Simulations of Finite-State Automata Using Java 6.0
Part One: Preliminaries

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 One: Preliminaries

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

1. Abstract.

In this paper and in the sequel, we continue the work originally done in [1] and
[2], except that now we use the Java programming language. This is due primarily to the
observation that, in its most recent revisions, especially in the facilities provided in its
java.util library, Java contains many of the predefined utilities required to simulate
the activities of finite-state automata using the advantages of object-oriented design. The
original work of simulation done in [1] was in Pascal, and the construction and
implementation of the minimal finite-state automaton in [2] uses C++.

2. Preliminaries.

We design a number of fundamental abstract machines which, when combined in a
specific order, produce the effect of the kind of calculations that mimic the activity of a
contemporary digital computer. The simplest of these machines is the deterministic
finite-state automaton. Such a machine M is capable of scanning any finite string of
input symbols, one character at a time. This scanning operation is governed by a finite
control, which is in exactly one of a finite number of states while scanning any symbol,
and may, depending on the current state of the machine and the specific symbol being
scanned, make a transition to another state (possibly remaining in the same state). At
least one of the states of M is designated as a final state; a string of input symbols is
accepted by M if, and only if, the state of M after scanning the entire string is one of the
designated final states.

One state of M is distinguished and named the initial state, and is viewed as the
unique state that initiates any activity of M. After beginning in the initial state, the
activity of M may be summarized by: scan a symbol, perform the transition (if such
exists) to the next state, read the next symbol from the input string, and so on, until the
entire input string is exhausted. If M is in a designated final state at the point when all of
the symbols of the input string have been scanned, then the string is accepted by M;
under all other circumstances, the input string is rejected. Given any automaton M, the
collection of all strings accepted by M is called the language generated by M, denoted by
L(M).

We may then characterize a deterministic finite-state automaton (DFA) M by
- A finite, non-empty set $Q$, whose members are the states of $M$;
- A unique element $I$ of $Q$, called the designated initial state;
- A non-empty subset $F$ of $Q$, of final states of $M$;
- A finite, non-empty set $A$, where $A \cap Q$ is empty, whose members are the input symbols or alphabet of $M$;
- A mapping $T$, called the transition map, which associates a unique member $q \in Q$ with each suitably chosen pair $(q,a)$, where $q \in Q$ and $a \in A$.

Suppose $s = a_1a_2 \ldots a_n$ is a string of symbols from $A$. We then write $T(q,s) = q'$ if $M$ winds up in state $q'$ after scanning each symbol of $s$ in succession and having begun the scanning process in state $q$. For purely technical reasons, if $e$ denotes the empty string of length zero and containing no symbols, we may write $T(q,e) = q$ to mean that there is no change in state. Finally, we may characterize $L(M)$ by

$$L(M) = \{s : T(i,s) = q', \text{ where } q' \in F\};$$

that is, if initiating $M$ with input string $s$ leaves $M$ in a final state after the entire input string is exhausted.

As an illustration of these ideas, suppose we consider the DFA $M$ given by

$$Q = \{i,d,h,n\}$$
$$A = \{l,r,o\}$$
$$F = \{d,h\}$$

and with a transition map $T$ defined as follows:

$$T(i,l) = i;$$
$$T(d,r) = h;$$
$$T(h,l) = n;$$
$$T(q,l) = d \text{ for any } q \in Q, \text{ provided } q \neq n;$$
$$T(q,a) = n \text{ for any } q \in Q, \text{ provided } a \neq l,r;$$
$$T(n,a) = n \text{ for any } a \in A.$$  

We may now summarize the description of $T$ in concise form by a transition table. For the example given above, the transition table for $T$ is given as in (Figure 1):
\[
\begin{array}{c|cccc}
/ & \text{o} \\
\hline
i & d & i & n \\
\hline
d & d & h & n \\
\hline
h & d & n & n \\
\hline
n & n & n & n \\
\end{array}
\]

(Figure 1)

3. Simulation of the DFA Using Java.

We wish to simulate the actions of the DFA of the previous section by designing a Java program. The Java programming language, in its most recent releases, possesses all of the necessary features required to facilitate this simulation.

