Design and Implementation of Finite State Automata Shell

Yasmeen F. Al-Ward

Computer Science Dept.
College of Science
Al-Nahrain University

Abstract

Finite State Automata, FSA, represents a main solution-model for different computational problems. The implementation of FSA for specific serious problem consumes times and efforts. The aim of this research is to design and implement a system shell which rapidly builds an FSA. This system involves no knowledge about the FSA underhand, but includes the general mechanism of parsing the input string in addition to a sub-system to collect the specification of a specific FSA. According to this shell, the building of a FSA requires no more than the time of presenting its specifications during what so-called construction phase. In the construction phase the tables of the FSA specifications are built such as the transition table which represents the transition function. In the second phase, the Application Phase, the machine becomes ready for use. The first phase includes many modules such as Specifications-File Reader and Machine Constructor modules, while the Application Phase includes Input-String File Reader, Parser Shell, and Output Visualizer. Many actual and randomly-generated machines had been used to test the shell. These FSAs are of 1 up to 100 states and 1 up to 39 alphabet symbols and they were built and tested successfully.

1. Introduction

Finite Automata is one of the oldest computational machines in the computer theory. It consists of input string, automaton, and output. The string represents sequence of events, program instructions, language lexemes or tokens, etc. The automaton, i.e. the computer, consists of a set of states. The output is binary output; i.e., the input string is acceptable or not. FSA represents the natural way to solve a problem. A problem has an initial state and there are some operations that can be applied on. When a suitable operation, (events, possible move, instruction, etc.), is carried on the initial state will be changed to a new one. This process is continued until the final state is reached, i.e. the solution of the problem. Naturally a problem has more than one solution which imposes more than one path from initial state to final state(s). Therefore, the FSA can have more than one final state. This natural manner of modeling leads to use the FSA as main tool to model the state space of a problem in the AI **[**1**]**, i.e. FSA is used in the problem-solving, the main field of AI. FSA is used also to represent the definition of the identifiers and reserved word in the lexical analyzer of programming language. Any programming languages translator, (compiler, interpreter, assembler, preprocessor, etc), depends on FSA to define the syntax of its reserved words and identifiers [2,3]. FSA is used widely in natural language understanding, NLU, equally in the design of NL interfaces or machine translation systems [1,4]. Another useful benefit of FSA is that its use to model the sequential digital devices, such as memory cell, counters, multiplexer, etc [5]. The FSAs are used in the editor design and string matching [6,7]. Many researchers depended on FSA in different fields of computer design such as computer security and software engineering [8, 9, 10]. The most recent applications of FSM is that its use in DNA computing and DNA analysis [11].

1.1 Formal definition

There are many publications about finite automata which describe its definition, complexity, method of implementation, drawbacks, applications, etc such as [12, 13, 14]. Therefore, this paper will concentrate on the aim of the research, i.e., the design of the shell system. It will give a much abbreviated introduction and the reader can refer to these publications above.

Deterministic FA is defined in [12, 13] as a quintuple M=(Q, $\sum$, δ, $q0$, F), where Q is a finite set of states. $\sum$ is an alphabet. $q0 \in Q$ is the initial state. $F \subseteq Q$ is the set of final states. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is defined such that $\forall q \in Q, \forall c \in \Sigma, \delta(q,c) \in Q$ is uniquely determined.

Figure (1) depicts the machine which accepts the identifier of most programming languages. It is a sequence of a letter followed by any sequence of letters, digits, or underscore such as there is no two adjacent underscores. The formal definition of this machine is as follows:

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q={State0, State1, State2}.

$\sum$=a, …,z, A,…, Z, 0,…,9, _}.

δ ={State0 ×Letter◊State0,
State0 ×Digit◊State1,
State0 ×Underscore◊State1,
State1 ×Letter◊State2,
State1 ×Digit◊State2,
State2 ×Letter◊State2,
State2 ×Digit◊State2,
State2 × Underscore ◊State2}

Letter, Digit, and Underscore are alphabet character categorization. The categorization is an optional abstraction of the transition function which diminishes the description of the machine. The complexity of string

recognition is the sum of the complexities of the transition of each character in the string [14].

