Enhanced pushdown automaton for recognizing multi-syntx programming languages

A S Kuznetsov¹, R Y Tsarev¹, T N Yamskikh¹, A N Knyazkov¹, K Y Zhigalov²,³, Adebanji Ogundade⁴

¹ Siberian Federal University, 79, Svobodny pr., Krasnoyarsk, 660041, Russia
² V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65, Profsyurnaya street, Moscow, 117997, Russia
³ Moscow Technological Institute, Block 2, 8, Kedrova, Moscow, 117292, Russia
⁴ Campbellsville University, 4420 Dixie HWY Suite 230, Louisville, KY 40216, USA

E-mail: tsarev.sfu@mail.ru

Abstract. Software used to digitize manufacturing operations should provide high degree of reliability, since its failure may cause downtime or disruption of the production process and, as a result, financial losses. One of the approaches to improve software reliability is N-version programming where data processing is performed by different functionally equivalent software components which source code can be written in various programming languages with a number of distinguishing syntactic properties. This paper presents an abstract device for recognizing multi-syntx programming languages. The synthesized device can be used for the development of language translators for N-version software systems.

1. Introduction

To create fault-tolerant software systems, there are many approaches that prove their efficiency for a large variety of practical tasks for both business and manufacturing [1]. The methodology of N-version programming is one of them [2]. Its distinguishing feature is the development of several versions of the same functional module or a program component [3]. In general, this methodology is an addition to applicable methods of designing fault-tolerant systems such as software testing and verification [4].

The software methodology in question is based on the available variety of input data, and it can be integrated in the following elements of the software development process:

- programming languages;
- algorithms for solving a problem;
- programming tools;
- testing methods.

The authors of some papers dedicated to the N-version approach note that in the process of software development it is important to use different programming languages to design versions [2, 5, 6] of software modules due to the fact that implementation of different versions of a module in different languages reduces the likelihood of the same type of errors related to the language.

The process of creating N-version software systems involves their architectural division into components and modules that are created independently of each other. However, it is often necessary to encode these modules in different languages within the same source code, such as when creating
dynamic HTML pages with the inclusion of code in JavaScript and / or VB Script [7]. The traditional process of developing N-version systems does not allow this to be done in a simple way. We propose to circumvent this limitation by using multi-syntax programming languages and the programming tool MuYacc for constructing translators of this class of programming languages [8].

Thus, within one source text of a program, languages with different properties from the point of view of syntax and, as a result, generated by different classes of grammars can be used simultaneously. In this regard, such languages can be referred to the class of multi-syntax languages.

In this case, the syntax of each of the constituent languages is described by context-free grammar. Special characters are extracted from the leader language, signaling the “main” parser to transfer control to the corresponding “auxiliary” parser. The number of lexical and semantic parsers can be equal to the number of languages used, and their interaction with each other is carried out according to one of the traditional schemes.

2. Enhanced pushdown automaton

The MuYacc system is based on the multi-syntax language recognizing device in the form of an enhanced pushdown automaton. In addition, each auxiliary parser is based on a standard pushdown automata model, which is commonly described [9] by the seven elements:

\[ P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), \]

where \( Q \) – the set of automaton states; \( \Sigma \) – the alphabet of input symbols, \( \Gamma \) – the alphabet of pushdown symbols; \( \delta \) – a graphically and tabular represented transition function; \( q_0 \) – the initial state of the automaton; \( F \) is the set of final states of the automaton; \( P, Z_0 \) – the symbol for the bottom of the pushdown store.

The \( \delta \) function with three arguments (\( q \) – the current state of the automaton, \( a \) – the current input character or the empty string \( \varepsilon \), and \( X \) – the top character of the pushdown store) as a result generates a pair of elements (\( p, \gamma \)), where \( p \) – the new state of the automaton; \( \gamma \) – a string of pushdown store symbols, replacing \( X \) at the top of the pushdown store. If \( \gamma = \varepsilon \), then the symbol is removed from the pushdown store. If \( \gamma = X \), then the content of the pushdown store does not change. For example, if \( \gamma = YZ \) then \( X \) is replaced by \( Z \), then \( Y \) appears on the pushdown store.

