A Task-driven Grammar Refactoring Algorithm

Ivan Halupka, Ján Kollár, Emília Pietriková

Dept. of Computers and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic

Corresponding author: ivan.halupka@tuke.sk

Abstract

This paper presents our proposal and the implementation of an algorithm for automated refactoring of context-free grammars. Rather than operating under some domain-specific task, in our approach refactoring is perfomed on the basis of a refactoring task defined by its user. The algorithm and the corresponding refactoring system are called mARTINICA. mARTINICA is able to refactor grammars of arbitrary size and structural complexity. However, the computation time needed to perform a refactoring task with the desired outcome is highly dependent on the size of the grammar. Until now, we have successfully performed refactoring tasks on small and medium-size grammars of Pascal-like languages and parts of the Algol-60 programming language grammar. This paper also briefly introduces the reader to processes occurring in grammar refactoring, a method for describing desired properties that a refactored grammar should fulfill, and there is a discussion of the overall significance of grammar refactoring.

Keywords: Grammar refactoring, evolutionary algorithm, refactoring processes, task-driven transformation.

1 Introduction

Our work in the field of automated grammar refactoring derives from the fact that two or more equivalent context-free grammars may have different forms. Although two equivalent grammars generate the same language, they do not necessarily share some other specific properties that are measurable by grammar metrics [2]. The form in which a context-free grammar is written may have a strong impact on many aspects of its future application. For example, it may affect the general performance of the parser used to recognize the language generated by the grammar [12], or it may influence, and in many cases limit, our choice of parser generator for use in implementing the syntactic analyzer [12].

Since there is a close relation between the forms in which a grammar is expressed and the purpose for which the grammar is designed, different grammars generating the same language become domain-specific formalizations of language. The ability to transform one grammar to another equivalent grammar therefore actually becomes the capability to shift between domains of the possible application of grammars. Although this ability makes each context-free grammar more universal in the scope of its application, its practical advantages may easily be overwhelmed by the difficulties that this approach can introduce. The problem is that grammar refactoring is in many cases a non-trivial task, and if done manually it is prone to errors, especially in the case of larger grammars. This is an issue, because there is in general no formal way of proving that two context-free grammars generate the same language, since this problem cannot be resolved.

In our work, we address this issue by proposing an evolutionary algorithm for automated task-driven grammar refactoring. The algorithm is called mARTINICA. The main idea behind our algorithm is to apply a sequence of simple transformation processes to a chosen context-free grammar in order to produce an equivalent grammar with the desired properties. The current state of development of the algorithm requires that the grammar's production rules be expressed in BNF notation. These refactoring processes are more closely discussed in section 4, while the refactoring algorithm itself is discussed in section 6. Desired properties of a grammar produced by the algorithm are defined by an objective function that we discuss in section 5. Finally, in section 7, we present experimental results when our algorithm was used for refactoring a context-free grammar.

2 Motivation

Grammarware engineering is an up-and-rising discipline in software engineering, which aims to solve many issues in grammar development, and promises an overall rise in the quality of grammars that are produced, and in the productivity of their development [5]. Grammar refactoring is a process that may occur in many fields of grammarware engineering, e.g. grammar recovery, evolution and customization [5]. In fact, it is one of five core processes occurring in grammar evolution, alongside grammar extension, restriction, error correction and recovery [1]. The problem is that, unlike program refactoring, which is well-established practice, grammar refactoring is little understood and little practised [5].

If there is a clear purpose for which the grammar is being developed, its specification for an experienced grammar engineer is usually not an issue. Problems arise when a grammar is being developed for multiple purposes [1], or when a grammar engineer lacks knowledge about the future purpose of the grammar. In the first case, the problem is usually solved by developing multiple grammars of one language [1]. This need to develop multiple grammars could be replaced by developing a single grammar generating a given language and automatically refactoring it to another form suited to satisfy certain requirements, thus increasing the productivity of the grammar engineer. In fact, this is one of the main objectives of our work in the field of grammar refactoring.

