Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1491–1498

pLERO: Language for Grammar Refactoring Patterns

Ján Kollár, Ivan Halupka, Sergej Chodarev and Emília Pietriková
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic
E-mail: {jan.kollar, ivan.halupka, sergej.chodarev, emilia.pietrikova}@tuke.sk

***Abstract*—Grammar-dependent software development and grammarware engineering have recently received considerable attention. As a significant cornerstone of grammarware engineering, grammar refactoring is, nevertheless, still weakly understood and practiced. In this paper, we address this issue by introducing pLERO, formal specification language for preserving knowledge of grammar engineers, complementing mARTINICA, the universal approach for automated refactoring of context-free grammars. With respect to other approaches, advantage of mARTINICA lies in refactoring on the basis of user-defined refactoring task, rather than of a fixed objective of the refactoring process. To understand the unified refactoring process, this paper also provides a brief insight into grammar refactoring operators, providing universal refactoring transformations for specific context-free grammars. To preserve knowledge considering refactoring process, we propose formalism based on patterns, seen as well-proven way of knowledge preservation in variety of domains, such as software architectures.**

I. Introduction

AUTOMATED grammar refactoring is the field where two or more equivalent context-free grammars may have different forms. Although two equivalent grammars generate the same language, they do not necessarily share other specific properties measurable by grammar metrics [1]. The form in which a context-free grammar is written may have a strong impact on many aspects of its future application. For instance, it may affect general performance of a parser [2], or it may influence, and in many cases limit, the choice of parser generator [2].

Since there is a close relation between the form in which a grammar is expressed and the purpose for which it is designed, different grammars become domain-specific formalizations if generating the same language. Thus, the ability to transform a grammar to another (equivalent), indeed, becomes the power to shift between domains of possible applications. Even if making each grammar more universal in its application scope, the practical benefits may be easily thwart by the difficulties. The problem is, refactoring is often a non-trivial task and if done manually, it is prone to errors, especially with large grammars. This is an issue, as in general there is no formal way to prove two context-free grammars generate the same language.

We addressed this issue in [3] by proposing mARTINICA, metrics Automated Refactoring Task-driven INcremental syntactIC Algorithm. Its main idea is to apply a sequence of simple transformation operators to a chosen context-free grammar to produce an equivalent grammar with the desired properties. Each refactoring operator transforms arbitrary context-free grammar to an equivalent context-free grammar which may have different form than the original. Properties the grammar should possess are defined by so called objective function. That is, the purpose of mARTINICA is to find a sequence of refactoring operator instances transforming particular context-free grammar to an equivalent with a form satisfying user-defined requirements. Current state of the algorithm development requires grammar production rules to be expressed in the BNF notation as it in general, unlike EBNF, expresses elementary properties, e.g. left/right recursion or iteration.

With respect to diversity of possible requirements on the qualitative properties, refactoring operators provide relatively universal grammar transformations. Although the relative universality of refactoring operators contributes to versatility of the algorithm, it also may lead to high computational complexity and, in specific cases, to inability to fulfill the refactoring task. Within the current research, we propose a solution of these issues based on patterns which, in this context, we consider to be a problem-specific refactoring operators.

In general, we consider a pattern to be a problem-solution pair in given context [4] [5]. Alexander argues each pattern can be understood as an element of reality, and of language [4]. As an element of reality, pattern reflects a relation between specific context, certain system of forces recurring in given context, and certain spatial configuration leading to balance in a given system of forces [4]. As an element of language, pattern reflects an instruction showing how certain spatial configuration can be repeatedly used to balance certain system of forces wherever specific context makes it relevant [4].

As such, patterns are tools for documenting existing, well proven design knowledge, supporting construction of systems with predictable properties and quality attributes [5]. Thence, the role of patterns in the field of grammar refactoring is:

1) To preserve knowledge of language engineers about when and how to refactor context-free grammars, and
2) To support process of grammar refactoring by providing this knowledge.