To begin with, we note that the states of \( M \) are given as members of the finite set \( Q = \{i,d,h,n\} \), where \( i \) stands for INITIAL, \( d \) for DIGIT, \( h \) for HYPHEN, and \( n \) for NOGOOD. In Java 6.0, it is possible for the programmer to define an enumerated type whenever appropriate.\(^1\) Accordingly, we may then represent the states of \( M \) as an enumerated type, using the syntax

\[
\text{enum States \{}\text{INITIAL,DIGIT,HYPHEN,NOGOOD}\};
\]

It is also possible to declare variables of the \text{States} data type, as well as variables of any other designated enumerated type. A variable of the \text{States} type may store any one of the four values listed above, provided that these values include the type name as a qualifier, as in

\[
\begin{align*}
\text{States.INITIAL} \\
\text{States.DIGIT} \\
\text{States.HYPHEN} \\
\text{States.NOGOOD}
\end{align*}
\]

As well, we may also represent the alphabet of \( M \) as the enumerated type

\[
\text{enum Alphabet \{}\text{SLASH,DASH,OTHER}\};
\]

\(^1\) Actually, enumerated types were first supported in Java 5.0 (2005).

\(^2\) We express any Java keywords in boldface.
Since we plan the input stream to be presented to this simulated machine as a stream of char-valued data, we will have to provide a subprogram, which we will call obtain, to be used to read single characters, and then convert each such character internally to one of the values of Alphabet. For example, if we assume the variable declaration

Alphabet symbol;

we may read a char value from the input stream (such as '/') and then use obtain to convert this value to Alphabet.SLASH, which will then be stored as the current value of symbol. We may perform similar conversions in case the character read is either '-.' or any other character value (denoted by 'o').

How do we implement transitions? This is done using a two-dimensional array called Transition, whose first subscript ranges over the members of States, and whose second subscript ranges over the members of Alphabet, with components ranging over the members of States. Consequently, we declare

States[][] Transition = new States[4][3];

There is one obstacle in using Java as the implementation language that did not occur in the original Pascal implementation in [1]. While both implementations use subscripts originating from values in certain designated enumeration types, Java demands that array subscripts be designated explicitly using nonnegative integers. To reconcile this with enumerated types, Java 5.0 first introduced the ordinal() attribute for each enumerated type. Using the examples from the enumerated types States and Alphabet defined above, we observe the behavior of ordinal() defined on these types by

    INITIAL.ordinal() = 0
    DIGIT.ordinal() = 1
    HYPHEN.ordinal() = 2
    NOGOOD.ordinal() = 3
    SLASH.ordinal() = 0
    DASH.ordinal() = 1
    OTHER.ordinal() = 2

As a result, Transition simulates the behavior of the transition table defined in Section 2 and represented in (Figure 1), where

    Transition[States.INITIAL.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
    Transition[States.INITIAL.ordinal()][Alphabet.DASH.ordinal()] = States.INITIAL;
    Transition[States.INITIAL.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
    Transition[States.DIGIT.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
    Transition[States.DIGIT.ordinal()][Alphabet.DASH.ordinal()] = States.HYPHEN;
    Transition[States.DIGIT.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
    Transition[States.HYPHEN.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
Another issue to resolve is how to present the input string. We will do so interactively, with ‘$’ as the terminating symbol. The reason for doing this is twofold:

- ‘$’ will never be interpreted as a member of Alphabet;
- the inclusion of ‘$’ at the end of the input string will guarantee that every symbol of the string will be scanned, and provides a well-defined condition for terminating the subsequent while-loop.

Using Java 5.0 and 6.0, interactive input has been greatly simplified with the inclusion of the predefined Scanner class (now part of the revised java.util library). A String value can then be input interactively in this environment using the code sequence

```java
Scanner scan = new Scanner(System.in);
String strValue = scan.next();
```

In order to read individual character from the input string, we will require a combination of a character counter in the form of an int-valued variable s, initialized to zero, and incremented each time it becomes necessary to read the next character from the input string. Initially, this requires the code sequence

```java
int s = 0;
char chValue = strValue.charAt(s);
```

Finally, in order to properly use the results provided by the Transition array, we require the coding of the subprogram Obtain described above. The coding is given by a method that returns an Alphabet value, and appears as

```java
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;
               break;
    default: sy = null;
    }
    return sy;
} // terminates text of switch
```
The current state of the machine begins with the initial state (States.INITIAL) and consequently is governed by the values returned by examining the Transition array. Thus, initially, we must have the declaration

```java
States currentState = States.INITIAL;
```

The ultimate decision as to whether to accept the input string depends upon the fact that the set of final states is given by States.Digit and States.HYPHEN. Accordingly, the input string will be accepted by the DFA if, and only if the final value for `currentState` is either States.Digit or States.HYPHEN.

