Simulations of Finite-State Automata Using the Standard Template Library
~ Variations on a Theme Using C++ ~

Nicholas J. DeLillo
As explained in the Abstract on the first page, the present paper is a recapitulation, in C++, of the full set of concepts developed the three earlier presentations, listed below, using Java:


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 THE STANDARD TEMPLATE LIBRARY
Variations on a Theme Using C++

Nicholas J. DeLillo

Professor of Mathematics and Computer Science
Manhattan College

1. Abstract.

In a sequence of earlier papers (see [1], [2], [3]) we simulated the behavior of deterministic finite-state automata (DFA) using Java 6.0 as the underlying implementation language. In these earlier papers, the simulations involved a number of the key principles of object-oriented design (OOD), most notably the use of sets of objects, and one- and two-dimensional arrays of objects. In fact, these earlier papers progressed from a single specific simulation to a very general setting in which any DFA can be simulated. In the current paper, we summarize these results, applying the C++ programming language, specifically some of the more useful predefined facilities of the Standard Template Library (STL). This is done primarily to illustrate that the main factor in the simulation is OOD, and not the specific choice of one particular object-oriented programming language.

2. Preliminary Ideas.

In this section, we review a number of theoretical ideas concerning abstract finite-state machines which, when applied in a specific sequential order, mimic the behavior of a contemporary digital computer. Many of these ideas have already been described in [1] and [3].

The simplest of these machines is called a deterministic finite-state automaton (DFA), having the capability of scanning any finite string of input symbols coming from some predefined finite and non-empty alphabet, 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 some input symbol. Depending on the current state of the machine and the specific symbol being scanned, the machine may perform a transition to another state, or possibly remain in the present state. At least one of the states of the machine is designated as a final state, and exactly one of the states is defined as the initial state: the state of the machine as it begins its operation.

After beginning in the initial state, the activity of the machine may be characterized by: scan an input symbol, perform the transition (if such exists) to the next state, scan the
next input symbol, and so on, until all of the symbols of the input string have been scanned. If the machine is in one of the designated final states at the point when all of the symbols of the input string has been scanned, they we say the input string is accepted by the machine. Under any other circumstance, the input string is rejected. Given any such machine M, the collection of all finite strings accepted by M is called the language generated by M, and is denoted by L(M).

Using these ideas, we may characterize a deterministic finite-state automaton (DFA) M by the aggregate consisting of

1. a finite, non-empty set Q, whose members are the states of M;
2. a unique element i of Q, called the designated initial state;
3. a non-empty subset F of Q, called the final states of M;
4. a finite, non-empty set A, where A\cap Q is empty, whose members are the input symbols or alphabet of M;
5. a mapping T, called the transition map, associating 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...a_n is a string of symbols from A. We then write T(i, s) = q' if M winds up in state q' after scanning each symbol of s in succession, having begun the scanning process in the initial state i. Finally, we may characterize L(M) by

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

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

To illustrate these ideas, suppose we consider the DFA M with the specifications

\[ Q = \{ i, d, h, n \} \]
\[ A = \{ /, -, o \} \]
\[ F = \{ d, h \} \]

and with a transition map T defined as follows:

\[ T(i, -) = i; \]
\[ T(d, -) = h; \]
\[ T(h, -) = n; \]
\[ T(q, /) = d \text{ for any } q \in Q, \text{ provided } q \text{ is not } n; \]
\[ T(q, a) = n \text{ for any } q \in Q, \text{ provided } a \text{ is not } / \text{ or } -; \]
\[ T(n, a) = n \text{ for any } a \in A. \]

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 described in (Figure 1):
<table>
<thead>
<tr>
<th></th>
<th>/</th>
<th>-</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>d</td>
<td>i</td>
<td>n</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>h</td>
<td>n</td>
</tr>
<tr>
<td>h</td>
<td>d</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

(Figure 1)

3. **Simulation of the Last DFA Using the Facilities of C++.**

In a manner similar to that of Java 6.0 described in [1], C++ supports the definition of new user-defined simple types by listing (enumerating) the finite sequence of values that comprise the domain of any such type. These values assume the form of identifiers and are separated by commas. These are enclosed within braces and preceded by the keyword `enum`. Any data type defined in this manner and named by an identifier is called an *enumeration type*.