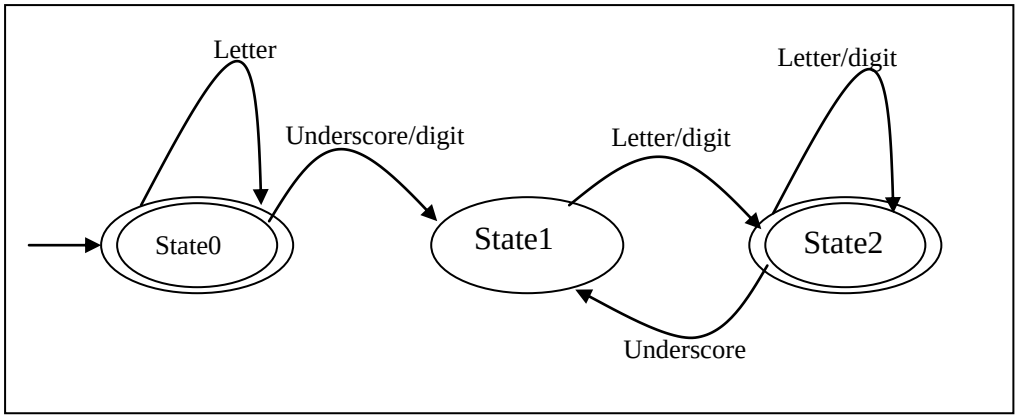

Figure (1) Identifier Definition Finite Automata

1.2 Research Objective

In spite of the importance and old existence of the automata, its implementation remains as domain-dependent implementation, i.e., the designer needs to implement the problem model newly for each new problem. This process consumes time and programming efforts. Many researchers attempted to produce tools to reduce these problems. Ngassam et. al. [15, 16, 17] suggested hard coding implementation of the FSA. But this approach is processor-dependent implementation and requires machine low level knowledge. Also, this approach changes the machine specifications from data to program instructions; therefore the execution time will be increased. Forlan [18] is a language to define and manipulate Regular Expressions REs. The REs and FA are equivalents, but FA has more readability, understandability, and provides natural method of problem solving models.

This research presents the details of the design and implementation a FA shell. This shell includes no knowledge about any machine but includes pre-programmed modules. When the shell is filled with the

specification of a machine, the shell will be a recognizer for the accepted strings. This shell is code-reusable system which diminishes the time and efforts of the implementation of the FSAs.

2. FSA-Shell Architecture

The shell's architecture is depicted in figure (2). It consists of many modules such as Machine Specification Reader, Machine Constructor, String Reader, Output Visualizer, and Parser Shell.