To incorporate patterns in automated grammar refactoring, we have coined a new term: *grammar refactoring patterns*. Each grammar refactoring pattern describes a way in which a context-free grammar can be transformed preserving generated language, and a specific situation of this to be possible. Description of the situation, in which transformation provided by

978-1-4673-4471-5/$25.00 © 2013, IEEE

a pattern can be applied, defines refactoring problem addressed by the pattern, while grammar transformation defines solution of the refactoring problem.

II. Motivation

Grammarware engineering as an up-and-rising discipline aims at solving grammar development issues, promising an overall rise in grammar quality, and development productivity [6]. Grammar refactoring may occur in many fields, e.g. grammar recovery, evolution and customization [6]. In fact, it is one of five core processes of grammar evolution, alongside the extension, restriction, error correction, and recovery [7]. However, unlike a well-proven practice of program refactoring, grammar refactoring is little understood and practised [6].

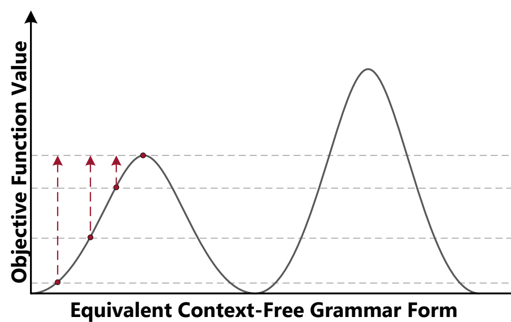

Fig. 1. Previous research approach

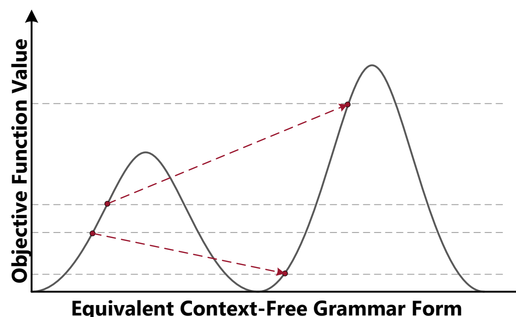

Fig. 2. Current research approach

Fig. 1 and 2 both reflect the objective function value at particular grammar forms. Horizontal axis, indeed, denotes a hypothetic area (not a dimension) of all the equivalent context-free grammar forms, and vertical axis denotes the objective function values of this grammar.

The points marked in the graphs represent forms of a context-free grammar. All the points originate from a single point, corresponding to the initial grammar form and its objective function value.

Our previous approach involved an improvement of the objective function through the application of refactoring operators [3]. Its potential to improve is expressed by the up arrows in Fig. 1. The issue lied in local extrema: If populations reached them, further slide over the function became uncertain.

The algorithm of mARTINICA solved this issue by enabling the populations to regress, but merely in a certain number of steps [8], which is one of the few possible heuristics. Two potential ways of the algorithm are to enable a progress to a certain value or a certain number of steps. However, neither of them is ideal and cannot work universally. Further, the issue lied in a negative impact on the computational complexity as well.

Consequently, the current approach considers *refactoring patterns*. If applied to a grammar, at the corresponding objective function it is possible to skip the local extremus (Fig. 2), what is their primary feature. Certainly, various patterns concern various objective functions. That is, this solution is not universal, however, it is ideal for domain-specific tasks, such as left recursion removal.

Since this approach is not heuristic and it always works, it is considered to be progressive according to the previous one.

Generally, the main idea behind our research lies in grammar modifications according to their objective functions, which is supplemented by the current research dedicated to creation of a tool for grammar modifications according to properties of refactoring operators.

III. Related Work

Unfortunately, it was possible to find very little reported research in the field of automated grammar refactoring. The small amount of the published work is mostly concerned with refactoring context-free grammars achieving some fixed domain-specific objective.

