Compiler-compiler of multi syntax programming languages for creating N-version software

A S Kuznetsov¹, R Y Tsarev¹, T N Yamskikh¹, A N Knyazkov¹, A N Pupkov¹, F A A Laleye²

¹ Siberian Federal University, 79, Svobodny pr., Krasnoyarsk, 660041, Russia
² Atomic Energy and Alternative Energies Commission, CEA Saclay / Siège, Gif-sur-Yvette Cedex, 91191, France

E-mail: askuznetsov@sfu-kras.ru

Abstract. Ensuring the reliability of the software has become increasingly complex and challenging. The most important problem is providing failure-free software operation to prevent financial losses or machinery breakdown. It can be solved with N-version programming which requires increased costs due to the use of redundant software components. In this paradigm, redundancy refers to diversity of software components. This paper discusses the implementation of software components in various languages with different syntax. We propose an efficient and time-reducing approach to the development of a multi-syntax N-version software solutions, as well as reduce the likelihood of their occurrence due to the use of an automated tool, which is crucial for its implementation in an industrial setting.

1. Introduction

Currently, there are several approaches to creating fault-tolerant software systems. The methodology of N-version software is one of the most obvious and promising of them [1, 2]. Suppose that a program or a software system implies that several modules that perform the same functions are launched during its execution [3]. These modules are implemented as tasks (processes or threads) with isolated or non-isolated address spaces [4]. The environment of their implementation can be known in advance, and for various reasons, software implementation of these modules is carried out in assembly languages, which provide additional advantage for program verification in comparison with other languages [5]. However, modules that implement control functions or perform interactive tasks should be encoded in high-level languages. For some of them programming systems offer the extension of such languages with the help of so-called inline assembler. A slightly more general case is the use of assembler fragments for several microprocessor systems. And finally, in the most general case, it is possible to use simultaneously several high-level languages in the same source code (for example, one module runs on a Java Virtual Machine and another on a Lisp Virtual Machine). In any case, a single language is assumed to be used as a leading one.

Thus, within the framework of a single source text, languages with different properties can be used simultaneously from the point of view of syntax and, as a result, generated by different classes of grammars. In this regard, such languages can be attributed to the class of multi syntax languages. It is
very important that the parsing such languages in general may require the use of a 0-type grammar according to Chomsky hierarchy [6].

It is “desirable” to define the syntax of each constituent language with context-free grammar. Special characters which signal the “main” parser that it must activate the corresponding “secondary” analyzer are singled out from the entire alphabet of the leading language. The number of lexical and semantic analyzers 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. At the same time, the number of generated syntax trees may differ from the number of constituent languages, that is, they may be ambiguous. This, as well as possible actions to synthesize the target code (for all intents and purposes) of the program, should generate the compiler of the required multi syntax language, which can be used to create N-version software.

2. Method

Formally, the syntax of the multi syntax language \( L \) can be defined as follows using the list of grammars \( G \), where \( N \) is the amount of “auxiliary language” grammar:

\[
G = (G_{\text{lead}}, G_1, \ldots, G_{N})
\]

where \( G_{\text{lead}} \) is the leading language grammar, the other grammars \( G_i \) are auxiliary, which have all intrinsic properties of context-free grammars [6]. At the same time, the production of the \( G_{\text{lead}} \) grammar looks like context-free, but with two big differences, that is, it is defined by six elements:

\[
G_{\text{lead}} = (T, N, P, S, V_{\text{switch}}, P_{\text{switch}})
\]

where \( T, N, P, S \) have the same meaning as in [6], \( V_{\text{switch}} \) is the set of special tokens, which is a subset of the set of terminal symbols \( T \).

\( P_{\text{switch}} \) is a dedicated set of products for switching between grammars, which is a subset of the entire set of products. In fact, they are pairs of the form \( \alpha \rightarrow A_i\beta \), where \( \alpha \) is the left hand side of the product; \( A_i \) is a special terminal \( (A_i \in V_{\text{switch}}) \), that causes a change of grammar from \( G_{\text{lead}} \) to \( G_i \); \( \beta \) is the tail of the right hand side of the product, which contains a dedicated nonterminal that names the \( G_i \) grammar product group.

Naturally, the entire set of parsers can be viewed as a single module, but in order to automate their generation, it is more convenient to consider them as separate modules (see Figure 1).

![Figure 1. The scheme of parsers interaction.](image)

In order to automate the process of multi syntax language compilers generation, it is proposed to develop a so-called “compiler-compiler”. To do this, it is wiser to use any well-proven tool, modify it in accordance with the task, debug and put it into operation. As mentioned in the paper [7], the most appropriate tools for this are related programs bison and BYacc. They build bottom-up parsers that work with the LALR (1) algorithm, covering the most commonly used programming languages. The same
paper [7] describes the development of the program with the code name MuYacc, where the regular expression mechanism is used for automation lexers building, and the production system is implemented for automation of parsers building. All that is left to do is to take into account the following limitations and recommendations, which are closely interrelated:

- “Auxiliary” parsers do not interact with each other horizontally, in particular, in order to avoid the loops of parse calls, as in the example shown in Figure 2. This means that the occurrence of such loops is checked before processing the input specifications.
- The leading parser and neither of the auxiliary parsers should be identical, that is, belong to the same language. This means that the equivalence of a set of lexemes, as well as production systems, is checked by comparing the resulting syntax trees.
- Actions to check semantic agreements and actions of the synthesis stage are performed in a programming language or described in XML format.
- One of the industrial programming languages is used as a leading one, for example, C++.