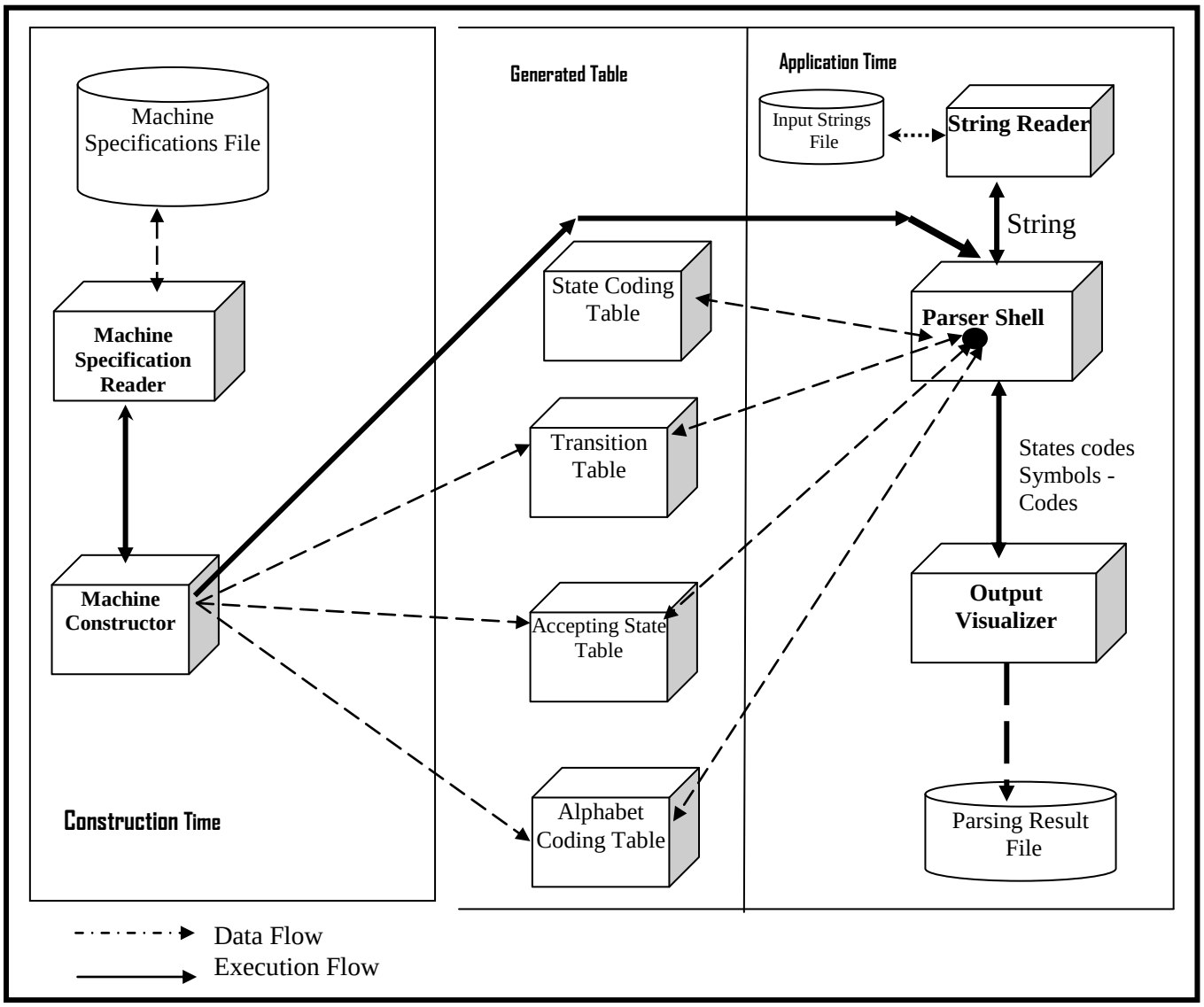

Figure (2) FSA-Shell Architecture

Some modules are related to the time of machine construction and the others are working at parsing time, i.e., post construction time. The user should determine the specifications of the machine and store them on a file called *machine Specification File*. The specifications will be read and analyzed by the Machine Specification Reader. The Machine Constructor module depends on the specification items sent by previous module to construct the machine.

The machine will be ready to use after the construction of Transitions Table (TT), Accepting State Table (AST), and Alphabet Coding Table (ACT) according to number of transitions, number of accepting states, and number of alphabets respectively. In this way the construction time is accomplished and Parsing Time is beginning.

The Parser Shell (PS) receives one string at a time from the String Reader (SR) and parses it according to the constructed features. The result of parsing will be sent to Output Visualizer (OV). From the design point of view each module is implemented as class or sub-class. The next section explains the design of each module in some details.

2.1 Construction Phase Modules

This module is responsible for reading and recognition of the specifications of a machine.