Kraft, Duffy and Malloy developed a semi-automated grammar refactoring approach to replace iterative production rules with left-recursive rules [9]. They present a three-step procedure consisting of grammar metrics computation, metrics analysis to identify candidate nonterminals, and transformation of the candidate nonterminals. The first and third step are fully automated, while the process of identifying nonterminals, to be transformed by replacing iteration with left recursion, is done manually. Since grammar metrics are calculated automatically, this approach is called metrics-guided refactoring. However, the resulting values must be interpreted by human, using them as a basis for making the decisions necessary for resuming the refactoring procedure. The work also provides an exemplary illustration of the grammar refactoring benefits, since left-recursive grammars are more useful for some aspects of the grammar application [10], and are also more useful to human users [11] than iterative grammars.

In the field of compiler design, the procedure of left-recursion removal is a well-known practice. Louden reports an algorithm for automated removal of direct and indirect left recursion [12]. This approach is further extended by Lohmann, Riedewald and Stoy [11], presenting a technique for removing left-recursion in attribute grammars and semantic preservation while executing this procedure.

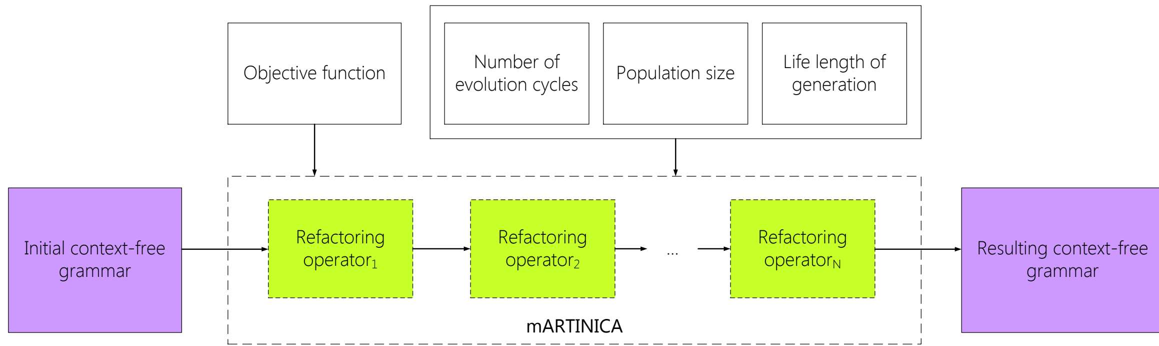

Fig. 3. Black-box view of mARTINICA

Lämmel presented suite of fifteen grammar transformation operators, four considering grammar construction, five considering grammar destruction and six considering grammar refactoring [13]. These operators are in large degree tailored for solving issues of two specific problem domains e.g. grammar adaptation and grammar recovery. Paper [13] also introduced the idea of incremental grammar refactoring through the sequence of simple transformations deriving from application of refactoring operators, however no specific automated refactoring approach, such as mARTINICA [3] was introduced.

Lämmel and Zaytsev introduced suite of four refactoring operators, specifically aimed for tackling refactoring tasks occurring in the process of grammar extraction from multiple diverse sources of information [14].

IV. Background

This section discusses refactoring operators as a basis for understanding grammar refactoring patterns and the core idea of the approach. This section also briefly introduces a method of describing a context-free grammar properties through the formalism of an objective function, used as a specification of the refactoring objective.

A. Refactoring Operators

Formally, grammar refactoring operator is a function taking a context-free grammar $G = (N, T, R, S)$ and using it as a basis for creating new grammar $G' = (N', T', R', S')$ equivalent to G. At this stage of development, the experiments were performed on the basis of eight operators: Unfold, Fold, Remove, Pack, Extend, Reduce, Split and Nop. The first three have been adopted from Lämmel's paper on grammar adaptation [13], while the others are proposed by us [8].

Grammar refactoring patterns are proposed as an addition to the base of refactoring operators. However, in this context, the key difference between refactoring operators and patterns is that the growth in the number of patterns (in the base of operators) does not have significant negative impact on the algorithm complexity, and the opposite is often true. This is caused by their domain-specific orientation and quite narrow scope of refactoring tasks to which individual patterns are applicable.

B. Objective Function

We adopt a modified understanding and notation of objective functions from mathematical optimization. An objective function describes properties of a context-free grammar to be achieved by refactoring. However, it does not describe the way in which the this should be performed, and the condition in which desired context-free grammar properties are achieved.