In cases when the grammar engineer lacks knowledge about some aspect of the future purpose of the grammar, its final shape may not satisfy some of the specific requirements, even if it generates correct language. In this case, the grammar must either be refactored or be rewritten from scratch, thus draining valuable resources. An automated or even semi-automated way of refactoring the grammar could produce significant savings in this redundant consumption of resources. These are not the only two scenarios where an efficient refactoring tool is needed. In fact, an automated approach can be useful in all cases where we have a grammar with a form that needs to be changed while preserving the language that it generates. In this case, we see two main domains for applying our algorithms, i.e. adaptation of legacy grammars, and grammar inference.

Parser generators and other implementation platforms for context-free grammars develop over time. Newly-established platforms and other tools operating with context-free grammars may require a form in which the grammar should be expressed that differs from the tools for the previous technological generation, or that operate with unequal efficiency over the same grammar forms. Kent Beck states that programs have two kinds of value: what they can do for today, and what they can do for tomorrow [4]. When we take this principle into the account, we can say that the ability to refactor a context-free grammar in order to adjust it to the requirements of current platforms is in fact the ability to add value to the legacy formalization of the language.

Grammar inference is defined as recovering the grammar from a set of positive and negative language samples [11]. Grammar inference focuses on resolving issues of over-generality and over-specialization of the generated language [3], while the form of the grammar is only a secondary concern. Grammar recovery tools in general do not allow their users enough fine-grained tuning options for recovering a grammar in the desired form, making it in many cases difficult to comprehend, and not useful until it has been refactored [6].

3 Related Work

We were able to find very little reported research in the field of automated grammar refactoring. The small amount of work that we did find is mostly concerned with refactoring context-free grammars in order to achieve 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 [6]. They present a three-step procedure consisting of grammar metrics computation, metrics analysis in order to identify candidate nonterminals, and transformation of the candidate non-terminals. The first and third step of this procedure are fully automated, while the process of identifying non-terminals to be transformed by replacing iteration with left recursion is done manually. This approach is called metrics-guided refactoring, since the grammar metrics are calculated automatically, but the resulting values must be interpreted by a human being, who uses them as a basis for making decisions necessary for resuming the refactoring procedure. The work also provides an exemplary illustration of the benefits of grammar refactoring, since left-recursive grammars are more useful for some aspects of the application of a grammar [8] and are also more useful to human users [9] than iterative grammars.

However, the procedure for left-recursion removal is a well-known practice in the field of compiler design. An algorithm for automated removal of direct and indirect left recursion can be found in Louden [10]. This approach is further extended by Lohman, Riedewald and Stoy [9], who present a technique for removing left-recursion in attribute grammars and semantic preservation while executing this procedure.

4 Refactoring processes

In our approach, we use grammar refactoring processes only as a tool for incremental grammar refactoring. Formally, a grammar refactoring process is a function that takes some context-free grammar $G = (N, T, R, S)$ and uses it as a basis for creating a new grammar $G' = (N', T', R', S')$ equivalent to grammar G. This function may also require some additional arguments, known as process parameters. We refer to each assignment of actual values to the required process parameters of the specific grammar refactoring process as refactoring process instantiation, and an instance of this refactoring process is referred to as a specific grammar refactoring process with assigned actual values of its required process parameters.

At this stage of development, we have experimented with a base of eight grammar refactoring processes (Unfold, Fold, Remove, Pack, Extend, Reduce, Split and Nop), the first three of which have been adopted from Ralf Läammel's paper on grammar adaptation[7], while the others are proposed by us. For the purpose of better understanding what refactoring processes really are, and how they work, in following subsections we briefly introduce two of them, namely Pack and Nop.

In our research, we tend to keep the process base as small as possible, and we try to keep the refactoring processes as universal as possible. This is mainly because, as the refactoring process base grows, the state space of possible solution grammars also grows, and thus the size of the process base has a significant impact on the calculation complexity of the algorithm. However, lack of domain-specific refactoring processes is compensated by the overall openness of the process base, which means that it is a relatively trivial task to expand it or reduce it. In fact, the only refactoring process required by the algorithm, which must reside at all times in the process base, is the Nop process.

4.1 Nop

Nop, or identical transformation, is a grammar refactoring process that transforms context-free grammar G into the same context-free grammar G, or in other words, it does not impose any changes on the grammar.