A very useful example of this is the description of the states of the machine M defined in the last section. This may be implemented in C++ by the enumeration type

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

The C++ language also permits the declaration of variables of any programmer-defined enumeration type. Consequently, the definition and initialization

```cpp
States current_state = INITIAL;
```

is possible in C++.

Moreover, we may also represent the alphabet of M as the enumeration type

```cpp
enum Alphabet {INITIAL, DIGIT, HYPHEN, NOGOOD};
```

Our design of the simulation of M involves the use of a finite sequence of `char` values as the value of the string on the input tape. Therefore, it will be necessary to convert each such `char` value into its equivalent `Alphabet` value, in order to provide the proper transition to a new state provided by M. We will require both a subprogram to convert the current `char` value read from the input tape to its equivalent `Alphabet` value, and also a simulation of the associated transition T of M using `Alphabet` values.

---

1 Any C++ keyword will be expressed in boldface.
The simulation of transitions will be accomplished by a two-dimensional array called Transition, whose first subscript ranges over the values appearing in States, and whose second subscript ranges over the values of Alphabet. These values appear as they are arranged in their respective enumeration type, although C++ allows any permutation of these values as well. In addition, the values of the components of Transition will be members of States. This simulation, along with initializations of the components of Transition, is permissible in C++ using

```cpp
States Transition[4][3] =
{{DIGIT, INITIAL, NOGOOD},
 {DIGIT, HYPHEN, NOGOOD},
 {DIGIT, NOGOOD, NOGOOD},
 {NOGOOD, NOGOOD, NOGOOD}};
```

This definition provides the complete simulation of the transition table described in (Figure 1).

The subprogram providing the necessary conversion of the char value read from the input string to its corresponding Alphabet value is accomplished by Obtain, coded as follows:

```cpp
Alphabet Obtain(char ch)
{
    Alphabet sy;
    switch(ch)
    {
        case '/': sy = SLASH;
                   break;
        case '-': sy = DASH;
                   break;
        case 'o': sy = OTHER;
    } // terminates text of switch
    return sy;
} // terminates text of Obtain
```

Another issue to resolve is how to represent the input string. This will be done interactively, with '§' as the char symbol used to indicate the termination of the string. The reason for 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.

In order to read individual characters 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. This is accomplished using the code sequence
char ch_value;
cin >> ch_value;

The initial state of M is represented by
States current_state = INITIAL;

and the value of current_state will change according to the next character read from
the input string and the current state of M, using Transition.

The ultimate decision as to whether to accept or reject the input string depends upon
the fact that the set F of final states is represented by \{DIGIT, HYPHEN\}. Accordingly,
the input string will be accepted by M is, and only if, the final value of current_state
is either DIGIT or HYPHEN.

The complete text of this simulation program is then given by

// Definition of enumerated types.
enum Alphabet {SLASH, DASH, OTHER};
enum States {INITIAL, DIGIT, HYPHEN, NOGOOD};

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

// Main function
int main()
{
    // Initialize machine. Use Transition array.
    States Transition[4][3] =
    {
        {DIGIT, INITIAL, NOGOOD}, {DIGIT, HYPHEN, NOGOOD}, {DIGIT, NOGOOD, NOGOOD},
        {NOGOOD, NOGOOD, NOGOOD}:
    } // Prompt user for input string:
cout << "Please input any finite string of /,-,o:" << endl;
cout << "Terminate string with $:" << endl;
char ch_value;
cin >> ch_value;
// Run machine
States current_state = INITIAL;
Alphabet symbol;
while(ch_value != '$')
{
symbol = Obtain(ch_value);
current_state = Transition[current_state][symbol];
cin >> ch_value;
} // terminates text of while-loop
if((current_state == DIGIT || current_state == HYPHEN)
cout << "Accepted" << endl;
else
cout << "Rejected" << endl;
return 0;
} // terminates text of main function