In our view, the objective function consists of two parts: *objective* and *state function*. Our refactoring algorithm works with only two kinds of objectives, which are minimization and maximization of a state function. We define a state function as an arithmetic expression whose only variables are the grammar metrics [1] calculable for any context-free grammar. As such, a state function is a tool for qualitative comparison of two or more equivalent context-free grammars.

Exemplary objective function prescribing minimization objective under state function consisted of count of nonterminals (var) and count of production rules ($prod$) is (1).

$$f(G) = minimize\ 2 * var + prod \tag{1}$$

C. mARTINICA: Refactoring Algorithm

The main idea behind mARTINICA (Fig. 3) lies in applying a sequence of grammar refactoring operators to a context-free grammar, to produce an equivalent grammar with a lower value of the objective function if the objective is minimization, or a higher if the objective is maximization. On the other hand, pLERO allows specifying one operator of such a sequence.

Since mARTINICA is an evolutionary algorithm, it also requires other input parameters, in addition to the *initial grammar* and the *objective function*, in order to be executed. It requires three other input parameters: *number of evolution cycles*, *population size* and *length of a generation life*. The first two are typical for algorithms of a similar type, while the third parameter is our own.

As shown in Fig. 4, presenting a white-box view, the algorithm starts with creation of an initial population of grammars. Each population member is then created in the basis of the initial grammar, transformed by semi-random sequence of refactoring precesses. After the initial phase, the algorithm iterates for count of evolution cycles through:

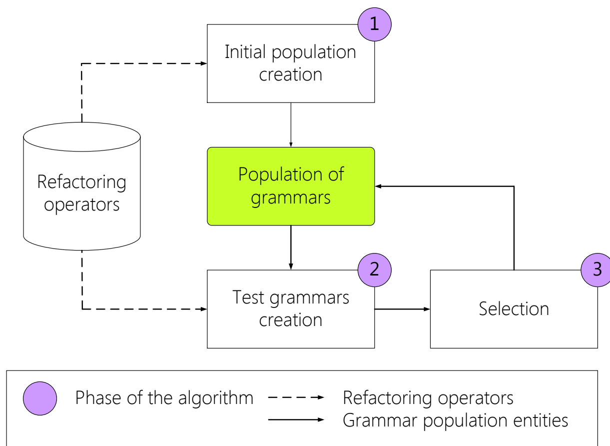

Fig. 4. White-box view of mARTINICA

1) *Test grammars creation* in which candidate population members are created. For each grammar in each generation three test grammars are created:
   a) Self-test grammar that attempts to redouble transformation that led to improvement in value of objective function in past generations of this population member.
   b) Foreign-test grammar that attempts to incorporate transformation that led to progress in value of objective function in past generations of some other population member.
   c) Random-test grammar that attempts to transform grammar towards optimization of value of objective function on the basis of random sequence of refactoring operators.
2) *Selection* in which population members are substituted by candidates with best value of objective function.

The resulting grammar reflects a population member of the last generation with the best value of the objective function.

Detailed description concerning proposal and implementation of mARTINICA algorithm along with some experimental results can be found in [3] [8].

V. Grammar Refactoring Patterns

In our view, each grammar refactoring pattern provides an equivalent transformation to context-free grammars. In this sense, the concept of grammar refactoring patterns is closely related to the concept of refactoring operators. However, there are several key differences between grammar refactoring patterns and refactoring operators.

First of all, refactoring operators provide problem-independent transformations, while grammar refactoring patterns provide problem-specific transformations. This means refactoring operators provide general transformations, with usage not bound by any specific class of refactoring tasks, while grammar refactoring patterns provide domain-specific transformations, intended for tackling the issues of particular class of refactoring problems.

Secondly, each of the refactoring operators can be applied to an arbitrary context-free grammar, including the situation of particular grammar form not allowing occurance of a specific transformation. In this case, the original grammar form is returned as a result of the transformation. On the other hand, each grammar refactoring pattern prescribes some specific preconditions a context-free grammar must fulfill in order to be transformable by a particular refactoring pattern.