4.2 Pack

Pack is a grammar refactoring process that creates transformed grammar G' on the basis of three process parameters, i.e., a mandatory parameter called the packed production rule (Pr), and optional parameters initial package symbol (Ps) and package length (Pl). These parameters must have the following properties:

$$\begin{aligned}&Pr \in R,\\&Ps \in \mathbb{N} \wedge Ps \geq 0 \wedge Ps < \mathit{rightSideLength}(Pr),\\&Pl \in \mathbb{N} \wedge Pl > 0 \wedge Pl \leq \mathit{rightSideLength}(Pr) - Ps.\end{aligned}$$

In cases when the initial package symbol is not defined, we assume that $Ps = 0$. If the package length is not defined, we define it as $Pl = \mathit{rightSideLength}(Pr) - Ps$. Function *rightSideLength* returns the number of symbols contained within the right side of the production rule.

Pack replaces some sequence of symbols contained within the right side of the packed production rule with a new nonterminal, and creates a rule whose left side is this new nonterminal and whose right side is the sequence of symbols mentioned above. This sequence of symbols is defined by the initial package symbol and the package length. More precisely, it is a sequence of Pl symbols starting from the symbol whose position within the packed rule is $Ps + 1$.

5 Objective function

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

In our view, the objective function consists of two parts: *objective* and *state function*. Our automated 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 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.

Until now, we have experimented with some grammar size metrics [2], e.g. number of non-terminals (*var*) and number of production rules (*prod*). An example of an objective function defining the refactoring task to be performed on grammar G executable by our algorithm is:

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

6 Refactoring algorithm

The main idea behind our grammar refactoring algorithm is to apply a sequence of grammar refactoring processes to a chosen context-free grammar, in order to produce an equivalent grammar with a lower value of the objective function, when the objective is minimization, or a higher value of the objective function when the objective is maximization. Since it is an evolutionary algorithm, it also requires some other input parameters, in addition to the *initial grammar* and the *objective function*, in order to be executed. The algorithm requires three other input parameters: *number of evolution cycles*, *population size* and *length of life of a generation*. The first two of these parameters are characteristic for algorithms of similar type, while the third parameter is our own.

As shown in Fig. 1, which presents a white-box view of our algorithm, the central figure in mARTINICA is an abstraction called *population of grammars*. In our view, population of grammars is a set containing a constant number of grammar population entities. Its main property is that, after performing an arbitrary step in our algorithm, the number of elements in the population of grammars is always equal to the population size.

Further, we define a grammar population entity as an arranged triple of elements: *post-grammar*, *process* *chain of grammar generation*, and *difference in objective functions*. A post-grammar is a context-free grammar equivalent to the initial grammar. The process chain for grammar generation is a sequence of refactoring process instances that was used to create the post-grammar from the corresponding post-grammar of the previous generation. The number of refactoring process instances in each grammar generation process chain is always equal to the length of life of a generation. The difference in objective functions is the difference between the values of the objective function calculated for a post-grammar of the current population and the corresponding post-grammar of the previous population of grammars.

6.1 Refactoring process instantiation

All process instances occurring in our algorithm are created automatically in one of three procedures, which are referred to as *random process creation*, *random parameter creation*, and *identical process creation*.

Random process creation creates instance of a random refactoring process with random parameters. The first step in this procedure randomly selects a process from the base of grammar refactoring processes. In this procedure, each grammar refactoring process has the same probability of being selected. The second step in the procedure defines concrete process parameters for this process on the basis of the grammar to which the process instance will be applied. All possible combinations of process parameters that respect the restrictions defined by a specific refactoring process have the same probability of being generated in this procedure.

Random parameter creation creates a process instance originating from some other process instance. The two mentioned process instances share the same refactoring process, but their process parameters may differ, since new process parameters have been created in a procedure analogous to the second step of the random process creation procedure. The only exception to this rule occurs when there is no acceptable combination of process parameters for a given refactoring process to be applicable to the given context-free grammar. In this, the random parameter creation procedure returns an instance of the Nop refactoring process.

Identical process creation creates an instance of the Nop grammar refactoring process.