The DFA that we have defined in Section 2 and simulated in the current section has several important and useful applications. First, we note that it is possible to represent any natural number (non-negative integer) 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 k-tuple (n1, n2, ..., nk) of non-negative integers, where k ≥ 1, as a string of slashes and dashes of the form

//.../-//.../ - ... -//.../

where the first string of slashes has length n1 + 1, the second has length n2 + 1, ..., and the last has length nk + 1. For example, the quintuple (3,4,1,6,0) is represented by the string ///-///////-/-///-///-. Noting this, we may show that M generates all strings of this type.

Of particular importance in this regard is the observation that all such strings in any alphabet containing slashes and dashes represent input values applicable to any primitive recursive function of any finite number of variables. Such acceptable forms may then be applied as input for any Turing machine, also capable of simulation using C++.

4. Extensions of the DFA Simulation of Section 3.

Although the program described in the last section successfully simulates the DFA M described in Section 2, it also contains a number of obvious limitations. First of all, the simulation depends upon the choice of the alphabet, number of states, number of final states, and the specific form of the transition map. In the sequel, we wish to present a simulation of any DFA, in which the alphabet, states, final states, and transitions can be input interactively by the user at run-time. We will achieve this result gradually, by first coding a simulation of a DFA in which the underlying alphabet, states, and final states remain fixed and the transition table is input dynamically by the user, and then the most general situation where alphabet, states, and final states, as well as transitions, are all input dynamically.

We begin this first level of generalization by again assuming the definition of the enumeration types States and Alphabet. However, the generalization will require that the rows of the transition table be input row by row, and will also require that the set of
final states be input dynamically. Each of these input values will have to be provided interactively, and will have to correspond to values already defined in States. The input values will be from char, with 'I' (for INITIAL), 'D' (for DIGIT), 'H' (for HYPHEN), and 'N' (for NOGOOD). Since these are char values that must correspond to values appearing in States, we will require a States-valued function, which we will call TheState, that converts the char value that is input into its corresponding States value. This will be accomplished using

```cpp
States TheState(char ch)
{
    States st;
    switch(ch)
    {
        case 'I': st = INITIAL;
            break;
        case 'D': st = DIGIT;
            break;
        case 'H': st = HYPHEN;
            break;
        case 'N': st = NOGOOD;
    } // terminates text of switch
    return st;
} // terminates text of TheState
```

The next objective is to construct the transition table. In order to do so, we must first fix the number of rows and columns of the table. The number of rows will be the number of distinct states of the DFA, that is, the size of the enumeration type States. Correspondingly, the number of columns will be the number of distinct characters of Alphabet; thus, if we wish to simulate the DFA of Section 2, we require the const definitions

```cpp
const int Alphabet_Size = 3;
const int States_Size = 4;
```

Using these definitions, we may then declare Transition as

```cpp
States Transition[States_Size][Alphabet_Size];
```

Now that the transition table has been represented by Transition, we may interactively input each row of the table as a sequence of char values, converting each such value into the corresponding member of States, using TheState. The next code sequence accomplishes this

```cpp
char ch_value;
for(int index1 = 0; index1 < States_Size; ++index1)
{
    cout << "Input each row of the transition table" << endl;
    cout << "as a sequence of " << Alphabet_Size << " characters" << endl;
    int index2 = 0;
    while(index2 < Alphabet_Size)
    {
```
cin >> ch_value;
Transition[index1][index2] = TheState(ch_value);
++index2;
} // terminates text of inner while-loop
cout << "Completed input of the current row" << endl;
} // terminates text of outer for-loop

The next step is to construct the set of final states of the DFA. We will do so interactively, using a set<
c> object from the Standard Template Library (STL). However, in order to do so, we must deal with several concessions. First of all, such an object required the use of the set interface, defined as one of the components of the STL. In C++, in order to apply this interface, we must include two preprocessor directives

#include <set>

and

using namespace std;

in our program file.

