SOME ASPECTS REGARDING THE CONSTRUCTION OF A PUSHDOWN AUTOMATA SOFTWARE EDITOR AND SIMULATOR

George RADUCU[1], Florin-Marian BIRLEANU[2]
Dept. of Electronics, Comp. Sc. and Electrical Eng., University of Pitesti, Romania
[1]r_george_94@yahoo.com, [2]florin.birleanu@upit.ro

Keywords: nondeterministic pushdown automata, context free languages, automata simulation software

Abstract: *The paper presents some key aspects regarding the design and implementation of a didactic software application that allows the user to edit and simulate pushdown automata. It was developed using the C# programming language, as a Windows Forms application based on .NET Framework 4.0.*

1. INTRODUCTION

Pushdown automata (PDA) are the key to understanding the inner working of parsers, which are the core of programming language interpretation and translation. They are basically nondeterministic finite automata with a stack added in order to enlarge the class of languages that they are able to accept. While finite automata are able to accept only the regular languages (that can be described by regular expressions), nondeterministic pushdown automata (NPDA) can accept any context-free language (that can be described by a context-free grammar). This is due to the stack, that allows to keep track of all the open "parentheses" in the input being processed. Understanding the operation of these automata is of great help for students wanting to implement their own formal language parsers. Any context-free grammar can be easily converted to an empty-stack accepting NPDA, which means that one can parse that grammar by simply simulating the automaton.

However, simulating an NPDA is not an easy task neither from a programming perspective, nor from a computational point of view. While there are already simulators for this type of automata [1-5], we took the task of understanding things more in depth by implementing from scratch an easy to use NPDA editor and simulator (as a .NET Framework 4.0 Windows Forms based C# application). The resulting application is useful for students that try to understand clearly the pushdown automata functionality. Besides that, understanding of the challenges and solutions proposed in this implementation can be very helpful. This is why we focus on them in the following sections of this paper.

2. EDITING THE AUTOMATON

In Figure 1 the main application functionality is presented. This flow chart will be next divided into smaller functions until every method will be clearly explained.

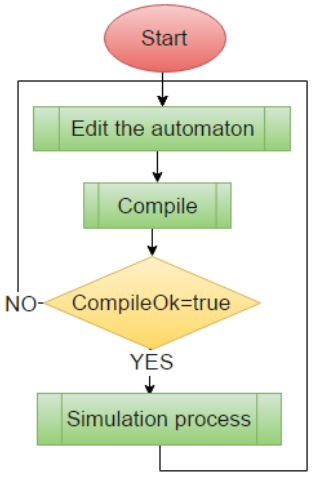

Fig. 1 – The main functionality of the application

In Figure 2 the methods used for editing an automaton are shown.

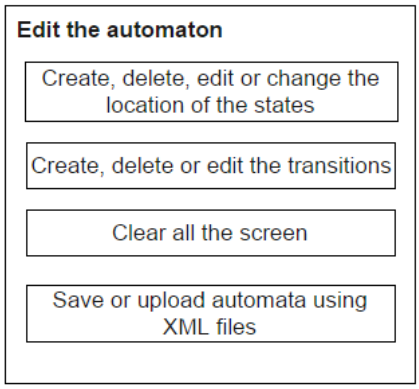

Fig. 2 - Methods used when editing an automaton

The states were represented by inheriting the Panel class from C#, and by setting it to double buffered into the new constructor. To edit the states an event for right-click on the panel was created and there was assigned a context menu strip object so that the user can choose the desired action.

For drawing the transitions three points were determined: the center of the starting state, a point placed approximately in the middle of the distance between the two states, but a little displaced in order to curl the transition, and one point on the destinaton state. In the next step, the Draw.Curve() method was used to finish the task.

Using XML files to save and upload automata is a good ideea because they allow a portable and flexible text-based description of the automata.

