Simulations of Finite-State Automata Using Java 6.0
Part Three: Generalizations

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.
1. Abstract.

In two earlier papers ([1],[2]) we simulated the behavior of deterministic finite-state automata (DFA) using the facilities of Java, Version 6.0. In [1], we studied a single DFA whose transitions and final states were explicitly given in the underlying code, and in [2], the rows of the transition table, as well as the final states were input interactively by the user. However, although the treatment given in [2] generalizes the design of [1], the alphabet and the states of the DFA remained fixed. In the present paper, we intend to simulate the action of any DFA using Java 6.0, with the understanding that the only constraints are those limited by the program with regard to the size of the alphabet and the number of states.

2. Planning the Design.

Our design plan is to simulate a number of automata using the methodology of the earlier papers. But if we follow the design of these papers, we would require a pair of enumeration types, one for specifying the alphabet, and the second for the set of states of the DFA. In brief, we would require a different program for each different choice of a DFA.

To circumvent this, we can attain the generality we seek using an integer-valued encoding of the alphabet and the set of states. Certain objects may be carried over from the earlier work, such as the simulation of the transitions as a two-dimensional array, and the set of the final states as a specific Set object, except that now these data structures must use int-valued entries. Specifically, in each of [1] and [2], we defined States and Alphabet as enumerated types. But now we must look at Alphabet as a type consisting of three values (SLASH, DASH, OTHER) without making any specific references to these identifiers, referring to these only as 0 (for SLASH), 1 (for DASH), and 2 (for OTHER). Similarly, States will be viewed merely as a type with four values (INITIAL, DIGIT, HYPHEN, NOGOOD), associated with 0 (for INITIAL), 1 (for DIGIT), 2 (for HYPHEN), and 3 (for NOGOOD). In this more general setting, we must translate these identifiers into the corresponding numerically coded entities.

Therefore, the user will be prompted first to input the DFA by first supplying the number of states: in the case of the DFA described in [1] and [2], the user responds to this prompt by inputting 4. Secondly, the user will be prompted to input the number of

---

1 We continue the convention of boldfacing Java keywords.
alphabet characters: if the user wishes to apply the alphabet as described in [1] and [2],
the user will input 3. It will then be necessary to input the actual alphabet characters as
members of a single character string; if the alphabet is that of [1] and [2], the user
responds by interactively inputting the character string \"/-o\" (for SLASH, DASH,
OTHER, respectively).

How will we view transitions in this more general setting? As before, Transition
will be viewed as a two-dimensional array, but now the array will have to be
"overdimensioned," since in this general setting, there is no fixed amount of states nor
alphabet characters. In addition, the component values of Transition will be of type
Integer, due to the numeric encoding of the states of the underlying DFA. We then
construct this array using the code

\[
\text{Integer[][] Transition = new Integer[10][10];}
\]

This implementation is then capable of handling transitions of DFA with a maximum
of 10 states, and an alphabet containing a maximum of 10 characters. Assume
StatesSize and AlphabetSize are int values representing, respectively, the number
of states and the number of alphabet characters of some DFA, and that these values have
been input interactively by the user. Then the following code sequence can be used to
input the values of the transition table:

\[
\text{System.out.println("Please input the rows of the transition table");}
\text{\&for(int row = 0; row < StatesSize; ++row)}
\text{\{ System.out.println("Please input the next row of the transition
\text{table:");
\text{\&for(int column = 0; column < AlphabetSize; ++column)}
\text{\{ int intValue = scan.nextInt();
\text{ Transition[row][column] = intValue;}
\text{\}} // terminates text of inner for-loop
\text{\} // terminates text of outer for-loop}
\]

It is also necessary to input the set of final states. This will be done in a manner
similar to that done in [2], except that now the members of the finalStates set will be
Integer values. We do so by first constructing finalStates as an initially empty
HashSet\footnote{We understand that scan has already been constructed using
\quad Scanner scan = new Scanner(System.in);} object, using

\[
\text{Set<Integer> finalStates = new HashSet<Integer>();}
\]

Then the executable code described below fills finalStates with just the final states of
the DFA:

\[
\text{System.out.println("Please input the final states of the DFA");}
\text{System.out.println("on separate lines, terminating with a -1:"};
\]

\footnote{finalStates could have just as easily been constructed as a TreeSet object.}
int intValue1 = scan.nextInt();
while (intValue1 != -1)
{
    finalStates.add(intValue1);
    intValue1 = scan.nextInt();
}  // terminates text of while-loop

Thus, if we input

1
2
-1

finalStates contains just the two Integer values 1 and 2.

We now turn to the task of inputting the characters of the alphabet. As stated earlier, these characters appear as the individual members of an input string, say str. These will then have to be converted to integer values, since they will reappear as values of the second subscript of the Transition array. Here we use a combination of two predefined methods of the String class: charAt and indexOf. The charAt method has one int-valued parameter, a non-negative integer value indicating a position in the string, and returning the character located at that position; indexOf returns the int value signaling the first occurrence in the string of the value of the single char-valued parameter. The individual int-valued code of each character of the alphabet is stored as a component of the one-dimensional Integer-valued array codes, which is constructed below immediately after the length of the input string is given by the user.

