International Conference on Computer Systems and Technologies - *CompSysTech'2003*

# RESOLVING NON-DETERMINISM IN NFA

Ivan Stoyanov, Stoyan Bonev

***Abstract:*** *The paper describes the authors' experience in simulating non-deterministic finite state automata (NFA) using concurrent programming. The non-determinism is resolved simultaneously by activating a separate thread for all possible transition paths. The approach under discussion may be used to implement specific recognizers in the practice of language processors writing.*

***Key words:*** *Language Processors, Recognizer, Non-Deterministic Finite State Automata.*

## 1. Introductory Terms

There exist two alternative approaches to describe formal languages as infinite sets of character strings [1, 3, 4]: Synthesis and Analysis. The synthetic one is based on the concept of a grammar used to generate syntax correct sentences. The analytic one is based on the concept of a recognizer used to scan input strings accepting them as valid sentences or rejecting them as invalid sentences. The typical recognizer structure includes an abstract device (primarily implemented as a software routine) known as a finite state machine (finite automaton) FSA. FSA are classified as deterministic whose transition function permits one only transition from a concrete state on a certain input symbol and non-deterministic whose transition function makes possible more than one transition from a certain state on the same scanned input character.

It is well known that both deterministic and non-deterministic Finite State Automata may be applied for recognizing the same regular sets. As [1] wrote, "while deterministic FSA can lead to faster recognizers than non-deterministic FSA, a deterministic FSA can be much bigger than an equivalent non-deterministic FSA".

This paper presents a program implementation of a non-deterministic finite state automata (NFA) using thread techniques to model the concurrency in multiple state transitions paths.

## 2. Theory Extract on NFA

The theoretical specification of finite state automata is documented [1, 2, 3 and 4] as 5-tuple formal system. The definition of a non-deterministic FSA is = ($\Sigma$, Q, $\delta$, $q_0$, $Q_f$), with input alphabet $\Sigma$, set of internal states Q, transition function $\delta$ presenting the mapping of $\Sigma x Q \rightarrow 2^Q$ (into subsets of Q), an initial (starting) state states $q_0$, and a set of final states $Q_f$.

## 3. Program Implementation

The practical implementation presented here is based on applying the Object Oriented Programming principles. The object-oriented model of the NFA implemented includes definition and processing with single and multiple instances of the classes described in table 1.

The object-oriented design was chosen to be implemented in Java (jdk1.4.1). Each state of the automata is a distinct object – an instance of class *Node*. The only data members are a name and two flags, showing whether the state is a starting or a final one.

The states of a NFA are connected with arcs - instance of class *Arc*. Arcs are transition paths that allow changing of the current state under a certain input character. Correspondingly, the class *Arc* has data members showing the source and the destination *Node* as well as the input character that is required to make the move (label).

*table 1*

| Class | Description | Instance |
|---|---|---|
| nfa | Used for parsing and storing the NFA 5-tuple. | single |
| nfaImpl | The main class – contains the actual objects (Arcs and Nodes) as well as methods for accessing them. | single |
| Node | Represents a NFA state. | multiple |
| Arc | Represents a NFA transition path – with source and destination Nodes. | multiple |
| nfaThread | The parsing thread that is created each time when there is a non-determinism. Contains the logic of NFA actions. The most important class. | multiple |
| Main | The entry point of the program. Used for creating the first parsing thread. | single |
| Input | Helper class for the input that is being parsed. | single |
| Output | Helper class for logging information. | single |

Each time the parsing process encounters an input character that leads to more than one state, a new *nfaThread* is created to service the occurred non-determinism:

```
new nfaThread(this.getStartupData()).start();
```

The number of threads created on each move is equal to the number of outgoing arcs with the given label (except one, reserved for the current thread). Since the class *nfaThread* extends the base class java.lang.Thread and thus runs in a separate OS-level thread, each *nfaThread* may be in a different NFA state at a given moment and may check a different input character for acceptance. All this is left to the scheduling mechanism of JVM. However, if the input string is acceptable, one of the threads will unconditionally reach a final state of the NFA while checking the last character. This is the only condition for success.

Each thread maintains a list of the nodes passed and passes a copy of this list to any new threads it creates. As a result, upon acceptance, the thread prints a “winning path”, i.e. the path that has been followed to reach a final state under the entire input stream. Note that there may be more that one “winning path”.

```
if (Input.isEndOfInput(charIndex)) {
  if (currentNode.isFinalNode()) {
     //accept the string
     //print the winning path
     //terminate the program
  }
}
```

If a certain thread reaches the end of input, while not in a final state at the same time, it must check if other threads are still working before rejecting the input and terminating the program. If there are active threads, the thread only terminates itself. The thread acts in a similar fashion when it does not have arcs to continue with and end of input not reached. If the thread is the last active one, then it should reject the input and exit the program.

As for the multiprogramming aspect, the simplest synchronization technique known as immutability is used. The objects in an NFA instance (Arcs and Nodes) are predefined and not changed during execution.