3. COMPILING THE AUTOMATON

The automaton compilation process is presented in Figure 3. This step will search for some errors and next it will prepare the memory for simulation. If no errors are found, the list of current states is initialized with the start state of the automaton and a stack containing the initial stack element.

At this point we must introduce two important classes that we used for representing internally the automaton: a pair formed by a state and a stack for representing the current state, and a class for representing a transition, formed by the start state of the transition, the transition label and the destination state of the transition. These two classes are illustrated in Figure 4.

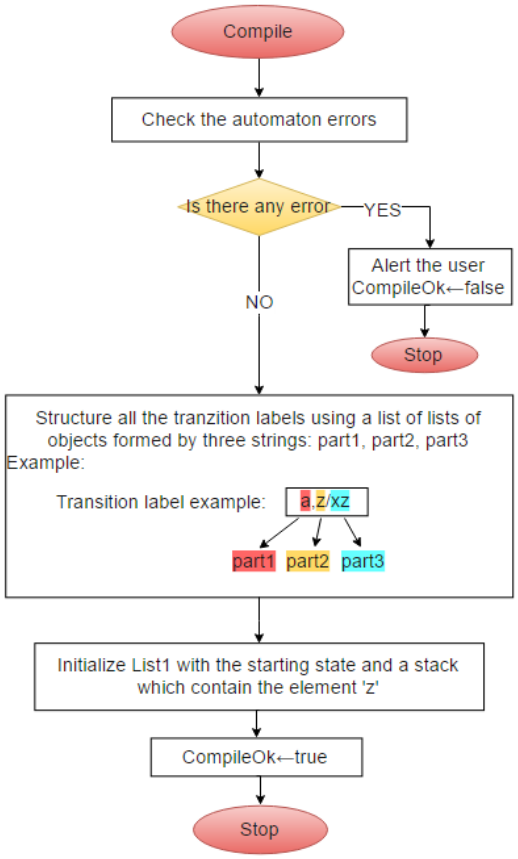

Fig. 3 – The automaton compilation process

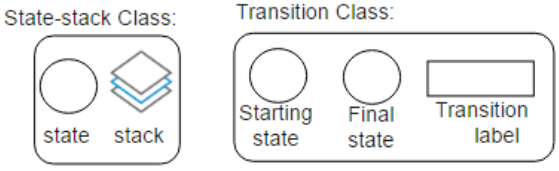

Fig. 4 – The State-stack and Transition classes

4. SIMULATING THE AUTOMATON

For performing the simulation we used two lists of State-stack objects. In the first list, the application looks for current states and when it finds the next states with their stacks, the objects are added to the second list. These lists are shown in Figure 5.

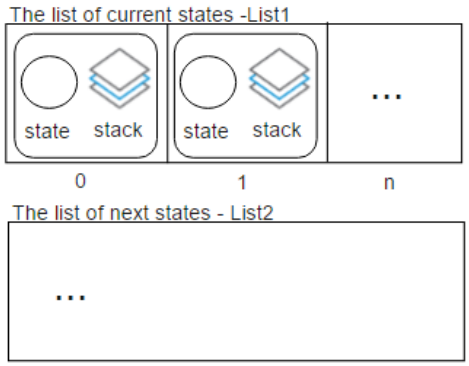

Fig. 5 – The two lists used for simulation

In Figure 6 we present the flow chart for the simulation process. If at some point during the simulation the list of current states (and their stacks) is empty, the input of the automaton is rejected. Otherwise, the simulation continues until all the symbols in the input are consumed and then the acceptance condition is checked in order to generate an accept/reject response.