We thus obtain the necessary int-valued encoding of the alphabet using the sequence

// Prompt user for String input.
System.out.println("Please input the alphabet of the DFA as");
System.out.println("the individual characters of a string: ");

String str = scan.next();
Vector<Character> strVector = new Vector<Character>();

for(int index = 0; index < str.length(); ++index)
    strVector.add(str.charAt(index));

Applying this code to the input string "/c" establishes the encoding

'/c' corresponds to 0
'-c' corresponds to 1
'c' corresponds to 2

It is important to note that the string describing the alphabet can be any finite string of characters, not just the string "/c" used above. The only constraints are that no

4 Actually, the single parameter we use here is of the String type.
character may be repeated in the string, and (currently) the length of the string is ten characters or less. However, the size of the alphabet may be enlarged by a simple editorial change in the code defining the Transition array, followed by a reccompilation of the resulting code. In addition, note that the order of appearance of the characters in the alphabet appearing in the input string also affects the components of both Transition and finalStates.

3. Simulating the Action of the DFA.

The code sequences described in the last section prepare the DFA to begin its formal computation. We now describe the code sequence simulating the act of accepting or rejecting an input string whose symbols come from the underlying alphabet, except for the final character `$` signaling the end of the string. We therefore assume that `$` is not a character from the alphabet of the DFA. We also adopt the convention that the integer 0 represents the initial state, and that the states of the DFA are consecutive non-negative integers. Therefore, the DFA begins its action in the initial state, scanning the initial character of the input string, terminating with the character `$`.

The following code sequence simulates the action of the DFA:

```java
// Initialize action of the DFA
int state = 0; // initial state
// Prompt user for input string
System.out.println("Please input any string, terminating with $:");
// Input string interactively.
String inputString = scan.next();
// Initial position of scanner of input string.
int position = 0;
// Keep DFA active until $ is scanned
while(inputString.charAt(position) != '$')
{
    // holds integer code of current alphabet character
    char chValue = inputString.charAt(position);
    int indexValue;
    indexValue = strVector.indexOf(chValue);
    // Go to state determined by Transition
    state = Transition[state][indexValue];
    // Scan next character of input string
    ++position;
} // terminates while-loop
// At this point, inputString has been completely scanned
if(finalStates.contains(state)) // if last state is a final state
    System.out.println("Accepted");
else
    System.out.println("Rejected");
```

The text of the complete program is coded below.

```java
import java.io.*;
import java.util.*;

public class Simulator
```
public static void main(String [] args) {
    // Use Scanner class for interactive input
    Scanner scan = new Scanner(System.in);

    // Declare overdimensioned array.
    Integer[][] Transition = new Integer[10][10];

    // Declare size of set of states and alphabet of DFA
    int AlphabetSize, StatesSize;

    // Prompt user for number of states:
    System.out.println("Please input the number of states of the DFA:");
    StatesSize = scan.nextInt();

    // Prompt user for number of alphabet symbols:
    System.out.println("Please input the number of alphabet symbols of
    the DFA:");
    AlphabetSize = scan.nextInt();

    // Prompt user for the alphabet of the DFA as a character string.
    System.out.println("Please input the alphabet of the DFA as");
    System.out.println("the individual characters of a string:");
    String str = scan.next();

    // Associate each character of alphabet with its corresponding
    // integer value.
    // Place each character of the input string str as the component,
    // in order, of a Vector<Character> object called strVector:
    Vector<Character> strVector = new Vector<Character>();

    // strVector is initially empty.
    // Now input the characters of str, in order of appearance,
    // into strVector:
    for(int index = 0; index < str.length(); ++index)
        strVector.add(str.charAt(index));

    // Now input the values of the transition table:
    System.out.println("Please input the rows of the transition table");
    for(int row = 0; row < StatesSize; ++row)
    {
        System.out.println("Please input the next row of the transition
        table:");
        for(int column = 0; column < AlphabetSize; ++column)
        {
            int intValue = scan.nextInt();
            Transition[row][column] = intValue;
        } // terminates text of inner for-loop
    } // terminates text of outer for-loop

    // Construct the set of final states of the DFA.
    Set<Integer> finalStates = new HashSet<Integer>();
    System.out.println("Please input the final states of the DFA");
    System.out.println("on separate lines, terminating with a -1:");
    int intValue1 = scan.nextInt();
while(intValue1 != -1)
{
    finalStates.add(intValue1);
    intValue1 = scan.nextInt();
} // terminates text of while-loop

// The code above initializes the components of the DFA. Now we
// want to run the machine with these specifications.