The complete text of the simulation program is then given by

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

class Acceptnum
{
    enum Alphabet {SLASH, DASH, OTHER};
    enum States {INITIAL, DIGIT, HYPHEN, NOGOOD};

    public static void main(String[] args)
    {
        // Initialize machine. Use Transition array
        States[][] Transition = new States[4][3];
        Transition[States.INITIAL.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
        Transition[States.INITIAL.ordinal()][Alphabet.DASH.ordinal()] = States.INITIAL;
        Transition[States.INITIAL.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
        Transition[States.DIGIT.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
        Transition[States.DIGIT.ordinal()][Alphabet.DASH.ordinal()] = States.HYPHEN;
        Transition[States.DIGIT.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
        Transition[States.HYPHEN.ordinal()][Alphabet.SLASH.ordinal()] = States.DIGIT;
        Transition[States.HYPHEN.ordinal()][Alphabet.DASH.ordinal()] = States.NOGOOD;
        Transition[States.HYPHEN.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
        Transition[States.NOGOOD.ordinal()][Alphabet.SLASH.ordinal()] = States.NOGOOD;
        Transition[States.NOGOOD.ordinal()][Alphabet.DASH.ordinal()] = States.NOGOOD;
        Transition[States.NOGOOD.ordinal()][Alphabet.OTHER.ordinal()] = States.NOGOOD;
    }
}
```
// Prompt user for input string:
System.out.println("Please input any finite string of /,-,0:");
System.out.println("Terminate string with $:");
// Establish interactive input stream.
Scanner scan = new Scanner(System.in);
String strValue = scan.next();
// pointer to current character in the input string
int s = 0;
char chValue = strValue.charAt(s);

// Run machine.
States currNet = States.INITIAL;
Alphabet symbol;
while(chValue != '$')
{
    symbol = Obtain(chValue);
    currNet = Transition[currNet.ordinal()][symbol.ordinal()];
    ++s;
    chValue = strValue.charAt(s);
} // terminates text of while-loop
if(currNet == States.DIGIT || currNet == States.HYPHEN)
    System.out.println("Accepted");
else
    System.out.println("Rejected");
} // terminates text of main method

// Text of Obtain subprogram follows.
public static Alphabet Obtain(char ch)
{
    Alphabet sy = null;
    switch(ch)
    {
        case '/': sy = Alphabet.SLASH;
            break;
        case '-': sy = Alphabet.DASH;
            break;
        case '0': sy = Alphabet.OTHER;
            break;
    } // terminates text of switch
    return sy;
} // terminates text of Obtain method

4. Consequences and Areas For Further Study.

The DFA M described in Section 2 and simulated in Section 3 have a number of important and useful applications. To begin with, we note that it is possible to represent any non-negative integer (natural number) n as a string of n + 1 consecutive slash symbols, as in //.../ (of length n + 1). Further, if we view a single dash (hyphen) as a separator between sequences of consecutive slash symbols, we may also represent any m-tuple of non-negative integers, where m ≥ 1, as strings of slashes and dashes of the form

//.../-//.../- ... -//.../
and where the first string of slashes has length $n_1 + 1$, the second has length $n_2 + 1$, \ldots, and the last has length $n_m + 1$. For example, the quintuple $(3,4,1,6,0)$ is represented by the string \(/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\backslash/\). Noting this, it turns out that $M$ generates all strings of this type; that is $L(M)$ contains all such strings.

Of particular importance in this regard is the observation that all such strings represent viable input values for primitive recursive function of $m$ variables, for any specific choice of $m$. The actual computation of any such function can then be performed on a Turing machine, which may also be simulated using Java. The design and implementation details of such simulations are areas of current research.

More flexible and general representations of simulations of deterministic finite-state automata appear in the sequel, where, for example, some of the specific structures of the containers defined in the Java Collections Framework will be applied.

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. Mallerba, 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
Sylvestre 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