DSL for Grammar Refactoring Patterns

Ivan Halupka(✉)

Technical University of Košice, Letná 9, 04200 Košice, Slovakia
ivan.halupka@tuke.sk

Abstract. Grammar refactoring is a significant cornerstone of grammarware engineering, aimed at adjusting a formal grammar to specific requirements derived from the application environment, without affecting the language that a grammar generates. In our research, we focus on tackling the problems related to formal specification and automated application of well-known and newly-discovered refactoring procedures. One of our research results is a language for specification of the refactoring patterns to which we refer to as pLERO. In this paper, we present an extension of pLERO language aimed at expanding the scope of its applicability to additional classes of refactoring problems, such as folding and unfolding of grammar productions.

Keywords: Grammarware engineering · Grammar refactoring · Structural patterns · pLERO language

1 Introduction

Grammar refactoring is a non-trivial process of changing the form in which a formal grammar is expressed, with preserving the language that a grammar generates. Two or more formal grammars that generate the same language are called equivalent. The objective of the classical grammar refactoring is adjusting the form in which a grammar is expressed to specific requirements considering the future purpose of a grammar. In our research we focus on context-free grammars, since they are the most commonly used formal apparatus for expressing the abstract syntax of programming languages.

Although grammar refactoring is both of theoretical and of practical significance, for various subdomains of gramarware engineering it is still weakly understood and poorly practiced [1]. A current gap between state-of-art and state-of-practice can be clearly seen in the compiler design, where current state-of-art provides limited number of specialized refactoring procedures. This, in turn forces language designers to perform the majority of the refactoring procedures manually on the basis of their intuition. This is a problem mainly because such refactoring can be significantly difficult and error prone, while results in many cases cannot be verified, since proving equivalence of two grammars is in general an undecidable problem.

In our previous work, we addressed this issue by proposing two approaches to automated grammar refactoring, more specifically a probabilistic approach based

© Springer International Publishing Switzerland 2015
V. Zsók et al. (Eds.): CEFP 2013, LNCS 8606, pp. 446–458, 2015.
DOI: 10.1007/978-3-319-15940-9_12

on evolutionary algorithm, called *mARTINICA* (metrics Automated Refactoring Task-driven INcremental syntactIC Algorithm) [2,3], and a deterministic approach based on formal specification language called *pLERO* (pattern Language of Extended Refactoring Operators) [4,5].

pLERO is the domain-specific language for specification of refactoring and other transformations on context-free grammars. The core idea behind the approach is to provide universal formal apparatus for automated application of the knowledge of grammar engineers. The main purpose of pLERO is to uniformly define deterministic solutions to recurring refactoring problems, such as left recursion removal and elimination of epsilon productions. To these solutions we refer to as grammar refactoring patterns.

pLERO is currently being developed in two distinct dialects, namely a imperative and a declarative. Refactoring patterns written in the imperative dialect of pLERO are more process-centric, meaning that they are intended for the specification of particular steps of a refactoring process, while refactoring patterns written in the declarative dialect are more result-centric and facilitate the understanding of a grammar's structural changes. Detailed description of the imperative dialect of pLERO can be found in [4], while description of the declarative dialect of pLERO can be found in [5]. Refactoring patterns expressed in both dialects currently operate on grammars expressed in BNF notation.

In this paper we consider the declarative dialect of pLERO and present its extension aimed at addressing the following aspects of a pattern's formal specification:

- Parameterization of patterns, since the recently published [5] specification of pLERO only included support for expressing parameterless refactoring transformations.
- Matching of the negative grammar structures, meaning expressing structural preconditions that grammar should not fulfill in order to be transformable by a pattern, as opposed to previous version of pLERO, where only matching of positive grammar structures could be formally specified.
- Equivalence precondition for grammatical structures whose properties are expressed at multiple levels of abstraction.
- Iteration over structurally different grammar productions.

2 Grammar Refactoring Patterns

Refactoring patterns are the only first-class citizens of the pLERO language, specifying structural transformations of grammar's productions, and as such they can be considered generic schemes of refactoring operations.

A grammar refactoring pattern consists of a nonempty set of transformation rules, and a set of declarations. Each transformation rule defines alternation of grammar's production rules which exhibit some structural properties, while each declaration specifies additional properties of formal structures that occur in some of the transformation rules. We understand the term *'structural property of production rule'* as the ordering of symbols and symbol types production.