Secondly, due to the current regulations imposed by the most recent version of the C++ standard, any such set<
c> object can contain at best only a finite number of values of a single predefined primitive data type, such as char. Thus, in order to construct the set final_states of final states of the underlying DFA, we will first have to invoke the constructor

set<
c> final_states;

This constructs final_states as an initially empty set of char values.

Next, in order to produce the set of final states that we require for our DFA, we will have to associate each member of States with a uniquely determined char value. For example, if we apply this principle to the DFA described in Section 2, we use the association

'I' for INITIAL
'D' for DIGIT
'H' for HYPHEN
'N' for NOGOOD

Further, the set template contains the predefined operation insert, which, when provided with a char value (in our case), inserts that value into the current set<
c> object constructed. Therefore, after constructing the set<
c> object final_states, we map apply the code sequence
// Prompt user for the final states of the DFA.
cout << "Please input the final states of the DFA" << endl;
cout << "as individual characters, terminating with a $." << endl;

char char_value;
cin >> char_value;
while(char_value != '$')
{
    final_states.insert(char_value);
cin >> char_value;
} // terminates text of while-loop

As an example, if we again refer to the DFA of Section 2, and after having constructed an initially empty set<char> object final_states, we respond to the prompt by setting char_value interactively and successively equal to 'D' and then 'H', final_states becomes the two-member set of char values given by {'D', 'H'}.

Now that the set of final states of the DFA has been constructed, the decision as to whether to accept or reject the input string depends on whether the current state of the DFA after all of the characters of the input string have been scanned is a member of the set<char> object final_states. This is possible in C++, but it requires making several key observations involving the C++ language in general, and the set interface of STL in particular.

Unlike Java, the set template of STL does not possess a predefined non-static method contain with a boolean return value true just when the value tested is in fact a member of the set object, and false if otherwise. Instead, the set template of STL contains a find function with a single parameter value such that, for any set<char> object set_object, set_object.find(value) returns an iterator referring to value just when value is a member of set_object. If this is the case, then applying a pointer operator * to that iterator returns the contents of value as a member of set_object.

It is very important to note another basic fact about objects constructed using the set interface. The C++ language does not permit set to be applied to any programmer-defined enumerated type. That is, the type parameter used for a set interface cannot be any enumerated type defined by the programmer. Specifically, C++ does not define the set<States> template for the enumerated type States as defined above, but does permit the use, for example, of the set<char> template. Consequently, while it is possible to define the entries of the Transition array as members of States, the final determination as to whether the value of current_state after all of the characters of the input string have been scanned, requires the conversion of this value of current_state to its equivalent char value to be done in order to arrive at the decision as to whether to accept or reject that string. For this reason, we require the definition of a function display, coded as

char display(States st)
{
    switch(st)
    {

case INITIAL: return 'I';
    break;
  case DIGIT: return 'D';
    break;
  case HYPHEN: return 'H';
    break;
  case NOGOOD: return 'N';
} // terminates text of switch
} // terminates text of display

The activity of the DFA with respect to accepting or rejecting any input string of finite length can then be captured in C++ by the code sequence

// Prompt user for input string
cout << "Please input any finite string of /, -, o." << endl;
cout << "Terminate string with $:" << endl;

char ch_value1;
cin >> ch_value1;
// Run machine
States current_state = INITIAL;
Alphabet symbol;
while(ch_value1 != '$')
{
    symbol = Obtain(ch_value1);
    current_state = Transition[current_state][symbol];
    cin >> ch_value1;
} // terminates text of while-loop

char char_value2;
char_value2 = display(current_state);
// Test whether this last state is a member of the set
// of final states.
set<char>::const_iterator qptr;
qptr = final_states.find(char_value2);
if(*qptr == char_value2)
    cout << "Accepted" << endl;
else
    cout << "Rejected" << endl;

This completes the text of the main function.

The complete text of this version of the simulation program follows.