6.2 Creating an Initial population

In the first phase of the Automated Refactoring Algorithm, the initial population of the grammars is created, and as such this phase is not repeated throughout the algorithm.

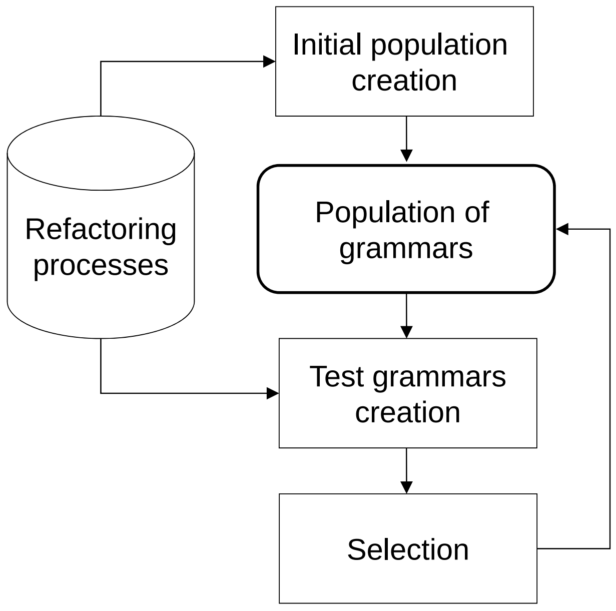

Figure 1: White-box view of mARTINICA.

The first step in this phase is to create grammar generation process chains for each grammar population entity. All process instances of each process chain created in this phase of the algorithm are created in the random process creation procedure, except for one, whose processes are all created in the identical process creation procedure. The reason for this exception is to guarantee that the initial grammar will be incorporated into the initial population of grammars. Since the sequence of process instances contained in the process chain must be applicable to the grammar for which are they being generated, in exact order, we must consider all changes to the grammar performed by one refactoring process instance in order to be able to generate the next process instance of the process chain. We solve this issue by generating intermediate grammars after each random process creation procedure by applying this refactoring process instance to the grammar for which the random refactoring process instance is being generated. We then generate the next random process instance of the process chain on the basis of the intermediate grammar. In order to better understand the idea behind this approach, we provide an example of creating a process chain consisting of three random refactoring processes for the initial grammar. This example is shown in Fig. 2.

The second step in the first phase of the algorithm creates corresponding post-grammars for each grammar population entity by applying its process chain to the initial grammar, and finally the third step calculates the difference of the objective function calculated for the initial grammar and the post-grammar of the corresponding grammar population entity.

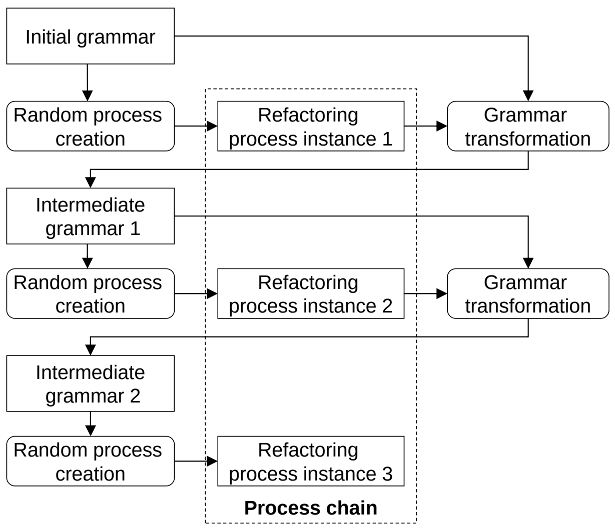

Figure 2: Creating a random process chain.

6.3 Creating test grammars

The second and third phase of the algorithm, called test-grammar creation and selection, are repeated in sequence for a number of evolution cycles. In test-grammar creation, we create three test grammar population entities for each grammar population entity. These entities are called *self-test grammar*, *foreign-test grammar*, and *random-test grammar.*

Self-test grammar is created on the basis of the corresponding grammar population entity and process chain, generated on the basis of the process chain of this entity. All refactoring process instances in the newly generated process chain are created in the random parameter creation procedure, and the algorithm for creating them is analogous to the algorithm for creating a random process chain in the initial population of the grammar creation phase. Self-test grammar is therefore a grammar population entity containing a grammar that was created on the basis of the same refactoring processes as those on which original tested grammar was created, but these processes may have different process parameters.