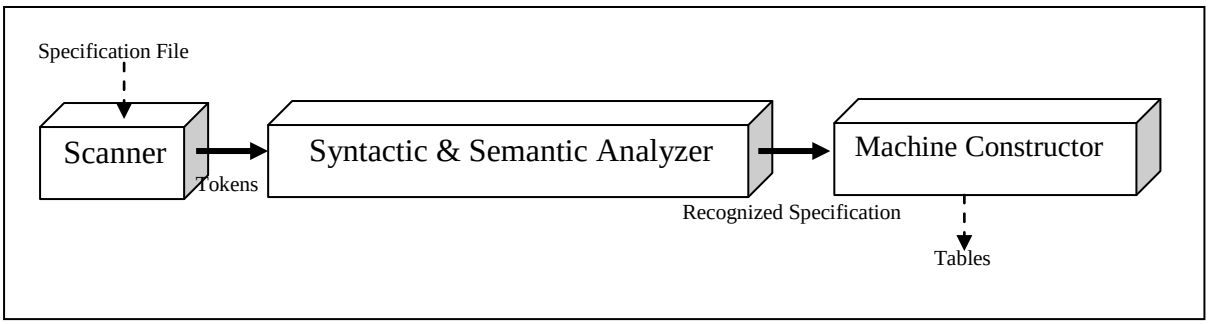

Figure (3) Sub-Modules of MSR

It is implemented as a class which depends on Java language. MSR can be divided into two sub modules that are scanner and syntax analyzer

which connected with Machine Constructor. Figure (3) depicts the sub-modules of MSR. The scanner sends stream of tokens which will be checked according to the predefined grammar by the syntax analyzer. According to the syntactically and semantically correct specifications many dynamic tables will be generated such as transitions tables, accepting states table, state coding table, and alphabet coding table. These tables are generated by tables' generator sub-module.

NofS=INTEGER --No. of States
{$STATE_0$
$STATE_1$
$STATE_2$
…
$STATE_{N}$}

NofA=INTEGER--No. of CHARACTERS in alphabet
{C0, C1, …, Cm}

[NofG=Integer --No. of Categories
{g0, g1, …, gk}]

NofAS=INTEGER --No. of Accepting States
{$STATE_I$
$STATE_J$
$STATE_L$
…
$STATE_{N}$}

NofT=INTEGER --No. of transitions
{$TRANSITION_0$
$TRANSITION_1$
…
$TRANSITION_L$ }

Figure (4) The General Structure of Machine Specification File

The machine specifications file contains the following specifications: Number of states, number of accepting states, lists of states and accepting states, Number of alphabet's characters, and number of transitions, and transitions table. The general structure of specification file is shown in

figure (4) and the grammar of the notation used to write the specification is shown in figure (5). This grammar is expressed as BNF notation.

The reserved words in the specifications file are NofS, NofA, NofAS, NofG, and NofT which mean number of states, number of characters in the alphabet, number of accepting states, number of categories, and number of transitions respectively. Some of machines contain categories for the symbols of the alphabet; therefore the number and definition of the categories are optional. Also, the notation involves some reserved characters such as '=', '{','}', and ','

```
<SPECIFICATION>::=NofS=INTEGER {<STATE-LIST>}<OPTIONAL>
                        NofAS=INTEGER {<STATE-LIST>}
                        NofT=INTEGER {<TRANSITION-LIST>}

<OPTIONAL>::= NofA=INTEGER {<CHARACTER-LIST>} |
                    NofA=INTEGER {<CHARACTER-LIST>}
                    NofG=INTEGER <CATEGORY-LIST>

<STATE-LIST>::=<STATE>|< STATE>, <STATE-LIST>
<STATE>::=user-defined-state

<TRANSITION-LIST>::=(<STATE>,<CHARACTER> ◊<STATE>)|
                (<STATE>,<CHARACTER>◊<STATE>) <TRANSITION-LIST>

<CHARACTER-LIST>::=<CHARACTER>|
                <CHARACTER><CHARACTER-LIST>

<CHARACTER> user defined character or string
<CATEGORY-LIST>::=ID={CATEGORIES}|
                ID={<CATEGORIES>}<CATEGORY-LIST>

<CATEGORIES>::=user-defined-category  |
            user-defined-category  ,< CATEGORIES>
```