In our approach, each pattern is represented as a specification consisting of a set of transformation rules, while transformation rule provides transformation on some subset of grammar's production rules that exhibit specific structural properties. In this notion of refactoring patterns, each instance of refactoring operator is actually a refactoring pattern which lacks explicit specification of required structural properties of grammar's production rules, and each refactoring pattern is in fact non-parametric refactoring operator.

VI. Core

For the purposes of patterns expression, we propose pLERO, pattern Language of Extended Refactoring Operators.

pLERO is currently being developed in two distinct dialects e.g. imperative [15] and functional. Refactoring patterns written in imperative dialect of pLERO are more process-centric, meaning that they are intended for specification of particular steps of a refactoring process, while refactoring patterns written in functional dialect are more result-centric and facilitate understanding of a grammar's structural changes. In this paper, we present the functional dialect of pLERO, while detailed description of the imperative dialect of pLERO can be found in [15].

A. pLERO

Through pLERO it is possible to define patterns for grammar refactoring or other transformations, applicable to grammars expressed in BNF. That is, pLERO is a language for defining parameterless operators of a problem class.

Pattern description consists of a set of transformation rules, while each rule comprises predicate describing the shape of a grammar's production rules, and transformation defining how production rules matched against predicate should be changed.

Predicate and transformation are expressed in similar fashion by formalism of meta-production rules. Each meta-production rule defines structure of some subset of a grammar's production rules. Predicate is specified by exactly one meta-production rule matched against grammar's production rules, while transformation is described by set of meta-production rules defining shape of production rules to be included in grammar during refactoring process.

Each meta-production rule can be divided in two parts, namely, left side of meta-production rule and right side of meta-production rule. Left side of meta-production rule specifies nonterminal on the left side of a grammar's production rule, while right side of meta-production rule specifies sequence of symbols that can be found on the right side of a grammar's production rules. Left side of meta-production rule

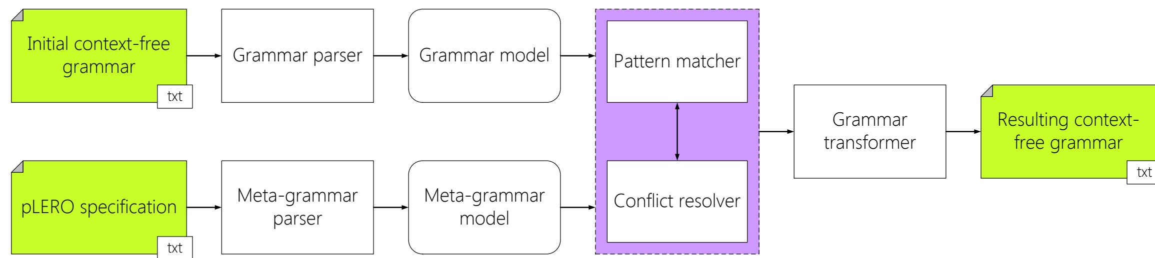

Fig. 5. Architecture of pattern application system

is some pattern variable, while right side of meta-production rule is concatenation of pattern variables.

Pattern variable specifies homogenous sequence of symbols in a grammar's production rules, consisting of variable prefix and name. Variable prefix describes possible matched symbols and their number, while variable name is identifier of this sequence. The prefix can be "`t`" for terminals, "`n`" for nonterminals, and "`s`" for both terminals and nonterminals. The letter specifying the symbol type can be followed by the asterisk "`*`" denoting the variable can match a sequence of symbols instead of a single symbol. For instance, the most generic variable type has prefix "`s*`" that can match any sequence of symbols. Variable prefix and name are separated by dot "`.`". After the dot, the variable name follows, e.g. "`s*.symbols`".

Pattern variable on the left side of meta-production rule may only have prefix "`n`" not followed by asterisk, denoting exactly one nonterminal. Each pattern variable on the right side of meta-production rule can have arbitrary valid prefix.