For instance, a production rule may exhibit the structural property that its right-hand side starts and ends with a nonterminal symbol.

A transformation rule consists of two parts, namely a *predicate* defining the structure of some subset of a grammar's production rules, and a *transformation* describing the way in which this structure should be changed. Although predicates and transformations have different purposes, they are both expressed in similar fashion using the formalism of *meta-production rules*. The predicate is specified by exactly one meta-production rule, while the transformation is defined by a set of meta-production rules.

Each meta-production rule specifies chosen structural properties exhibited by some subset of the grammar's productions. A meta-production rule is divided into a left-hand side describing left-hand side of a grammar's production rule, and a right-hand side specifying structure of a right-hand side of a grammar's production rule. The left-hand side of a meta-production rule comprises exactly one *pattern variable*, while the right-hand side of a meta-production rule is a sequence of *pattern variables*.

Each pattern variable defines a homogeneous sequence of grammar symbols, and as such consists of a *variable name* and a *variable prefix*. The variable prefix describes a type of grammar symbols that can occur in sequences assigned to a pattern variable, and the three possible variable prefixes are: $'t'$ denoting terminal, $'n'$ denoting nonterminal and $'s'$ denoting both terminal and nonterminal, while each of these prefixes can be followed by $'*'$, denoting sequences of arbitrary length, or $'\{m\}'$ denoting sequences of exactly $'m'$ symbols. The variable name serves as an identifier of a specific sequence of grammar symbols, and it enables us using this sequence in other parts of a transformation rule in which the pattern variable occurs (local pattern variable). It also enables us using this sequence in other transformation rules or declarations (global pattern variable) and adding a new nonterminal to the grammar (new pattern variable).

Each pattern specification in pLERO must follow the same notion template, as shown in Fig. 1, which has suffered minor changes since its publication [5] due to extension of pLERO language itself.

More detailed description of the pLERO language, the pattern matching and the pattern application processes can be found in [5]. In what follows we only discuss pattern declarations and ways of pattern parameterization, since these are parts of the language that have been subjected to change.

3 PLERO Extension

3.1 Pattern Parameterization

The main idea behind our previous refactoring approach to which we refer to as mARTINICA [2,3] was to perform grammar transformation on the basis of certain mathematically expressed objectives, while the refactoring process consisted of a series of incremental applications of the refactoring operators. In this case, we operated with a constant set of refactoring operators, which were implemented

```
pattern [pattern_name]
          (argument_1 annotation_1,
           ...
           argument_k annotation_k)
{
    [declaration_1];
    [declaration_2];
       ...
    [declaration_n];
    [transformation_rule_1];
    [transformation_rule_2];
       ...
    [transformation_rule_m];
}
```

Fig. 1. Template of a pattern notation

in the Java language. Initially pLERO was designed as complementary to this approach, with the intention of providing a simple DSL in which language developers can specify their own refactoring procedures and incorporate them in the base of refactoring operators. Patterns defined using the pLERO formalism were not parameterized, since in mARTINICA parameters are mostly generated randomly, and thus it was decided that any input arguments that were needed for refactoring were to be generated by the pLERO pattern matching environment.

However, the following two factors motivated us to incorporate support for pattern parameterization in the pLERO formalism:

- Recognition of the potential of pLERO to be used as formalism for preservation of newly discovered refactoring procedures and as a stand-alone tool for their application.
- Need for passing grammar-specific data that cannot be randomly generated or inferred from grammar's productions, such as start symbol.

Each refactoring pattern may have an arbitrary number of parameters. Each pattern parameter consists of an argument and an annotation. An argument can be a meta-production rule denoting the production rule of a specific structure, a pattern variable denoting specific sequence of symbols or an integer variable denoting the length of a sequence of symbols. Each argument has annotation describing its meaning, and each pattern variable occurring in arbitrary argument is considered to be a global pattern variable whose value cannot be altered during the pattern matching process. The types of the pattern arguments do not need to be declared, since they are inferred during the matching process. The way in which parameters are specified can be seen in Fig. 1.

3.2 Declarations

In this section we present five declarations within the pLERO language: variables, new symbols, join, equivalence and nonequivalence. The first two were part of the most recently published version of pLERO, however they were never closely examined, and the rationale behind them was never provided, which is the main reason why we also include their descriptions in this section.