Figure (5) the BNF of Machine Specification File

The syntax analyzer was designed according to the grammar presented in figure (6). The analyzer presents error massage for each syntactically or semantically erroneous description. Figure (7) depicts the specifications of the machine presented in Figure (1). This machine

contains three categories of alphabet symbols; Letters, Digits, and Underscore. Therefore the specifications files involves NofG clause which defines number of categories followed by the definition of each category. From FSA design point of view, the categorization eases the design and theoretical description. But it represents a challenge for the FSA-Shell. According to the example presented in figure (1), the specifications file involves eight transitions, but indeed the number of the transitions equals 188 transitions, because the letter transition will be converted to 2*26 transitions for each occurrence of its occurrences, while digit transition will be converted to 10 transitions for each occurrence. For more explanation, another example is presented in figure (8). This machine involves no categorization; therefore the mentioned transitions in the file will be the generated ones in the transition table.

```
NofS=3
    {state0, state1, state2}
NofA=38
NofG=3
    Letters ={a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u,
              v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M,
              N, O, P, Q, R, S, T, U, V, W, X, Y, Z }
    Digits ={0,1,2,3,4,5,6,7,8,9}
    Underscore={_}
NofAS=2
    {state0, state2}
NofT=8
    {(state0, letters)->state0,
    (state0, digits)->state1,
    (state0, underscore)->state1,
    (state1, letters)->state1,
    (state1, digits)->state1,
    (state1, underscore)->state2,
    (state2, letters)->state1,
    (state2, digits)->state1}
```

Figure (7) Machine Specification of Figure (1)

There are many generated tables which represent the machine specifications. These tables will be used in the next phase, i.e., application time phase. The FSA-shell should be able to manipulate unlimited specifications such as the number of symbols and number of states. Therefore, Tables' Generator performs two jobs; coding process and table generating depending on the first job, and for same reason these tables are dynamic tables to hold deferent machines specifications.

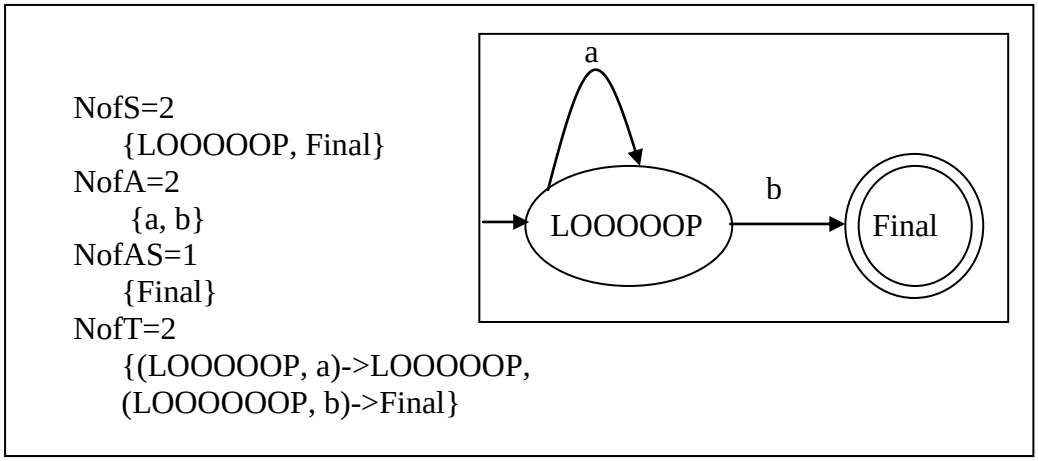

Figure (8) a* b Machine with its specification file

Table(1) State Coding Table	
State	Code
State0	0
State1	1
State2	2

Table(2) Alphabet Coding Table	
Symbol	Code
a	0
b	1
c	2
…	…
z	28
0	29
…	…
9	38
_	39

Table(3) Accepting State Table	
State	Code
State0	0
State2	2

In the coding process, each state will be given a unique number. Also, each symbol of the alphabet will be given a unique number. Therefore, state coding table consists of two fields; state and code.