Each specification of refactoring pattern in pLERO must comply with the same template (Fig. 6) which allows specification of global pattern variables denoting same symbol sequences in all transformation rules of a pattern during entire refactoring process.

```
pattern [pattern_name] {
   variables:
         [prefix1].[variable_name1],
         [prefix2].[variable_name2],
            ...
         [prefixn].[variable_namen];
   new symbols:
         [prefixI].[variable_nameI],
         [prefixII].[variable_nameII],
            ...
         [prefixN].[variable_nameN];
   [transformation_rule1];
   [transformation_rule2];
      ...
   [transformation_rulem];
}
```

Fig. 6. Template of a pattern notation

The template also enables to specify new nonterminal symbols, which need to be generated for the use in production of new production rules. Notion of individual transformation rules must also follow specific template shown in Fig. 7.

```
predicate -> transformation
```

Fig. 7. Transformation rule decomposition

While variables may consist of all the possible prefixes, new variables may not; more specifically they cannot consist of partially deterministic constructs such as "`*`" or "`s`", and in current version of pLERO only "`n`" is allowed. Reason for this is that these constructs do not specify unambiguous concatenation of symbols and though it is not possible to generate definite sequence of symbols on their basis. Moreover new variables may be used only in meta-production rules contained within transformation part of transformation rule, and their use in predicate is prohibited. Reason for this is that new variables correspond with sequences of symbols that occur only in refactored grammar, and not in the original grammar.

1) Pattern Matching:

In order to apply transformation provided by refactoring pattern on some context-free grammar it is first necessary to match this grammar against this pattern. Process of pattern matching has two purposes:

- Determining if a grammar is transformable by a pattern
- Determining which pattern variable represents which sequence of symbols within production rules of a grammar

To each assignment of specific sequence of symbols to particular pattern variable we refer as to variable binding and to each variable representing definite sequence of symbols we refer as to bound variable.

Variables are bound during the matching of the rule and used in the replacement construction process. Global variables keep their value after they are bound during the first successful match. Other variables (to which we refer as to local variables) are bound only during the application of a rule and cleaned before the next matching.

The matching of a predicate against a grammar production is successful if all the pattern variables can be bound to a

part of the production and no unmatched symbols are left. Variables can match only some type of symbols, based on their prefix. Simple variables must match exactly one symbol of a specified type, while sequence variables can match any number of symbols (including zero).

For instance, the predicate "`n.1 ::= n.2 s*.1 t.1`" would match production "`A ::= B 'c' 'd' 'e'`", resulting in bindings "`n.1`" = "`A`", "`n.2`" = "`B`", "`s*.`" = "`'c' 'd'`", "`t.1`" = "`'e'`", and also "`B ::= D 'f'`", "`n.1`" = "`B`", "`n.2`" = "`D`", "`s*.`" is empty, "`t.1`" = "`'f'`". Since it does not start with a nonterminal, it would not match "`C ::= 'd' 'e' 'f'`".

If the pattern "`n.1 ::= s*.1 n.2 s*.2`" is matched against the production "`A ::= B C D`", the resulting binding would be "`n.1`" = "`A`", "`s*.1`" is empty, "`n.2`" = "`B`", and "`s*.2`" = "`C D`".

Variable prefix specifies only structure of some sequence of symbols, and it does not define particular symbols of a specific grammar. On the other hand, variable name is an identifier of a specific variable binding established during a particular pattern matching process. For instance, the predicate "`n.1 ::= n.1 n.2`" would match production "`A ::= A B`", however it would not match production "`A ::= B C`" since in this case variable `n.1` would be bound to two different nonterminals (`B` and `C`).

The matching of sequences is non-greedy. This means that short sequences are performed first during the matching. The process continues while the entire production is matched.

However, there are some cases in which conflicts in matching of predicate against production can arise, e.g. conflict always occurs if predicate contains two consecutive sequences of arbitrary symbols ("`s*.A s*.B`"). In this case, we have adopted first-match found resolution strategy.