Global Variables. In general, pattern variables with the same name and prefix occurring in different transformation rules represent distinct sequences of symbols. One advantage of such approach is the relatively large separation of concerns between individual transformation rules, which leads to a high level of structural integrity for production rules matched against specific predicate. This constraint also lowers the risk of the accidental structural corruption. However, in terms of generating the language, structurally diverse production rules may be closely interlinked. Preservation of grammar's equivalence may require that transformation of production rules exhibiting some structure must be conditioned by transformation of productions with different structures. Moreover, such production rules may have common substructures that need to be preserved or handled in a similar fashion, independently of structural differences that are observable when considering production rules as a whole.

In our experience, this scenario is actually quite common and such interlinkage is present in almost every meaningful refactoring pattern. A trivial example of such a connection between structurally different productions can be found in the pattern specifying the well-known procedure of immediate left-recursion removal. If a left-recursive nonterminal is reachable in any derivation, in order for the grammar to terminate, it must contain both left-recursive and non left-recursive productions of such nonterminal. In the process of left-recursion elimination both left-recursive and non-left-recursive productions need to be transformed, while the transformation pattern is different for each of these two structural classes of production rules. On the other hand, the recursive nonterminal on the left-hand side of each transformed production needs to be preserved, independently of the other structural properties.

In order to resolve this issue, we allowed sharing of pattern variables between transformation rules, however all shared variables must be explicitly declared using the *'variables'* keyword and the template notion, as depicted in Fig. 2. Pattern variables that are not specified using variables declaration and that are not implicitly global (such as pattern arguments) are interpreted as local pattern variables.

Generated Nonterminals. A refactoring process often involves the incorporation of new nonterminal symbols in a grammar. An example of a refactoring procedure, always leading to the incorporation of one new nonterminal in a grammar is the application of refactoring operator to which is referred to as pack [2,3]. This operator and its formal specification are more closely examined in Sect. 5.2

```
variables:
    [prefix1].[variable_name1],
    [prefix2].[variable_name2],
        ...
    [prefixn].[variable_namen];
```

Fig. 2. Variables declaration template

Names of nonterminals that need to be incorporated in a grammar could be passed as pattern arguments, or could be set to constants using equivalence declaration. However, this could lead to a naming conflict with existing nonterminals, which can break the structure of the entire language generated by a grammar. In order to resolve this issue, we created a declaration generating nonconflicting names of nonterminal symbols. This declaration is specified using the *'new symbols'* keyword and the template for it is depicted in Fig. 3. Each nonterminal pattern variable that is specified in a new symbols declaration represents a nonterminal with unique name that is not part of the original grammar, and as a consequence of this, variables declared in such fashion cannot be present in predicate, but only in the transformation part of a transformation rule.

```
new symbols:
    n.[variable_name1],
    n.[variable_name2],
        ...
    n.[variable_namen];
```

Fig. 3. New symbols declaration template

Production Alternatives. In the process of derivation the sentences of a language every nonterminal symbol in each derivation can be expanded by arbitrary production rule whose left-hand side is this nonterminal. This means that from a language standpoint, multiple productions with the same nonterminal on their left-hand side are alternatives directing the way in which language sentences develop. However, in BNF notation these alternatives are expressed as separate productions, and moreover from a structural standpoint, they may significantly differ among themselves.

In our experience, during the execution procedure of the various refactoring transformations, it is required that productions with equivalent left-hand sides be treated jointly, as alternatives occurring in one production whose left-hand

side is particular nonterminal. An example of such transformations is the application of well-known refactoring operators, which is referred to as fold and unfold [6]. In terms of BNF, unfolding means the replacement of specific nonterminal on the right-hand side of an arbitrary production with all right-hand sides of productions whose left-hand side is this nonterminal, while unfolding presents inverse transformation to fold and its execution is conditioned by existence of productions containing each alternative, as the only structural difference between them. The problem with the approach above is that during the language design phase any form of iteration (with the exception of iteration deriving from pattern recognition process) was excluded from pLERO, mainly for the reasons of simplicity and computational complexity.