The *alphabet coding table* involves two fields; Symbol and code. Tables (1), (2), and (3) depict these coding tables according to the machine illustrated in Figure (1). As a result the accepting state table will be built depending on the state coding in table (1). The transitions table is composed of three fields; *Coded Source State, Coded Symbol, and Coded Target State*. Hence the generated transition table of the machine of Figure (1) will be as follows:

Table (4)Transition Table of the Machine

Comment (not a part of the table)	Coded Source State	Coded Symbol	Coded Target State
(state0,a)◊state0	0	0	٠
(state0,b)◊state0	0	1	0
(state0,c)◊state0	0	2	0
…	…	…	…
(state0,0)◊state0	٠	29	٠
…	…	…	…
(state0,9)◊state0	0	38	0
(state0,_)◊state1	0	39	1
(state1,a)◊state2	1	0	2
(state1,b)◊state2	1	1	2
…	…	…	…
(state2,a)◊state2	2	0	2
(state2,b)◊state2	2	1	2
…	…	…	٠٠
(state2,0)◊state2	2	28	2
…	…	…	…
(state2,9)◊state0	2	38	2
(state0,_)◊state1	2	39	1

After generating the tables above, the *construction phase* is accomplished and the AF-Shell system becomes ready to check the input strings, i.e., the *application phase* is beginning. In the application phase there is no need for the modules of the previous phase. The active modules in the application phase are String reader, Parser Shell, and the Visualizer. The next section presents the design of these modules in some details.

2.2 Application Phase Modules

This phase consists of three modules input strings reader, parser shell, and output visualizer. The first module reads one string at a time from the input strings file. This string will be sent to the shell parser to check its acceptance and the result of the checking will be sent to the visulalizer to be written on the output file in special manner.

2.2.1 Input string Reader

The input of this module is batch file of input strings. Each string is enclosed between two braces; '{'and '}'. The symbols of the string are separated by spaces or a user-defined symbol.

```
[] string get symbol ()
{
        While not end-of-file;
        {       string symbol="";
                Char ch;
                While not-end-of-line and ch!=' ' do
                        While char ch=getchar(InputStringsFile)!=' ' do
                                symbol= symbol + ch;
                Return symbol;
        }
}
```

Figure (9) Token Constructing Algorithm

The separator is used in spite of the symbol is consisting of one character; in other word the string of single-character symbols is written

as a single word, but in the input strings file is written as separated character.

For example, the string “aaaab” of two letters-alphabet Σ ={a, b} is written {a a a a b}. The multi-character string will be written in the same manner; for example, consider the string {sight smile love} which is obtained from the alphabet Σ ={*sight*, *smile*, *hate*, *love*, *heart*, *mind*}. The reading process is described in figure (9). This module can be regarded as a procedure in the parser module. The shell parser frequently calls this module to present a symbol. This symbol will be used to find a transition which consists of the current state and the obtained symbol. The end of line character represents the end of the input string and the beginning of next string. When end of line is encountered, the parser should reach an accepting state if the string is accepted; otherwise the string is rejected string. This module is designed depending on the java.IoStreame package.

2.2.2 Parser Shell

The core of the FSA-Shell system is the parser shell. It is called shell because it is empty, i.e., it has no domain-specific knowledge about the under hand FSA. It is filled with such information during the Construction Phase. It is very similar to the Expert System Shell.

The parser Shell checks many cases to decide the acceptance or rejection of a string. These cases are:

a) *If the current symbol is the last symbol in the input string and the current state is an accepting state, the string is accepted.*
b) *If the current symbol is the last symbol in the input string and the current state is not accepting state, the string is will be rejected.*
c) *If the current symbol is not the last symbol in the input string and there is no transition from the current state with this symbol, the string will be rejected.*

d) If the current symbol is not the last symbol in the input string and there is a transition from the current state with this symbol, then fetch next symbol.