#include <iostream.h>
#include <set>
using namespace std;

// Definition of enumerated types States and Alphabet.
enum States {INITIAL, DIGIT, HYPHEN, NOGOOD};
enum Alphabet {SLASH, DASH, OTHER};

// Definition of the respective sizes of these
// enumerated types.
const int States_Size = 4;
const int Alphabet_Size = 3;

// Definition of the Transition array.
States Transition[States_Size][Alphabet_Size];

// Here is the code for the TheState function:
States TheState(char ch)
{
States st;
switch(ch)
{
case 'I': st = INITIAL;
  break;
case 'D': st = DIGIT;
  break;
case 'H': st = HYPHEN;
  break;
case 'N': st = NOGOOD;
}  // terminates text of switch
return st;
}  // terminates text of TheState

// Display function. Used to display the components of the
// Transition array and final decision about the input string.
// 'I' stands for INITIAL, 'D' for DIGIT, 'H' for HYPHEN,
// and 'N' for NOGOOD.
char display(States st)
{
switch(st)
{
case INITIAL: return 'I';
  break;
case DIGIT: return 'D';
  break;
case HYPHEN: return 'H';
  break;
case NOGOOD: return 'N';
}  // terminates text of switch
}  // terminates text of display

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

int main()
{  
// Prompt user for the rows of the transition table.  
// These values are converted from input char values  
// to corresponding States values using TheState.  
char ch_value;  
for(int index1 = 0; index1 < States_Size; ++index1)  
{  
cout << "Input each row of the transition table" << endl;  
cout << "as a sequence of " << Alphabet_Size << " characters" << endl;  
int index2 = 0;  
while(index2 < Alphabet_Size)  
{  
cin >> ch_value;  
Transition[index1][index2] = TheState(ch_value);  
++index2;  
} // terminates text of inner while-loop  
cout << "Completed input of current row" << endl;  
} // terminates text of outer for-loop  
  
// Construct an initially empty set of final states  
set<char> final_states;  
// Prompt user for the final states of the DFA  
cout << "Please input the final states of the DFA" << endl;  
cout << "as individual characters, terminating with a $." << endl;  
char char_value;  
cin >> char_value;  
while(char_value != '$')  
{  
final_states.insert(char_value);  
cin >> char_value;  
} // terminates text of while-loop  
  
// Prompt user for input string  
cout << "Please input any finite string of /, -, o." << endl;  
cout << "Terminate string with $:" << endl;  
char ch_value1;  
cin >> ch_value1;  
// Run machine  
States current_state = INITIAL;  
Alphabet symbol;  
while(ch_value1 != '$')  
{  
symbol = Obtain(ch_value1);  
current_state = Transition[current_state][symbol];  
cin >> ch_value1;  
} // terminates text of while-loop  
char char_value2;  
char_value2 = display(current_state);  
// Test whether this last state is a member of the set  
// of final states.  
set<char>::const_iterator qptr;  
qptr = final_states.find(char_value2);  
if(*qptr == char_value2)
cout << "Accepted" << endl;
else
  cout << "Rejected" << endl;
return 0;
} // terminates text of main function

5. Overview and Criticism of the Previous Versions of the Simulator.

The version of the simulator presented in the last section generalizes the version presented in Section 3 in several ways. The original version of Section 3 simulates one single DFA over the alphabet \{ /, -, o\} with states \{ i, d, h, n\}, namely the DFA whose transition map is defined by the table in (Figure 1), and with final states \{ d, h\}. The simulator defined in Section 4 again uses the same alphabet and set of states, but allows more flexibility in that the user can supply any transition map using that alphabet and set of states, and may also define any nonempty subset of final states, so long as these final states come from \{ i, d, h, n\}. The entries in the transition table, and in the set of final states are input interactively.