We addressed this issue by proposing a declaration creating special kind of iterator over productions whose left-hand side is the same nonterminal and which exhibit particular structural properties. This declaration is specified using the 'join' keyword and the template notion, as depicted in Fig. 4. The pattern variable before the 'where' keyword represents a nonterminal over which iterator is created, while the meta-production rule after the 'where' keyword describes the structure of productions included in the iterator. All pattern variables included in this declaration are implicitly global, and in case of their occurrence in a predicate they specify the need for matching against all possible pattern bindings in the iterator. On the other hand, if they occur in a transformation, they specify the creation of productions which contain all possible pattern bindings in the iterator.

```
join n.[variable_name]
   where [meta_production_rule];
```

Fig. 4. Join declaration template

Notice that this declaration combines right-hand sides of production rules whose left-hand side is a same nonterminal into one production rule of EBNF notation, whose left-hand side is this nonterminal and whose right-hand side consists of right-hand sides of the combined productions between which EBNF alternative meta-operator has been put.

Equivalence and Nonequivalence. Various refactoring procedures can be performed only in the case of structural equivalence or nonequivalence of particular sequences of grammar symbols. In most cases, the first case is not an issue, since in pLERO, the precondition of the structural equivalence can be specified by using same pattern variables for equivalent structures, alternatively declaring these variables as global. However, there is an exception when this solution cannot be used, and that is in the case when one sequence of symbols needs to be

expressed using two or more distinct sequences of pattern variables. For example, when we specify refactoring operator which is referred to as pack, the production rule that needs to be transformed is passed as a pattern argument, which is typed as meta-production rule denoting arbitrary production. However, the transformation rule which specifies this operator needs to operate on more fine-grained structures of the production that is passed as pattern argument, since the application of the pack operator in general case requires dividing the production in three parts (symbols before packed sequence, packed sequence itself and symbols after packed sequence). Some refactoring procedures may also require that a grammar does not exhibit some structural properties, for example if refactoring precondition is that grammar must be in Chomsky normal form, then one of structural preconditions is that grammar does not include epsilon productions.

The mentioned refactoring problems present our motivation to extend pLERO with declarations of equivalence and nonequivalence. The equivalence precondition for two sequences of pattern variables (separated by 'and' keyword) is specified by keyword 'equivalence', and notion template, as depicted in Fig. 5, while nonequivalence precondition for two sequences of variables is specified in similar fashion, by replacing 'equivalence' keyword with 'nonequivalence'. All pattern variables used both in equivalence and nonequivalence declaration are implicitly global.

```
equivalence [prefix_1].[variable_name_1]
                ...
            [prefix_k].[variable_name_k]
   and
            [prefix_n].[variable_name_n]
                ...
            [prefix_m].[variable_name_m];
```

Fig. 5. Equivalence declaration template

4 Related Work

We were not able to find related research considering grammar refactoring patterns; however, any refactoring approach closely aimed for solving refactoring issues of a particular problem domain [7–9] can in some sense be considered a pattern.

Lämmel presented a suite of fifteen grammar transformation operators, four considering grammar construction, five considering grammar destruction and six considering grammar refactoring [6]. These operators are in large degree tailored for solving issues of two specific problem domains e.g. grammar adaptation and grammar recovery.

Lämmel and Zaytsev recently introduced a suite of four refactoring operators, specifically aimed for tackling refactoring tasks occurring in the process of grammar extraction from multiple diverse sources of information [10].

5 Discussion

This section examines the process of formal specification of two chosen refactoring operators using the pLERO language. The following discussion elaborates on difficulty of specifying solutions to commonly occurring refactoring problems using formal apparatus provided by previous version of pLERO. Subsequently, it describes a way in which proposed language extensions tackle these issues and thus provides justification of the new language features with relation to the purpose of the pLERO language.

Domain-specific languages trade generality for expressiveness in a limited domain [11]. We believe that a relative comparative advantage of using domain-specific language over formal apparatus provided by general-purpose languages should be evaluated in the terms of balance between the generality and the expressiveness of the language. Therefore, in our view, the growth of domain-specific language's expressive power is generally not a sufficient reason for its extension. In an ideal case, domain-specific languages should only be extended in situations in which a particular extension does not have a significant negative impact on the balance between the generality and the expressiveness of the language in the domain (in the opposite case, benefits of its usage over using a general-purpose language may be questioned).