Figure (10) depicts the skeleton of the parser shell. Line No.2 invokes the SR module to fetch new symbol. Line No.3 finds the code of the symbol depending on the *binary search* or depending JDBC according to the choice of the user.

```
…
 0 While not-end-of-file(InputStrings)
 1 {
 2    Read next symbol from InputStringsFile, call String
        Reader;
 3    Find the code of the next symbol in the symbols-codes
        file
 4   If no code for the symbol then
 5   {   --present error massege to the user
 6      Send Report to OutputVisualizer Module
 7      Return 0; -- Rejected String
 8   }
 9  Find the code of the current state in the states-codes
        file
 10  Find the transition (current-state, Symbol)◊Next-State in
         the Transition Table;
 11   If no transition then
 12   { Send Report to OutputVisualizer Module
 13     return 0; -- Rejected String
 14   }
 15 }
 16 If not-end-of-file(InputStrings)
     --Check Current-State in Accepting states table
 17   If accepting_state(AcceptingStateTable)
 18   {   Send Report to OutputVisualizer Module
 19        Retern 1 –accepting state
 20   }
 21   Else
 22   { Send Report to OutputVisualizer Module
 23      Return 0 --Rejected String
 24   }
……
 25 } --ParsingShell
```

Figure (10) Parsing Shell skeleton

If there is no code for the symbol, this means that the string is written incorrectly, and the shell presents an error message to the user and the input string is rejected.

Line No.9 finds the code of current state which is stored in the states-codes table, also this done by binary search or JDBC. In Line No.10, the shell searches for next move depending on the symbol and the current state. Steps No.11-14 check case c above and send a message to the OV. If the symbol is the last symbol, i.e., the end of line is reached, and the current state is an accepting state this means that the string is accepted state. This checking is done by Step No.16 to step No.20. If the current state is not an accepting state, this means that the string is rejected string. The determining of a state depends on a function called accepting_state which searches in the accepting states table to find the state in this table. This process continues until the end of file character is read, i.e., all the strings are parsed.

2.2.3 Output Visualizer Module

The parser shell sends data about each parsing step to this module. This data includes current symbol, its code, and its type. Also it involves the current state and its code in addition to the target state if any. This data is used to draw the path of the parsing from the initial state to final state(s), if the string is accepted state, or to a target state of a given transition.

3. Discussion and Conclusions

There are many notices, conclusions, and recommendations that can be considered in this work, some of these are:

- The shell manipulates deterministic finite automata only. As a future development, it is possible to modify the shell in order to handle nondeterministic finite automata (NFA). This development

can be accomplished by two types of modification. The first one is related to modify the parser module, but this approach will complicate the design of the parser and deteriorate the performance of the shell because the execution time will increased according to the number of nondeterministic states. This approach can be implemented depending on depth first search algorithm, [1,4], to find all the possible paths from start state to one of accepting states. The second approach can be accomplished by designing NFA-DFA converter according to NFA-DFA converting algorithm[12,13]. The NFA will be converted to DFA and then will be constructed by the shell.

- It is well-known that FSA is closed under many operations such as complement, intersection, union, and reverse. An important and easy future modification is design of modules to perform these operations and the results can be handled directly by the shell without any modification. Note that the current shell can manipulate the complemented and reversed machines as its now without any extra code.
- The generated tables, in the first prototype of the AF-Shell, are implemented as dynamic tables. These tables consume large amount of MM especially when the under consideration FSA has high specifications. Also, the parser needs to frequently call the binary-search to check the availability of (state, symbol) in the transition table. The coding tables are required to be accessed by using binary search to specify the code of the symbol and the code of the state. In the second prototype, the tables are created as MS Access or SQL tables and will be accessed by using JDBC. In this way the MM consumption and comparison operations which done by binary search will be carried on the shoulders of DBMS. But

this implementation needs some extra user knowledge related to JDBC preparation. Also, JDBC implementation allows the expert user to modify and update the tables to make the table tuning and machine developments directly without the modification of machine specification file. Anyway, the user is enabled to choose one of these implementations.