Thus, the simulation program of Section 4 goes one step further than its counterpart in Section 3. But it is not enough: we seek further flexibility. Ideally, we would like to produce a general simulator program that does not assume that the alphabet is \{ /, -, o\}, and that the set of states is \{ i, d, h, n\}. Instead, we seek to design a simulator program for any DFA, regardless of the underlying alphabet and set of states. In order to accomplish this, the user will have to provide the characters of the alphabet interactively, as the individual members of some input string. As well, the definition and construction of the Transition array would no longer involve entries from some predefined enumerated type of states, but would instead involve int-valued entries. This would imply that the set of states of the machine be represented as consecutive non-negative integer entries, with 0 serving as the designated initial state. As a result, the set of final states of the DFA would be represented as a non-empty subset of the set of states of the DFA, causing the set of final states to be a finite non-empty set of non-negative integers.

6. Designing the General Simulator Program.

It is clear that in order to design the general simulator program, we can no longer use the specific pair of enumerated types for the states and alphabet of the DFA used earlier. To attain the generality we seek, we use an integer-valued encoding of the alphabet and set of states. However, certain objects can be carried over from our earlier work, most notably the simulation of the transitions of the DFA as a two-dimensional array, and the set of final states as a specific object definable from the set library available in STL, but now with int-valued entries.

Since specific enumerated types for States and Alphabet will not be available in this general setting, we must view these as types consisting of finite sequences of consecutive non-negative integer values. For instance, we will replace the Alphabet
type consisting of the identifiers SLASH, DASH, OTHER by the int values 0 (for SLASH), 1 (for DASH), and 2 (for OTHER). Similarly, the earlier description of States as the enumerated type consisting of the identifiers INITIAL, DIGIT, HYPHEN, NOGOOD will have to be replaced by the respective int values 0 (for INITIAL), 1 (for DIGIT), 2 (for HYPHEN), and 3 (for NOGOOD). In other words, in this more general setting, these identifiers must be translated by this integer-valued encoding.

Therefore, our design of the general simulator will require that the user must be prompted to input the specifications of the underlying DFA by first supplying the number of states. If we wish to consider the DFA described earlier, the user will respond to the prompt by inputting 4. Secondly, the user will be prompted to supply the number of alphabet characters: if the DFA described earlier is used, the user responds to this prompt by inputting 3. However, this does not give any idea as to what the actual alphabet characters are. In order to input the actual characters of the alphabet, the user must be prompted to input these as the members of a character string. Again, if we refer to the previous DFA, the user will respond to this prompt by inputting the character string "/-o" (for SLASH, DASH, OTHER), respectively, or any other permutation of these three characters.

We now treat the construction and initialization of Transition in this more general setting. As before, Transition will be defined as a two-dimensional array, but now the array will have to be "overdimensioned" to 10 rows and 10 columns, since there is no predefined number of states nor alphabet characters. In addition, the components of Transition will be of the int type, due to the int-valued encoding of the states of the underlying DFA. The construction of the Transition array is accomplished using the code

```cpp
int Transition[10][10];
```

This implementation is then capable of handling transitions of any DFA with a maximum of 10 states, and with an alphabet consisting of a maximum of 10 distinct characters. Assume States_Size and Alphabet_Size are int variables representing, respectively, the number of states and the number of alphabet characters of some DFA, and that their respective values have been provided interactively by the user. Then the following code can be used to input, one row at a time, the values of the transition table:

```cpp
cout << "Please input the rows of the transition table:" << endl;
for(int row = 0; row < States_Size; ++row)
{
    cout << "Please input the next row of the transition table" << endl;
    for(column = 0; column < Alphabet_Size; ++column)
    {
        int int_value;
        cin >> int_value;
        Transition[row][column] = int_value;
    } // terminates text of inner for-loop
} // terminates text of outer for-loop
```
It will also be necessary to input the set of final states interactively. This will be done in a manner similar to the previous simulation program, except that the set<char> final states object will be replaced by a set<int> object. We accomplish this by first constructing final_states as an initially empty set<int> object, using the constructor

```cpp
set<int> final_states;
```

Then the code sequence

```cpp
cout << "Please input the final states of the DFA" << endl;
cout << "on separate lines, terminating with -1:" << endl;
int int_value;
cin >> int_value;
while(int_value != -1)
{
    final_states.insert(int_value);
    cin >> int_value;
} // terminates text of while-loop
```

completes the input of all of the final states, coded as int values, into the set final_states. Thus, if we input the sequence

```
1
2
-1
```

final_states will contain the two int values 1 and 2.