In order to demonstrate, that the proposed language extensions fulfill this condition, we have also implemented both discussed refactoring operators in Java. However, comparison of expressive powers of different languages may be difficult, especially since to the best of our knowledge, there is no generally accepted methodology for performing such task. Therefore we decided to compare a number of language statements used to implement refactoring operators in both languages. For the analysis of Java code, we used tool Resource Standard Metrics (available at http://msquaredtechnologies.com), while in pLERO we evaluated this metric as sum of number of transformation rules and number of declarations. In the analysis of Java source code, we included only Java methods that implement logic of refactoring operator, while other parts of source code, such as grammar parser and grammar model were excluded from the analysis.

5.1 Case A: Unfold

Unfold is the refactoring operator that replaces each occurrence of a nonterminal on the right-hand side of some production rule with all possible combinations of right-hand sides of production rules whose left-hand side is this nonterminal. For instance, consider the grammar containing set of three production rules $\{A \rightarrow$ 'a' B 'a', $B \rightarrow$ 'a', $B \rightarrow$ 'b'$\}$. In case we unfold the nonterminal B, the resulting grammar will contain four production rules $\{A \rightarrow$ 'a' 'a' 'a', $A \rightarrow$ 'a' 'b' 'a', $B \rightarrow$ 'a', $B \rightarrow$ 'b'$\}$.

The unfolding operator is widely used in various procedures of grammarware engineering, such as post-processing of inferred grammars, and grammar convergence. Grammar inference is a process of extracting a correct grammar for unknown target language from a finite set of language examples [12]. The problem is that majority of approaches to grammar inference primarily aim at extracting a grammar of a correct language, focusing on issues related to over-generality and over-specialization of inferred grammar [13], while the form in which the extracted grammar is presented remains only a secondary concern if addressed at all. In this case, the unfolding operator may be repeatedly used on a grammar with the aim of reducing the count of grammar's nonterminal symbols, or reducing depths of derivation trees constructed for sentences of language generated by a grammar. Grammar convergence is a method of establishing and maintaining the connection between grammar knowledge contained within heterogeneous software artifacts. In this case, the unfolding operator is preferably used in the process of transformation of software artifacts, predominantly because it leads to semantics-preserving grammar transformations [10].

The Java method used to implement the unfolding operator consists of 37 language statements spanning over 47 effective lines of code, while the specification of the unfolding operator in pLERO required only 2 language statements. This pLERO specification is depicted in Fig. 6 and it consists of the proposed join declaration and the transformation rule that defines transformation on productions containing the unfolded nonterminal on their right-hand sides.

```
pattern Unfold
    (n.A 'Unfolded nonterminal')
{
    join n.A where n.A ::= s*.x;
    n.B ::= s*y1 n.A s*y2
          -> n.B ::= s*y1 s*.x s*.y2;
}
```

Fig. 6. Unfold pattern specification

Since there is an arbitrary number of productions whose left-hand side is the unfolded nonterminal, and all such productions need to be considered in each application of unfolding operator, it is clear that some form of iteration over grammar's productions is required. Iterations derived from multiple applications of pattern are in this case not sufficient. The rationale behind this claim can be derived from the fact that between two consequent applications of pLERO pattern no states are preserved, and thus the iterations derived from multiple applications of pattern must preserve grammar equivalence in each step of refactoring procedure. In the case of formal specification of unfolding operator this

condition cannot be satisfied without using the 'join' declaration, since grammar equivalence is preserved only if all productions containing the unfolded nonterminal on their left-hand side are used in the transformation, and since number of such productions is arbitrary, they generally cannot be matched in a single transformation step.

5.2 Case B: Pack

Pack is the refactoring operator that replaces the specific sequence of symbols contained within right-hand side of some production rule with newly created nonterminal, and creates new production whose left-hand side is this nonterminal and right-hand side is this sequence of symbols. Such sequence can be defined by the position of its initial symbol within production's right-hand side and by its length. For instance, consider the grammar containing set of two production rules $\{A \rightarrow$ 'a' 'a' B 'a' 'a', $B \rightarrow$ 'a' 'b'$\}$. In case we pack sequence of three symbols, starting from the second symbol of the first production the resulting grammar will contain three production rules $\{A \rightarrow$ 'a' NT 'a', $NT \rightarrow$ 'a' B 'a', $B \rightarrow$ 'a' 'b'$\}$, while NT will correspond with newly created nonterminal.