-

تصميم وتنفيذ نظام صدفي لماكنة الحالات المنتهية

ياسمين فرحان الورد
قسم علم الحاسبات/ كلية العلوم
جامعة النهرين

الخلاصة

ان ماكنة الحالات المنتهية هي طريقة النمذجة الفضلى للمسائل الاحتسابية. ان تصميمها وتنفيذها لنمذجة مسألة معينة تستهلك وقتا وجهدا كبيرين. الهدف من هذا البحث هو تصميم وتنفيذ نظام قشري، خالي من البيانات، لبناء ماكنات الحالات المنتهية دون جهد او وقت يذكر سوى زمن اعداد خصائص الماكنة المعنية. ان النظام يحتوي آلية الاعراب العامة للسلسة المدخلة التي يمكن تطبيقها على اي ماكنة بغض النظر عن خصائصها بالاضافة الى النظام الضمني لتجميع الخصائص. لذا فان زمن بناء الماكنة وفقا لهذا النظام هو زمن تحديد خصائصها ضمن طور البناء وبعدها تصبح الماكنة جاهزة للاستخدام وهذا ما سمي بطور التطبيق. خلال طور البناء تبنى العديد من الجداول التي تمثل خصائص الماكنة واهمها جدول التنقلات التي تمثل دالة التنقل. الطور الاول يتضمن جزئين رئسيين هما قارئ خصائص الماكنة ومشيد الماكنة. اما المدخل لهذا الطور فهو ملف الخصائص. طور التطبيق يتضمن الاجزاء: قارئ ملف السلاسل، وصدفة المعرب، ومظهر النتائج. ان النظام استخدم لبناء وتجريب العديد من المكائن الحقيقية والمكائن المولدة عشوائيا التي تتراوح حالاتها بين حالة واحدة الى مائة حالة، و عدد رموزها يتراوح بين رمز واحد وتسع وثلاثون رمز. جميع هذه المكائن انجزت بنجاح.

References

[1] George F. Luger, "*Artificial Intelligence: Structures and Strategies for Complex Problem Solving*", 7/E, ISBN-10: 0321263189, ISBN-13: 9780321263186, Addison-Wesley, 2007.

[2] Ravi Sethi, Jeffrey Ulman, "*Compiler: Principles, Tools, and Techniques"*, 2/E, Addison-Wesley, 2006.

[3] Alfred V. Aho, "*principle of Compiler Design*", Addison-Wesley, 1977.

[4] S. Russell and P. Norvig, "*Artificial Intelligence: A Modern Approach"*, 5/E, Prentice Hall, 2005.

[5] David J. Comer, "*Digital Logic and State Machine Design*", Oxfored Press, 2001.

[6] D. E Knuth and J.H Morris, Jr and V. R. Pratt. Fast Pattern Matching in Strings. SIAM J. Comput. Volume 6, 323-350, 1977.

[7] Oracle White Papers, "*Regular Expression A New SQL Capabilities IN Oracle database 10G and 11G*", Oracle Press, 2005.

[8] Gerard J. Holzmann, Maregrate H. Smith, "*An automatic verification method for distributed system software based on model extraction*", IEEE transaction on Software Engineering, Vol. 28, No.4, April, 2002.

[9] C. Michael and A. Ghosh, "*Using Finite Automate to Mine Execution Data for Intrusion Detection*", RAID 2000.

[10] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, " *A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors*", State University of New York, 2004.

[11] Gonzalo Navarro, and Mathieu Raffinot, "*Flexible Pattern Matching In String: Practical on-line search for texts and biological sequences*", Cambridge University Press 2002.

[12] Dexter C. Kozen, "*Automata and Computability*", Springer, 2007.

[13] Daniel I. A. Cohen, "*Introduction to Computer Theory*", 2/E., John-Wiley, 2000.

[14] J. Corbett, M. Dwyer, et. Al., "*Extraction Finit-State Models From Java Source code*", Proc. Int'l Conf. Software engineering, 2000.

[15] E. Ketcha Ngassam, "*Hardcoding Finite Automata*", M. Sc. Dissertation, University of Pretoria, 2003.

[16] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, "*Preliminary Experiments in Hardcoding Finite Automata*", CIAA, Santa Barbara, 299-300, September 2004.

[17] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, "*Hardcoding Finite State Automata Processing*", SAICSIT, Johannesburg, 111-121, September 2005.

[18] "Formal Language Tool", http://people.cis.ksu.edu/~stough/forlan.