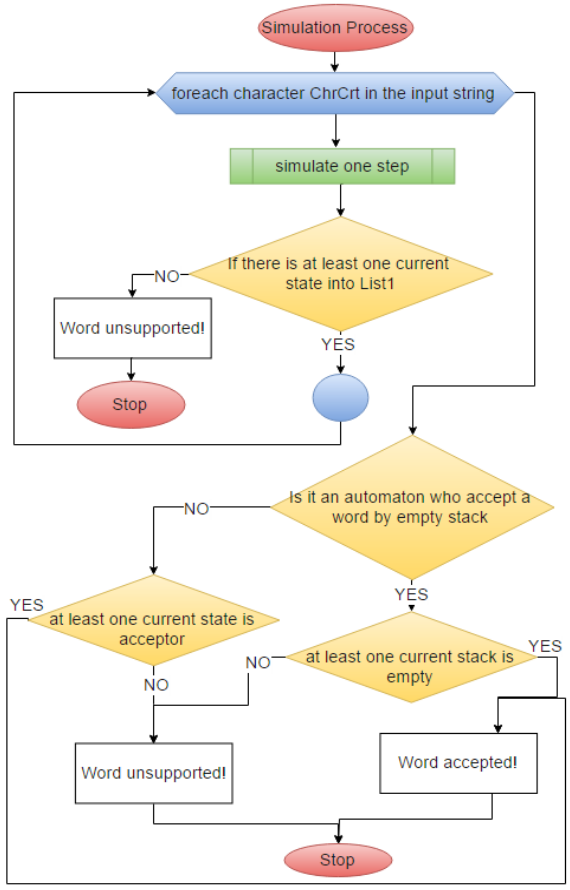

Fig. 6 – The flow chart of the simuation process

A single step in the simulation process is shown in Figure 7. More precisely, in this step the list of the new state-stack pairs is computed starting from the list of current state-stack pairs by following all the available epsilon-transitions and then by following the transitions corresponding to the current symbol in the input.

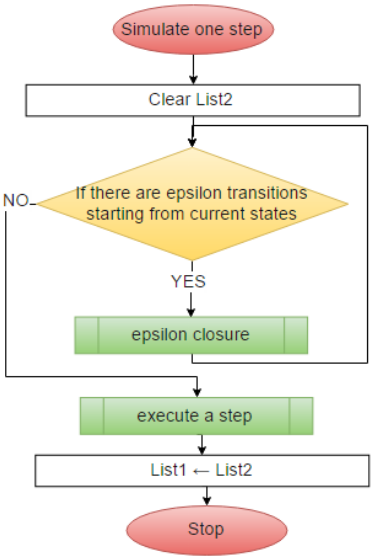

Fig. 7 - One step in the simulation

The epsilon closure is performed as sketched in Figure 8.

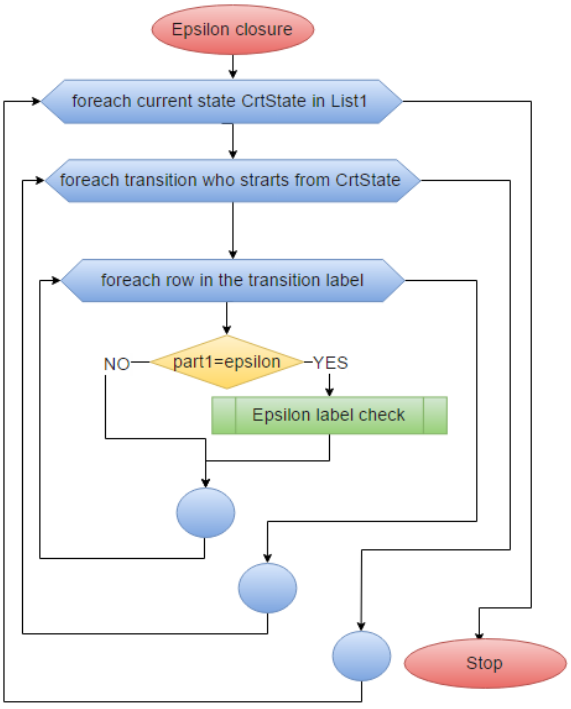

Fig. 8 – The epsilon closure

After the epsilon closure, the non-epsilon transitions are executed. This function is represented by the name “execute a step” and its functionality can be seen in Figure 9.