For the enhanced pushdown automaton \( P_{mul} \) this set is enlarged by introducing three additional elements. Thus, the required abstract device is described as follows:

\[ P_{mul} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F, M, S, \phi), \]

where the first seven elements have the same meaning as in the standard pushdown automata model, \( M \) – a set of auxiliary automata, \( S \) – special states corresponding to switching to an auxiliary automaton from the set \( M \), \( \phi \) – a switch function of the corresponding auxiliary automaton from the set \( M \).

The set \( M = \emptyset \) or \( M = \bigcup_{j=1}^{N} M_j \), where \( M_j, j = 1, N \) – the corresponding pushdown automaton from the size of the set \( N \). If the set \( M \) is empty, then this is a characteristic of a standard pushdown automaton. This should not be treated as an error in case of software implementation.

Similarly, the set \( S \subseteq Q \) can also be empty (\( S = \emptyset \)). In general case, \( S = \bigcup_{j=1}^{K} S_j \), where \( S_j, j = 1, K \) – a special state from the corresponding size of the set \( K \). The number of special states can be greater than or equal to the number of auxiliary automata \( (K \geq N) \), due to the possibility of switching from one special state to the same automaton.

The automaton is considered erroneously specified in the following cases:

a) The number of special states is less than the number of auxiliary automata \( (K < N) \), since it is worthless to insert in the device such a resource-intensive element not getting a single chance of switching to it. However, if desired, this limitation can be removed, for example, by reserving one or more auxiliary automata for further use.
b) If \( M = \emptyset \) and \( S \neq \emptyset \) there is no need to select special states.

c) If \( M \neq \emptyset \) and \( S = \emptyset \) for the same reasons as in paragraph a) of this list.

Assuming further that the enhanced pushdown automaton is not specified erroneously, we describe the switching function \( \varphi \), which can be graphically or tabular represented, similarly of the transition function \( \delta \).

To set such function, it is necessary to distinguish between two cases: when there are auxiliary automata in enhanced pushdown automata and without them. In the first case, there should be indicated the number (non-negative integer \( J_M \)) of the auxiliary pushdown automata from the set \( M \) and the number of the special state from the set \( S \) (\( J_S \)). As a result, we obtain the function \( \varphi_{\text{notempty}} \) which takes five arguments and produces an output as a set of three elements:

\[
\varphi_{\text{notempty}}(q,a,X,J_M,J_S) : \{p,\gamma,M_{J_u}\},
\]

where \( p \) – a new state of the automaton; \( \gamma \) is a string of pushdown store symbols, \( M_{J_u} \) - automaton with the number \( J_M \) from the set \( M \).

If you look at it from a constructive point of view, it is the moment when the transition to the auxiliary pushdown automaton takes place in the form of an appropriate procedure call or transition in a table depending on the implementation type, which in turn depends on the required efficiency of the compiler.

Now let us suppose that \( M = \emptyset \). In other words, there are no auxiliary automata in our abstract device. In this case, it is necessary for the switching function \( \varphi \) to behave in the same way as the transition function \( \delta \) of a standard automaton. We obtain the function \( \varphi_{\text{empty}} \), which, on closer examination differs from the transition function only in its name:

\[
\varphi_{\text{empty}}(q,a,X) : \{p,\gamma\}.
\]

In a short form, the switching function \( \varphi \) is defined as follows:

\[
\varphi(q,a,X,J_M,J_S) = \begin{cases} 
\varphi_{\text{notempty}}(q,a,X,J_M,J_S) : \{p,\gamma,M_{J_u}\}, & \text{if } M \neq \emptyset \text{ and } S \neq \emptyset \text{ and } |M| \leq |S|, \\
\varphi_{\text{empty}}(q,a,X) : \{p,\gamma\}, & \text{if } M = \emptyset \text{ and } S = \emptyset. 
\end{cases}
\]