The next task is to input the actual characters of the alphabet. This will be done by inputting the alphabet as the individual characters of a string. In order to accomplish this in C++, it will be necessary for the program file to include the preprocessor directive

```cpp
#include <string>
```

to facilitate and to enable the processing of character strings.

The formal code sequence for inputting the characters of the alphabet may then be given by

```cpp
// Prompt user for the alphabet of the DFA as a character string.
cout << "Please input the alphabet of the DFA as" << endl;
cout << "the individual characters of a string:" << endl;
string str_value;
cin >> str_value;
```

Now that the alphabet has been input as the current value of str_value, we can use some additional features of the string data type. In particular, there are three functions that operate on strings that will be very helpful: length, find, and at. We give a brief summary of the functionality of each:
1. the length() function, when applied to a string variable, returns an unsigned integer whose value equals the number of char values currently in the string, counting any possible repetitions. For example, if str_value is a string variable, then str_value.length() returns the length of the current contents of str_value;

2. the find function searches a string to find the first occurrence of a particular substring, named as the single parameter, and returns an unsigned integer giving the result of the search. As was the case of the length function, find must be applied as a qualified reference, and also requires a single parameter, appearing as a substring. In our particular application, the substrings will be single characters: the specific and distinct characters of the alphabet of the DFA. We will only require the application of find in conjunction with the at function, which we discuss next;

3. the at function allows us to access any individual character of a given string. Therefore, the at function must be applied as a qualified reference to any string variable, and requires a single non-negative int value, where that value yields the position of the character in the string. We will use this auxiliary function in conjunction with the find function, in the following manner:

If we assume that str_value currently stores “/-o”, then the code sequence

```cpp
for(int str_index = 0; str_index < str_value.length(); ++str_index)
    cout << str_value.find(str_value.at(str_index)) << endl;
```

outputs

```
/ - o
```

Thus, applying this code to the input string “/-o” establishes the encoding

`/` corresponds to 0

`-` corresponds to 1

`o` corresponds to 2

It is important to note that the alphabet of the DFA can be any finite string of characters, not just the string “/-o”. The only constraints are that no character can be repeated in the string, and (currently) the length of the string must be ten characters or less. However, the size of the alphabet may be enlarges by a single editorial change in the code defining the Transition array, followed by a recompilation of the resulting formal code. In addition, we should note that the order of appearance of the characters in the alphabet comprising str_value also affects the components of both Transition and final_states.
6. Simulating the Action of the DFA.

The code sequences described in the last section prepare the DFA to begin its formal computation. We may 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 '§' used to signal 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 individual characters of the input string, terminating with the character '§'.

The following code sequence simulates the action of the DFA:

```c++
// Initialize the action of the DFA.
int state = 0;
// Prompt user for input string.
cout << "Please input any string, terminating with \$;" << endl;
// Input string interactively.
string input_string;
 cin >> input_string;
// Initialize position of scanner of input string.
int position = 0;
// Keep DFA active until \$ is scanned.
while(input_string.at(position) != '§')
{
    // Look at rest of input string.
    input_string = input_string.substr(position, input_string.length());
    // index_value holds integer code of current alphabet character
    char ch_value = input_string.at(position);
    int index_value;
    index_value = input_string.find(ch_value);
    // Go to state determined by Transition
    state = Transition[state][index_value];
    // Scan next character of input string.
    ++position;
} // terminates text of while-loop
// At this point, input_string has been completely scanned.
// Now test whether this last state is a member of the set
// of final states.
if(*final_states.find(state) >= 0)
    cout << "Accepted" << endl;
else
    cout << "Rejected" << endl;
```