2) Pattern Application:

Each transformation rule of refactoring pattern describes structure of some production rules and specifies new production rules that should replace this production rule. Predicate is a concatenation of pattern variables, which can match a sequence of production rule symbols and then represent these symbols in the transformation.

If a variable of the same type and name is present in a transformation rule, it will represent the same sequence of symbols in all its occurrences. In the transformation, meta-production rules have to consist only of the variables occurring on the predicate side of the transformation rule or in global variables. After the predicate matches any grammar production, its variables are bound to parts of the production and the replacement productions are constructed on the basis of transformation patterns.

If applied to a grammar, all transformation rules of a pattern are traversed in the order of their specification. Predicate is then matched against all the unprocessed productions of the original grammar. If the match is successful, replacement production is constructed and the production is replaced in the grammar.

Order of specification of transformation rules within a pattern is important, for it serves as conflict resolution mechanism in case when there are multiple predicates that can be matched against one production rule.

On the other hand, multiple production rules can be matched against one predicate, but only if all global variables of a predicate are bound to a same sequence of symbols in each production, and in that case replacement productions are constructed for each such rule.

B. Implementation

To be able to perform experiments and to demonstrate the correctness of the approach, automated pattern application system (Fig. 5) has been implemented, in which pLERO plays a central role.

The system takes the initial grammar and the pLERO pattern specification from the two different text files, and after the refactoring it creates new text file containing the resulting grammar.

The first text file is parsed by grammar parser which creates its representation in the form of grammar model, while the second is parsed by meta-grammar parser which creates meta-grammar model.

The core of the system is divided in two coexisting entities:

1) *Pattern matcher* – The purpose is matching of grammar model against meta-grammar model
2) *Grammar transformer* – The purpose is construction of replacement productions and generating of refactored grammar.

To resolve various conflicts occurring during the process of pattern matching, various resolution strategies are implemented in a separate module to which we refer to as a *conflict resolver*.

VII. Experimental Results

As an example, see Fig. 8 and 9 containing fragment of Algol 60 grammar [16] and pattern for immediate left-recursion removal (not direct). Then, Fig. 10 and 11 reflect equivalent grammar fragments produced after two sequential pattern applications.

After the first application of the pattern immediate left-recursion concerning nonterminal "`term`" was removed.

After the second application of the pattern immediate left-recursion concerning nonterminal "`factor`" was removed.

VIII. Conclusion

The most significant contribution, that we expect based on the results presented in this paper, is the contribution to automated grammar evolution. As such, our refactoring approach presents an appropriate basis for creation of new theory concerning automated task-driven grammar refactoring, while the provided experimental results as well as the other experiments [3] [8] explicitly demonstrate correctness and effectiveness of this approach.

However, achievement of this goal also requires deeper understanding and intensified research in refactoring operators,

```
term ::= factor
term ::= term multiplying_operator factor
multiplying_operator ::= 'x'
multiplying_operator ::= '/'
multiplying_operator ::= '÷'
factor ::= primary
factor ::= factor '↑' primary
primary ::= unsigned_number
primary ::= variable
primary ::= function_designator
primary ::= '('arithmetic_expression')'
```

Fig. 8. Fragment of Algol 60 grammar [16]

```
pattern LeftRecursionRemoval {
   variables: n.A;
   new symbols: n.A1;
   n.A ::= n.A s*.x ->
            n.A1 ::= , n.A1 ::= s*.x n.A1;
   n.A ::= s*.x ->
            n.A ::= s*.x n.A1;
}
```

Fig. 9. Example of a pattern for immediate left-recursion removal

```
term ::= factor N4
N4' ::=
N4' ::= multiplying_operator factor N4
multiplying_operator ::= 'x'
multiplying_operator ::= '/'
multiplying_operator ::= '÷'
factor ::= primary
factor ::= factor ↑ primary
primary ::= unsigned_number
primary ::= variable
primary ::= function_designator
primary ::= '('arithmetic_expression')'
```

Fig. 10. Resulting grammar after first application of refactoring pattern

as well as quality-based grammar metrics. Crucial part of this research are refactoring patterns, since they operate with knowledge derived from experience of language engineers, and thus they present an appropriate tool for converging of state-of-art and state-of-practice in the field of grammar refactoring.