Now we give a definition of the enhanced pushdown automaton taking into account the equations (2) and (3):

\[
P_{\text{mul}} = \begin{cases} 
(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F,M,S,\varphi_{\text{notempty}}), & \text{if } M \neq \emptyset \text{ and } S \neq \emptyset \text{ and } |M| \leq |S|, \\
(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F,M,S,\varphi_{\text{empty}}), & \text{if } M = \emptyset \text{ and } S = \emptyset, \\
\text{Otherwise error occurs.} 
\end{cases}
\]

Taking into account (4) the following formal definition of an enhanced pushdown automaton, which differs from (1) only by an explicit indication of an error in the description will be equivalent for (5):

\[
P_{\text{mul}} = \begin{cases} 
(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F,M,S,\varphi), & \text{if } |M| \leq |S|, \\
\text{Otherwise error occurs.} 
\end{cases}
\]

3. The operation of enhanced pushdown automaton

The operation of a standard automaton as well as a degenerate enhanced pushdown automaton is described by changing configuration [10], where the three elements which involve the state q, the
remaining part of the input string of characters constituting the program $\omega$, and the contents of the pushdown store $\gamma$ are considered as the configuration of a pushdown automaton. Thus, \( ID = (q, \varepsilon, \gamma) \).

If \( P \) is a pushdown automaton, \( \delta(q, a, X) \) containing \( (p, \alpha) \), then \( P \) defines such a relation for all chains $\omega$ from $\Sigma^*$ and $\beta$ from $\Gamma^*$, that is \( (q, a\alpha, X\beta) \rightarrow P (p, \varepsilon, \alpha\beta) \).

In a way similar to these concepts, we define the configuration or the instant description of $ID_{mul}$ of our enhanced pushdown automaton $P_{mul}$. Since an auxiliary pushdown automaton can be activated at some point in time in the selected state, in addition to the current state, the remaining part of the input and the contents of the pushdown store, the enhanced pushdown automaton is also described through the configurations of all its pushdown automata. The latter is the set $ID_{aux}$:

\[
ID_{aux} = \bigcup_{1}^{N} ID_j,
\]

where $N$ – the number of auxiliary automata. Thus, we get an instant description of the enhanced pushdown automaton:

\[
ID_{mul} = \begin{cases} (q, \varepsilon, \gamma, \emptyset), & \text{if } P_{mul} = P, \\ (q, \varepsilon, \gamma, ID_{aux}), & \text{otherwise.} \end{cases}
\]

We also define the binary transition relation \( P_{mul} \) for the enhanced pushdown automaton. If $P_{mul}$ – the enhanced pushdown automaton described, \( \phi(q, a, X, J_M, J_S) \) containing \( (p, \alpha, M_{j_u}) \), then \( P_{mul} \) defines such relation for all strings $\omega$ from $\Sigma^*$ and $\beta$ from $\Gamma^*$ that is \( (q, a\alpha, X\beta, M_{j_u}) \rightarrow P_{mul} (p, \varepsilon, \alpha\beta, M_{j_u}) \).

We also use the symbol \( P_{mul}^* \) to represent zero or more transitions of the enhanced pushdown automaton. Thus, we have the following inductive definition.

Basis: \( I \rightarrow P_{mul}^* I \) for any instant description of $I$.

Induction: \( I \rightarrow P_{mul}^* J \), if there is some instant description of $K$, that satisfies the conditions $I \rightarrow P_{mul} K$ and $K \rightarrow P_{mul}^* J$.

Thus, \( I \rightarrow P_{mul}^* J \), if there is such sequence of instant descriptions $K_1, K_2, \ldots, K_n$, for which $I = K_1, J = K_n$, and $K_i \rightarrow P_{mul} K_{i+1}$ for all $I = 1, 2, \ldots, n - 1$.