## 4. Exploring the NFA

The concept perceived in accordance with the object oriented model described above permits to build a flexible NFA simulator. The flexibility achieved is based on reading an input text configuration file when starting an execution. The contents of the input file serve to initialize the input alphabet, the internal states set, the transition function, the initial internal state and the set of final internal states.

Fig.1 and table2 present the transition graph and transition table of NFA simulating a recognizer for strings derived using a regular expression "(a|b)*ab" (a string composed of letters 'a' and 'b' terminated by an "ab")

*Figure 1*

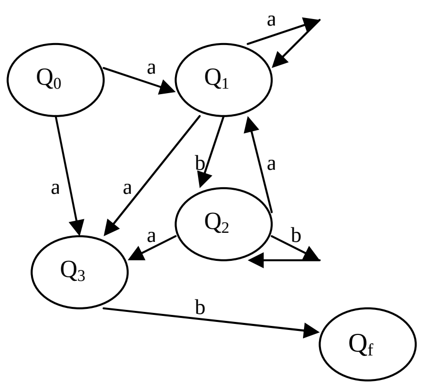

*Table 2*

| | a | b |
|---|---|---|
| $Q_0$ | $Q_1$, $Q_3$ | - |
| $Q_1$ | $Q_1$, $Q_3$ | $Q_2$ |
| $Q_2$ | $Q_1$, $Q_3$ | $Q_2$ |
| $Q_3$ | - | Qf |
| $Q_f$ | - | - |

The input configuration file presenting the NFA described above follows:

```
[input alphabet]
a,b
[set of internal states]
q0,q1,q2,q3,q4
[transiton table]
q1,q3:,
q1,q3:q2
q1,q3:q2
,:q4
,:
[initial(starting) state]
q0
[set of final states]
q4
```

The execution starts by reading the configuration file input.nfa and creating an NFA instance. Then the first parsing thread is started on the initial state. Any subsequent threads are started indirectly by the initial thread. The program creates a log file and each thread writes a message upon startup, change of state and termination.

When given the input string "aaaaaab", the program yields

```
parsingThread-6 succeeded!
Path: q0 q1 q1 1 q1 q1 q3 q4
```

The path displayed is the "winning path" for this execution. The log file generates the following information (the output of each thread is indented with a specific number of spaces):

```
Thread parsingThread-0 starting from state q0
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q1 under input a
 Thread parsingThread-1 starting from state q3
 Thread parsingThread-1 terminating in stateq3 under input a
  Thread parsingThread-2 starting from state q3
  Thread parsingThread-2 terminating in stateq3 under input a
   Thread parsingThread-3 starting from state q3
   Thread parsingThread-3 terminating in stateq3 under input a
    Thread parsingThread-4 starting from state q3
    Thread parsingThread-4 terminating in stateq3 under input a
     Thread parsingThread-5 starting from state q3
     Thread parsingThread-5 terminating in stateq3 under input a
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q2 under input a
      Thread parsingThread-6 starting from state q3
      Thread parsingThread-6 moving to state q4 under input b
```

However, given the input "aabb" and the same NFA, the program returns :

```
No success - end of input and not in the final state.
```

with the corresponding log information:

```
Thread parsingThread-0 starting from state q0
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q1 under input a
Thread parsingThread-0 moving to state q2 under input a
Thread parsingThread-0 moving to state q2 under input a
 Thread parsingThread-1 starting from state q3
 Thread parsingThread-1 terminating in stateq3 under input a
  Thread parsingThread-2 starting from state q3
  Thread parsingThread-2 moving to state q4 under input b
  Thread parsingThread-2 terminating in stateq4 under input b
```

The string was rejected because it does not terminate with "ab".

## 5. Future Development and Conclusion

The non deterministic FSA simulator may be applied when solving problems with non determinism in different practical areas like language processors design and implementation. A classical illustration is the well known non determinism in Fortran syntax where the string

```
DO  122 I
```

can be considered both as the beginning of a loop statement like

```
DO  122 I = 1, 10, 2
```

as well as a part of an assignment statements like

```
DO  122  I  = 1.10
DO  122  I  =  1
```

Solving a non-deterministic problem using parallel programming is a real alternative to the traditional deterministic approach. The authors' purpose was to show how this can be done. Better results may be expected and achieved only on a multiprocessor machine.

## 6. References:

**1.** Aho A., R.Sethi, J.Ullman, Compilers, Principles, Techniques and Tools, Addison Wesley Publishing Company, 1986.

**2.** Bonev S., E.Elmasllari, Implementing Finite State Automata using Object Oriented Programming, Proc. 15th Int. Conference Systems for Automation of Engineering and Research SAER 2001, Sep 21-23, Varna, Bulgaria, pp 172-176.

3. Tremblay J.P., P.Sorenson, The Theory and Practice of Compiler Writing, McGraw Hill Book Company, 1985.

4. Yankov B., Translators and Operating Systems, Sofia, Tehnika Publ., 1993, (in Bulgarian).

## 7. Authors:

*Ivan Dimov Stoyanov BS in Computer Science, American University in Bulgaria, ids200@aubg.bg*

*Stoyan Bonev, Assoc. Prof., PhD., American Unversity in Bulgaria,* sbonev@aubg.bg*, 07388416*