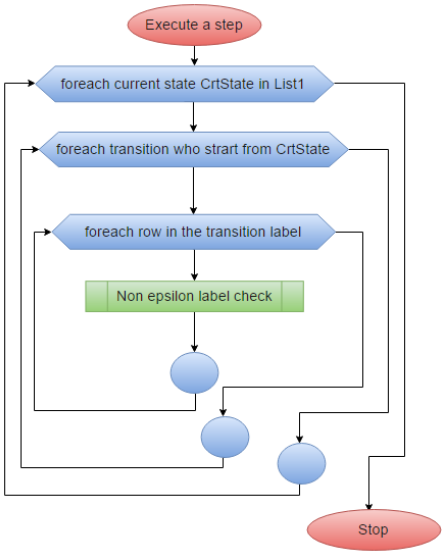

Fig. 9 – The "Execute a step" function

The “epsilon label check” function used by the epsilon closure is shown in Figure 10.

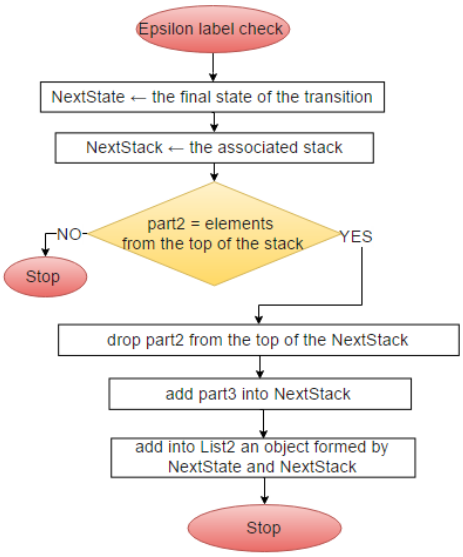

Fig. 10 – The "Epsilon label check" function

Finally, the "non epsilon label check" function is shown in Figure 11.

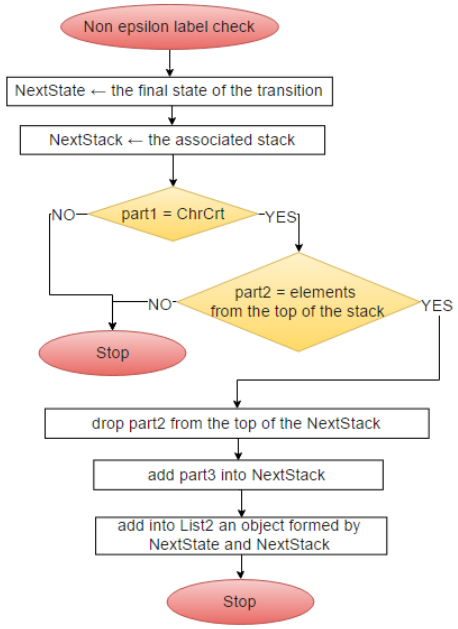

Fig. 11 – The "Non-epsilon label check" function

5. CONCLUSION

Simulating a nondeterministic pushdown automaton is a challenging programming task, as at each time step during the simulation the automaton can be simultaneously in several states, each state having its own stack contents. We have presented in this paper a solution to this challenge by showing the inner workings of a functional simulator for NPDAs.

6. REFERENCES

[1]. Susan H. Rodger, Thomas W. Finley, “*JFLAP – An Interactive Formal Languages and Automata Package*”, Jones&Bartlett Publishers, Sudbury, MA, 2006.

[2]. Mohamed Hamada, “*Pushdown Automata Simulator*” in M. Kuo et al. (Eds.): *Learning by Playing: Game-based Education System Design and Development*, Springer, Berlin, Heidelberg, 2009.

[3]. Felix Erlacher, “*Pushdown Automata Simulator*”, Bachelor Thesis, Universitat Innsbruck, Institut fur Informatik, 2009.

[4]. Kyle Dickerson, website: http://automatonsimulation.com

[5]. Carl Burch, website: http://www.cburch.com/proj/autosim