4. Languages accepted by an enhanced pushdown automaton

Having obtained such description of an abstract device and the mechanism of its operation, we can now define the concept of a multi-syntactic language, recognized by the enhanced pushdown automaton.

Thus, an enhanced pushdown automaton accepts a multi-syntactic language by the final state, if it starts its work in the initial configuration, having received the input string of characters that make up the program, reads it completely, transferring control to the corresponding auxiliary pushdown automaton (if there is any), and reaches one of its final configurations, if the auxiliary automaton reach their final configurations or empty their pushdown stores.

As the result, if $P_{mul}$ – an enhanced pushdown automaton, described by the equation (6), then the language $L(P_{mul})$, accepted by the enhanced pushdown automaton by the final state is the equation:

\[
L(P_{mul}) = \{ \alpha(\sigma_0, \alpha, Z_0, ID_{mul}) \rightarrow P_{mul}^* (q, \varepsilon, \alpha, ID_{mul}) \}.
\]

The proof.

1. First, let us suppose that an enhanced pushdown automaton does not contain auxiliary automata, then, as it was mentioned above, it does not differ from the standard pushdown automaton. Consequently, the proof of its repeatability to a finite configuration does not differ from the proof given, for example, in [11].

2. Further we shall assume that our enhanced pushdown automaton contains only one auxiliary pushdown automaton. Since, by definition, it gets the control function in one of the specially designated
states of the enhanced pushdown automaton, and after returning control function to the latter, the transition to the next “normal” state is performed in accordance with the transition function and the auxiliary automaton doesn’t have any other impact on our abstract device; it can be removed from the enhanced pushdown automaton. Then the proof of repeatability can be narrowed down to (1).

3. Now let us suppose that our enhanced pushdown automaton has two auxiliary pushdown automata. Since, as it was mentioned above, they do not affect each other, we can remove them one by one from the enhanced pushdown automaton. Then the proof of repeatability can be narrowed down to (2).

4. In the same way we can prove the finiteness of the algorithm of an enhanced pushdown automaton operation with any number of pushdown automata that is more than one.

The multi-syntax language is accepted by an enhanced pushdown automaton by the empty stack, if it starts its work in the initial configuration, having received the input string of characters that make up the program, reads it completely, and finally finishes its work if there is only a bottom marker in the pushdown store:

\[ L(P_{mul}) = \left\{ \omega \mid \left( q_0, Z_0, \epsilon, ID_{mul} \right) \Rightarrow^{*} \left( q, \epsilon, \epsilon, ID_{mul} \right) \right\}. \]

Limitations for auxiliary automata are the same as in equation (7). In the sixth chapter of the book [11], we find the proof of the equivalence of languages accepted by the empty stack and by the final state. Consequently, constructing an enhanced pushdown automaton that accepts a language by the empty stack, equivalent to the same device that accepts a language by the final state, we can use the proof of equation (7).

5. Table specification for parsing multi-syntax programming languages

Thus, the proposed enhanced pushdown automaton can be used as the basis of the program for parsing multi-syntax programming languages. However, at each step, it may require several pushdown stores of arbitrary size at each moment in time, which is not always acceptable even for the resources of modern computers. For a number of reasons, including the one indicated, the programming language translators, as a rule, are built on the basis of table-driven syntax analysis algorithms [10]. These algorithms are adequately covered in the special literature, and the authors of this article decided to modify one of the table layout algorithms for syntax analysis to use it later for the creation of the MuYacc system.