Pack may be used in various situations, with aim of reducing length of grammar's productions, reducing number of direct child nodes for each node of constructed derivation trees and improving grammar comprehension. The Java method used to implement the unfolding operator consists of 18 language statements spanning over 25 effective lines of code, while the specification of the unfolding operator in pLERO required only 4 language statements. This pLERO specification is depicted in Fig. 7 and it consists of three declarations and one transformation rule.

```
pattern Pack
   (n.A ::= s*.x 'Packed production',
    int.I 'Initial package symbol',
    int.L 'Package length')
{
   variables s{int.I}.y1, s{int.L}.p,
         s*.y2;
   equivalence s*.x and
         s{int.I}.y1 s{int.L}.p s*.y2;
   new symbols n.B;
   n.A ::= s{int.I}.y1 s{int.L}.p s*.y2
         -> n.A ::= s{int.I}.y1 n.B s*.y2,
            n.B ::= s{int.L}.p;
}
```

Fig. 7. Pack pattern specification

The general form of the pack operator, specifying pack operator for all possible sequences of symbols on right-hand side of production rule cannot be specified without parameterization of patterns since pLERO does not provide any other formalism for exact specification of sequences of symbols with variable length and ambiguous structural properties. By ambiguity in the above sentence we understand, the inability to exactly identify some sequence of symbols within production on the basis of definition provided by predicate of transformation rule. However, specific forms of pack operator (for instance, applying pack operator on sequence of three symbols starting from the second symbol of a production) can be described without parameterization of patterns, but since the count of such situations is infinite, the pack cannot be specified as their unification. The same applies for the equivalence declaration, since general form of pack operator also cannot be specified without it, and the reason for this is that arbitrary structure of production within pattern argument could not be unambiguously matched against specific structure of transformation rule describing the pack operator.

6 Conclusion

The most significant contribution of 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 patterns as well as some other experimental results [3,5] demonstrate the correctness and the applicability of our approach. We believe that the proposed extensions significantly increase the applicability of pLERO language for specification of various patterns occurring in the domain of grammar refactoring, while preserving relative balance between languages generality and expressive power.

In the future we would like to focus on increasing the abstraction power of the pLERO language, so it would formalize 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.

Acknowledgments. 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. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. (TOSEM) **14**(3), 331–380 (2005)
2. Halupka, I., Kollár, J.: Evolutionary algorithm for automated task-driven grammar refactoring. In: Proceedings of International Scientific Conference on Computer Science and Engineering (CSE 2012), pp. 47–54. Technical University of Košice, Slovakia (2012)

3. Halupka, I., Kollár, J., Pietriková, E.: A task-driven grammar refactoring algorithm. Acta Polytech. **52**(5), 51–57 (2012)
4. Kollár, J., Halupka, I.: Role of patterns in automated task-driven grammar refactoring. In: 2nd Symposium on Languages, Applications and Technologies (SLATE 2013), pp. 171–186. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2013)
5. Kollár, J., Halupka, I., Chodarev, S., Pietriková, E.: pLERO: language for grammar refactoring patterns. In: 4th Workshop on Advances in Programming Languages (WAPL 2013), Kraków, Poland (in print)
6. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
7. Louden, K.: Compiler Construction: Principles and Practice. PWS Publishing, Boston (1997)
8. Lohmann, W., Riedewald, G., Stoy, M.: Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars. Electron. Notes Theoret. Comput. Sci. (ENTCS) **110**, 133–148 (2004)
9. Kraft, N., Duffy, E., Malloy, B.: Grammar recovery from parse trees and metrics-guided grammar refactoring. IEEE Trans. Softw. Eng. **35**(6), 780–794 (2009)
10. Lämmel, R., Zaytsev, V.: An introduction to grammar convergence. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 246–260. Springer, Heidelberg (2009)
11. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
12. Stevenson, A., Cordy, J.R.: Grammatical inference in software engineering: an overview of the state of the art. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 204–223. Springer, Heidelberg (2013)
13. D'ulizia, A., Ferri, F., Grifoni, P.: A learning algorithm for multimodal grammar inference. IEEE Trans. Syst. Man, Cybern. - Part B **41**(6), 1495–1510 (2011)