In the future, we would like to focus on achieving greater abstraction power of the pLERO language, so it would for-

```
term ::= factor N4
N4 ::=
N4 ::= multiplying_operator factor N4
multiplying_operator ::= 'x'
multiplying_operator ::= '/'
multiplying_operator ::= '÷'
factor ::= primary N20
N20 ::=
N20 ::= '↑' primary N20
primary ::= unsigned_number
primary ::= variable
primary ::= function_designator
primary ::= '('arithmetic_expression')'
```

Fig. 11. Resulting grammar after second application of refactoring pattern

malize other knowledge considering refactoring problems and context of their occurrence, such as consequences of pattern's application on grammar's quality attributes. We would also like to adopt our approach to EBNF notation, which is structurally richer and would cause pattern matching to be more deterministic.

However, our vision goes even further, since mARTINICA and pLERO currently cover only one aspect of grammar adaptation, e.g. grammar refactoring, while the ultimate goal is to create universal approach covering other processes concerning grammarware engineering, e.g. grammar construction and destruction.

In case of interest, it is possible to download automated pattern application system from:
`http://plero.fei.tuke.sk`

ACKNOWLEDGMENT

This work was supported by project VEGA 1/0341/13 *Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.*

REFERENCES

[1] J. Cervelle, M. Crepinsek, R. Forax, T. Kosar, M. Mernik, and G. Roussel, "On defining quality based grammar metrics," in *Proceedings of International Multiconference (IMCSIT '09)*. Los Alamitos, USA: IEEE Computer Society Press, 2009, pp. 651–658.

[2] T. Mogensen, *Basics of Compiler Design*. Copenhagen, DK: University of Copenhagen, 2007.

[3] I. Halupka and J. Kollár, "Evolutionary algorithm for automated task-driven grammar refactoring," in *Proceedings of International Scientific Conference on Computer Science and Engineering (CSE'2012)*. Slovakia: Technical University of Košice, 2012, pp. 47–54.

[4] C. Alexander, *The Timeless Way of Building*. New York, USA: Oxford University Press, 1979.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York, USA: John Wiley & Sons, 1996.

[6] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 3, pp. 331–380, 2005.

[7] T. Alves and J. Visser, “A case study in grammar engineering,” in *Proceedings of 1st International Conference on Software Language Engineering (SLE’ 2008)*. Berlin-Heidelberg: Springer-Verlag, 2008, pp. 285–304.

[8] I. Halupka, J. Kollár, and E. Pietriková, “A task-driven grammar refactoring algorithm,” *Acta Polytechnica*, vol. 52, no. 5, pp. 51–57, 2012.

[9] N. Kraft, E. Duffy, and B. Malloy, “Grammar recovery from parse trees and metrics-guided grammar,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 780–794, 2009.

[10] R. Lämmel and C. Verhoef, “Semi-automatic grammar recovery,” *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.

[11] W. Lohmann, G. Riedewald, and M. Stoy, “Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars,” *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 110, pp. 133–148, 2004.

[12] K. Louden, *Compiler Construction: Principles and Practice*. Boston, USA: PWS Publishing, 1997.

[13] R. Lämmel, “Grammar adaptation,” in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME ’01)*. London, UK: Springer-Verlag, 2001, pp. 550–570.

[14] R. Lämmel and V. Zaytsev, “An introduction to grammar convergence,” in *Proceedings of the 7th International Conference on Integrated Formal Methods*. London, UK: Springer-Verlag, 2009, pp. 246 – 260.

[15] J. Kollár and I. Halupka, “Role of patterns in automated task-driven grammar refactoring,” in *2nd Symposium on Languages, Applications and Technologies (SLATE’13)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 171–186.

[16] R. L. Sites, *Algol-60 Version 5 Reference Manual*. Control Data Corporation (CDC), 1979. [Online]. Available: http://www.computinghistory.org.uk/det/7244/Algol-60-Version-5-Reference-Manual/