![Figure 2. The loops of the parsers calls and mismatch between the leading language and the auxiliary language.](image)

3. The input specification of the development tool of the multi syntax language compiler

The project with the code name MuYacc is based on the BYacc input specification, which generally consists of three parts, separated by characters `%%`:

- declarations,
  `%%`
- rules (expressions),
  `%%`
- user defined functions.

The declarations section lists the terminal characters of the implemented programming language alphabet, preceded by the keyword `%token`, and provides some rules for resolving the ambiguity of the language grammar. The rules of this grammar are listed in the second part of specification, the lexer is written in the language of regular expressions in the first part of the input specification. Other procedures are defined in the third part.

In the process of creating an automated development tool for programming language translators there exist two possible ways to implement lexers using regular expressions. They can be placed:

- in the declarations section,
- in the user defined functions section.

The first option – the implementation of regular expressions in the declarations section, is more convenient, as in this case the specification is more demonstrative.

To do this, the keyword `%regexp` is used in the input specification. It defines a specific regular expression (its name, description, actions), even if it is not used as a terminal character in the alphabet of the language being implemented and is not contained in its grammar products. To describe a terminal
character, it is enough to create a regular expression with the same name. Below is a fragment with the description of terminal characters and simplified grammar products for arithmetic expressions:

```c
%regexp letter [a-z]
%regexp digit [0-9]
%regexp IDENT letter(letter|digit)* %action%  {strcpy($$,yytext);}
%regexp NUM digit+ %action% {$$=atoi(yytext);}
%token <str> IDENT
%token <num> NUM
%
expr  :   expr '+' expr | '(' expr ')' | IDENT | NUM;
%%
```

The use of the construct \%regexp ensures the use of:

1. the existing bison/Byacc specifications with little or no changes (to do this, it is enough to add fields for tokens description);
2. the existing software tools (based on the given specification format, it is quite easy to generate specifications for flex and Bison, and then generate the source texts for the translator parts).

When generating specifications for Flex and Bison there occurs some problem with the development of specification for Flex. To get it, it is necessary to process all constructs \%regexp and \%token. As for Bison specification, it is enough to copy everything that is not related to \%regexp.

The ability to describe multi syntax is implemented by four new keywords \%tokengroup, \%endtokengroup, \%syntaxgroup, \%endsyntaxgroup, placed, respectively, in the first and second parts of the input specification. They possess all the properties described below.

The keywords \%tokengroup and \%endtokengroup play the role of operator brackets that enclose groups of regular expressions. Similarly, for groups of products the words \%syntaxgroup and \%endsyntaxgroup act as the brackets. Both “opening” brackets name the beginning of the group and give names to the procedures of lexical and syntax analysis, respectively, for further use (the property name). The name of the “special token” (the property kword), as well as the names of the corresponding lexeme groups and regular expressions (the property tokengroup) are also indicated for product groups. Moreover, it is possible to import groups of tokens and products from external files. Herewith, their existence is checked. Group nesting is prohibited for the reasons described above.

To designate the “special token” in the input specification of the leading parser, we can use one more keyword \%switchtoken. This token must appear twice in the specification: as a regular token and as a “special” one. Examples of possible specifications are given below (actions for code generation are not provided).

1) without import from the external file

```c
%tokengroup name="arith_tokens"
%regexp letter [a-z]
%regexp digit [0-9]
%regexp IDENT letter(letter|digit)* %action%  {strcpy($$,yytext);}
%regexp NUM digit+ %action% {$$=atoi(yytext);}
%token <str> IDENT
%token <num> NUM
%endtokengroup
%switchtoken ARITH
%
analyzer    :
   | analyzer  \n   | expressions analyzer
;
expressions : ARITH '{' arith_expressions '}'
;  
```
The improved leading language parser resulting from the program operation is suspended temporarily in the process of "special lexeme" recognition; there thus follows switching to the actions of the corresponding auxiliary parser, and then the return to the top of the parsers hierarchy is performed. Other than that, the leading parser operates in the same way as an ordinary one.

4. Conclusion
The approach described above should significantly reduce the time needed for compiler development, ease the process of finding errors in parsing programs based on existing flex and Bison specifications, and also reduce the likelihood of their occurrence due to the use of an automated tool. This should confirm the advantages of implementing the developed program for compiling multi syntax programming languages, especially for N-version software.

References
[1] 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
[2] Franz M 2018 Making Multivariant Programming Practical and Inexpensive IEEE Security and Privacy 16 (3) 90-94
[3] Boiten E A 2016 Diversity and adjudication Journal of Logical and Algebraic Methods in Programming 85 (5) 860-866
[4] 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
[5] Hu T, Bertolotti I C and Navet N 2018 Towards seamless integration of N-version programming in model-based design Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 1-8
[6] Parkes A P 2002 Introduction to Languages, Machines and Logic: Computable Languages (London: Springer) p 351
[7] Kuznetsov A S 2006 Instrumental development tools for translators of multi syntax programming languages Bulletin of the University Complex 7 (21) 227-230