CS from Pace--Westchester
Sylvester Tuchy, Computer Science, Pace--Westchester

The Seidenberg School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. Technical Reports is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
The Seidenberg School of Computer Science and Information Systems
Goldstein Academic Center
Pace University
861 Bedford Road
Pleasantville, NY 10570-2799