It is known that LALR (1) (Look Ahead Left Recursive) covers the most of the syntax aspects of programming languages used in practice. A modification of this method allowed us to obtain the algorithm mLALR (1), which inherited the necessity to create tables, which rows correspond to the states of the parser, and columns to the terminal and non-terminal symbols of the context-free grammar describing each language-component of the multi-syntax language. In this case, the classical LALR (1) algorithm with the elements that can take the values of four types [10] is used for parsing auxiliary languages. The parser for the leader language uses tables with the elements of five types, the first four of which are inherited from the LALR (1) algorithm:

1. SHIFT elements. These elements are of the form \( S_i \) and mean: place the symbol corresponding to the column into the pushdown store; put the state \( k \) into the pushdown store and transfer to the state \( k \); if the considered symbol is terminal, then accept it. If this shift involves running of an auxiliary language parser, then this action is performed.

2. REDUCE elements. They have the form \( R_m \) and mean: perform a reduction using the rule \( m \); remove \( N \) characters and \( N \) states from the pushdown store \((N \) is the number of characters on the right hand side of rule \( m \)); transfer to the state at the top of the pushdown store. The non-terminal from the left hand side of the output \( m \) is considered as the next input symbol.

3. ERROR elements. These are empty table cells corresponding to syntax errors.

4. ACCEPTING elements, having the form \( Accept \). They complete the parsing process (input string is accepted). If an auxiliary language is being parsed, then return a successful value to the leading language parsing procedure.

5. SHIFT-AND-EXECUTE elements. These elements have the form \( E_k \) and mean: place the symbol corresponding to the column into the pushdown store; put the state \( k \) into the pushdown store; find and
run the parser of the corresponding auxiliary language; if this parser ends in an error state, then complete the parsing of the leading language with an error; otherwise, transfer to the state \( k \). Since the symbol under consideration is terminal, then accept it.

The leader language parser starts its operation from the initial state 0 and, transferring from one state to another, it can end its work either in the Accept state or in the error state.

6. Conclusion

On the basis of the enhanced pushdown automaton described above we implemented a compiler-compiler, designed to automate the process of generating multi-syntax language translators. The compiler-compiler was practically used to create an experimental compiler of the C programming language with the possibility of including code in the assembler language of the x86 architecture in two syntax notations: Intel and AT&T, commonly used in manufacturing industries. The beta testing of the compiler demonstrated almost double reduction in the number of errors when building the second version of dual-variant modules with a total code size of about 8 KLOC, regardless of their design order.

References

[1] Patel K and Hierons R M 2018 A mapping study on testing non-testable systems Software Qual. J. 26(4) 1373-1413
[2] Chen L and Avizienis A 1978 N-version programming: a fault-tolerance approach to reliability of software operation Proceedings of the Annual International Conference on Fault-Tolerant Computing 3-9
[3] Franz M 2018 Making multivariant programming practical and inexpensive IEEE Secur. Priv. 16 (3) 90-94
[4] Ionescu T B and Scheuermann W 2016 Improving the reliability of decision-support systems for nuclear emergency management by leveraging software design diversity Journal of Computing and Information Technology 24 (1) 45-63.
[5] Gruzenkin D V, Chernigovskiy A S and Tsarev R Y 2018 N-version software module requirements to grant the software execution fault-tolerance Advances in Intelligent Systems and Computing 661 293-303
[6] Sommerville I 2006 Software Engineering, 10th ed. (London: Pearson) p 816
[7] Verdú J, Costa J J and Pajuelo A 2016 Dynamic web worker pool management for highly parallel javascript web applications Concurrency Computation 28(13) 3525-3539
[8] Kuznetsov A S, Kovalev I V and Veretennikov E A 2007 Automaton of the process of multi-syntax programming language compilers generation Vestnik of SibSAU 3 (16) 73-75
[9] Atig M F, Bollig B and Habermehl P 2008 Emptiness of multi-pushdown automata is 2ETIME-complete Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5257 LNCS 121-133
[10] Aho A V, Lam M S, Sethi R and Ullman J D Compilers: principles, techniques, and tools, 2nd ed (Boston: Addison Wesley) p 1040
[11] Hopcroft J E, Motwani R and Ullman J D 2006 Introduction to automata theory, languages, and computation, 3rd ed (London: Pearson) p 560