Thus, after the entire input string is scanned, state contains the int value representing the state of the DFA at that point. The built-in iterator of find scans the int-valued members of final_states. If a match is found, that is, if there is a match between this last state when the entire input string is scanned with one of the members of final_states, then the input string is generated by the DFA. Consequently, the input string is accepted. Under any other circumstances, the input string is rejected.
Here is the formal code of the entire program file:

```cpp
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main()
{
    // Declaration of a variable for the number of states of the DFA
    int States_Size;
    cout << "Please input the number of states of the DFA" << endl;
    cin >> States_Size;

    // Declaration of a variable for the number of distinct alphabet
    // symbols of the DFA
    int Alphabet_Size;
    cout << "Please input the number of distinct alphabet symbols" << endl;
    cout << "of the DFA" << endl;
    cin >> Alphabet_Size;

    // Declaration of Transition as a two-dimensional array.
    int Transition[10][10];

    // Prompt user for the rows of the Transition table
    cout << "Please input the rows of the transition table:" << endl;
    for(int row = 0; row < States_Size; ++row)
    {
        cout << "Please input the next row of the transition table" << endl;
        for(int column = 0; column < Alphabet_Size; ++column)
        {
            int int_value;
            cin >> int_value;
            Transition[row][column] = int_value;
        } // terminates text of inner for-loop
    } // terminates text of outer for-loop

    // Construct a set of int values for the final states of the DFA
    set<int> final_states;

    // Now input the final states of the DFA into the set final_states
    cout << "Please input the final states of the DFA" << endl;
    cout << "on separate lines, terminating with a -1" << endl;
    int int_value1;
    cin >> int_value1;
    while(int_value1 != -1)
    {
        final_states.insert(int_value1);
        int_value1;
    }

    return 0;
}
```

cin >> int_value1;
} // terminates text of while-loop

// Prompt user for the alphabet of the DFA as a character string.
cout << "Please input the alphabet of the DFA as" << endl;
cout << "the individual characters of a string:" << endl;
string str_value;
cin >> str_value;

// Initialize the action of the DFA.
int state = 0;
// Prompt user for input string
cout << "Please input any string, terminating with a $" << endl;
// Input string interactively
string input_string;
cin >> input_string;
// Initialize position of scanner of input string.
int position = 0;
// Keep DFA active until $ is scanned
while(input_string.at(position) != '$')
{
    // Look at rest of input string.
    input_string = input_string.substr(position,input_string.length());
    // index_value holds integer code of current alphabet character
    char ch_value = input_string.at(position);
    int index_value;

    index_value = input_string.find(ch_value);
    // Go to state determined by Transition
    state = Transition[state][index_value];
    // Scan next character of input string
    ++position;
} // terminates text of while-loop

// At this point, input_string has been completely scanned.
// Now test whether this last state is a member of the set
// of final states.
if(*final_states.find(state) >= 0)
cout << "Accepted" << endl;
else
cout << "Rejected" << endl;

return 0;

7. Consequences and Suggestions for Further Research.

This paper parallels the research done earlier using Java 6.0, resulting with the
publication of [1], [2], and [3]. However, the present paper performs this work using
C++ as the implementation language, with particular emphasis on the facilities provided
by the Standard Template Library. The original research was conducted using Pascal,
and appeared in [4].
There remains a vast area for future research in simulating finite automata using either C++ or Java as the implementation language, as well as other possible choices of object-orientated implementation languages. For example, one area for future research involves the problem of simulating the conversion of a nondeterministic finite automaton used as a language generator into an equivalent deterministic form, following along the lines of the CONVER program of [4], which uses Pascal. In an earlier paper (see [5]), it was shown, using the features of object-oriented design as implemented in C++, that any DFA generating a specific language can be made equivalent to a minimal-state DFA achieving the same result. However, a Java encoding of this same result has yet to be accomplished.

Besides these, there still remains the entire area of the simulation of pushdown automata and the simulation of the activity of Turing machines, particularly with regard to their importance in testing the computability of certain important number-theoretic functions.

8. 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