// Initialize action of the DFA.
int state = 0; // machine begins execution in initial state

// Prompt user for input string:
System.out.println("Please input any string, terminating with $:");
// Input string interactively.
String inputString = scan.next();

// Initial position of scanner of input string.
int position = 0;

// Keep DFA active until $ is scanned.
while(inputString.charAt(position) != '$')
{
    // inputValue holds integer code of current alphabet character
    char chValue = inputString.charAt(position);
    int inputValue;
    inputValue = strVector.indexOf(chValue);
    // Go to state determined by Transition
    state = Transition[state][inputValue];
    // Scan next character of input string
    ++position;
} // terminates while-loop

// At this point, inputString has been completely scanned.
if(finalStates.contains(state)) // if last state is a final state
    System.out.println("Accepted");
else
    System.out.println("Rejected");

} // terminates text of main method
} // terminates text of class Simulator

Let us trace an execution of this program. After scan has been constructed, and
Transition, AlphabetSize, and StatesSize have been allocated storage, the user is
prompted with

Please input the number of states of the DFA:

Suppose we agree to implement the DFA described in[1], and the user responds to the
prompt with 4. The user is then prompted again with the message
Please input the number of alphabet symbols of the DFA:

The user responds to this prompt with 3. The user is prompted again with

Please input the alphabet of the DFA as
The individual characters of a string:

The user responds to this prompt with "/0". Actually, any sequence of length three using all of these symbols can be used. The only difference is the coding of these symbols with the digits 0, 1, 2 will affect the rows of Transition in that the rows will be in a different order, but this will not affect the final outcome. In response to the actual input string used above, the codes array is created, using the correspondences

'/' corresponds to 0
'\-\' corresponds to 1
'\0' corresponds to 2

At this point, the user is prompted with

Please input the rows of the transition table

After the next prompt

Please input the next row of the transition table

the user inputs

1
0
3

completing the input of the first row of the transition table.

The last prompt

Please input the next row of the transition table

Is repeated, and the user responds by inputting

1
2
3

completing the input of the second row of the transition table.

The same prompt is repeated, to which the user responds

1
completing the input of the third row of the transition table.

The same prompt appears one last time, to which the user responds with

completing the input of the fourth and last row of the transition table. Execution of the program continues with the prompt

Please input the final states of the DFA on separate lines, terminating with a -1:

Suppose the user responds to this prompt with the sequence

1
2
-1

This constructs the set finalStates with the entries 1, 2. After all of this is completed, the simulated DFA is ready to test any input string. Suppose the user provides the input

///////////$

in response to the prompt

Please input any string, terminating with $:

for (3, 4, 1) as explained in [1]. Then position is initialized at 0, indicating that the DFA is scanning the leftmost / of the input string. The value of the variable index is initialized at 0, since this is the integer code corresponding to '/': that integer value is found as one of the components of the codes array. The value of state becomes 1, position increments to 1, and the scanning continues. Processing in the while-loop eventually terminates, with state assuming a last value of 1. Since this last value appears as a member of finalStates, the input string is accepted.


Several possible options could be used for the design of this simulator. For example, we defined Transition as an “over-dimensioned” two-dimensional array, with 10 rows and 10 columns. In the specific case of Simulator, we used only four rows and three columns; the remaining positions in the array representing wasted allocated storage. It may be possible to redefine Transition as a dynamic data structure, using one or more
classes defined in the Java Collections Hierarchy, perhaps using the predefined classes 
LinkedList, ArrayList, or Vector objects, as a means of avoiding this allocation of 
wasted storage. Such a design would involve, among other things, how to reference a 
state parameter (viewed as an integer value) or an alphabet character (again viewed as an 
integer value).

The treatment of DFAs used in [2] and in the present paper makes heavy use of 
interactive input. However, in the original treatment of the simulation of DFAs using 
Pascal, as discussed in [5], the specifications of the underlying DFA appeared in a text 
file. It might be possible to explore the possibility of doing the same in Java – use the file 
processing capabilities of Java for text files instead of using interactive input. (See [4] 
and [7]).

We may also note that the text of the main method of the Simulator class contains 
all of the necessary code for installing and initializing the simulated DFA, as well as the 
work of processing any input string of characters from the underlying alphabet to decide 
whether that input string is accepted or rejected by that DFA. It may be meaningful to 
modularize this code so that the main method may be used simply as a toggle mechanism 
to invoke several static methods for initializing the DFA and then testing whether that 
DFA accepts or rejects some specific input string.

The next step beyond what this paper has accomplished might be to design the 
counterpart of the Pascal program convert, as described in [5], beginning with the 
specifications of a nondeterministic finite-state automaton, converting this automaton to 
an equivalent DFA. With this DFA constructed, we would proceed as in [6], where the 
C++ language was used, to convert this form of a DFA to an equivalent minimal-state 
DFA, now using Java 6.0 as the implementation language.

One final observation: the results of [1], [2], and the present paper could also be 
redone using C++ as the implementation language. This is particularly useful, since C++ 
also supports enumeration types and object-oriented design.
5. Bibliography.


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. Molluzzo, 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 Tuohy, 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