Foreign-test grammar is created in a similar procedure as for self-test grammar, with the exception that the new population entity is not created on the basis of a tested grammar process chain, but on the basis of some other grammar population entity process chain. This population entity is randomly selected from the population of grammars.

Random-test grammar is created in a procedure analogous to the procedure for creating a random grammar population entity in the first phase of the algorithm, with the exception that the random process chain is not being generated for an initial grammar, but for a grammar contained within the tested grammar population entity.

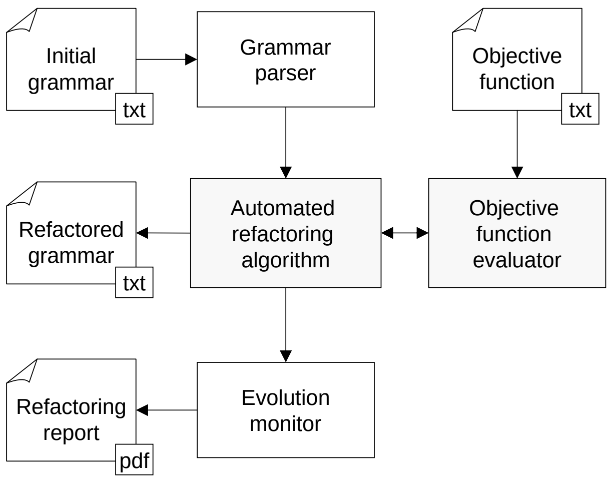

Figure 3: mARTINICA system architecture.

6.4 Selection and evaluation

In the selection phase of the algorithm, we compare the value of the objective function of each grammar within the population of grammars with the values of the objective function of the corresponding test grammars, and we choose the grammar with the best value of the objective function. This is the grammar which will be incorporated in the next generation of the population of grammars. When the chosen grammar is the tested grammar no changes occur, and the corresponding grammar population entity is preserved in the population of grammars. Otherwise, the tested grammar population entity is removed from the population of grammars and is substituted by the test grammar population entity with the best value of the objective function.

The fourth and final phase of the algorithm is performed after all evolution cycles have ended. In this phase, we compare the values of the objective function calculated for each grammar within the population of grammars, and we choose the grammar with the highest or lowest value, depending on our objective. This is the solution grammar, and as such is the result of automated refactoring.

7 Experimental results

7.1 mARTINICA implementation

In order to be able perform experiments and demonstrate the correctness of our approach, we implemented a grammar refactoring system in which mARTINICA plays a central role. The entire system is implemented in Java, and its architecture is shown in Fig. 3.

The refactoring system takes the initial grammar to be refactored and the objective function from two different text files and, after refactoring has been performed, it creates two files. The first of these is a

program	::=	*PROGRAM ident BEGIN commandSequence END*
commandSequence	::=	*command COMMA commandSequence*
commandSequence	::=	*command*
command	::=	*assignement*
command	::=	*declaration*
assignement	::=	*variable ASSIGN expression*
declaration	::=	*VAR ident TYPE type*
expression	::=	*variable operation expression*
expression	::=	*constant operation expression*
expression	::=	*variable*
expression	::=	*constant*
operation	::=	*PLUS*
operation	::=	*MINUS*
type	::=	*INTEGER*
type	::=	*REAL*
ident	::=	*IDENT*
variable	::=	*IDENT*
constant	::=	*NUMBER*

Table 1: Inital grammar.

text file containing the resulting grammar, and the second is a pdf file containing the evolution report.

The core of the systems is divided into two coexisting entities: an automated refactoring algorithm, and an objective function evaluator. The automated refactoring algorithm contains the implementation of the entire mARTINICA algorithm, the refactoring process base and the interactive user interface for obtaining the number of evolution cycles, the population size and the length of life of the generation. The initial grammar is taken from the text file, parsed by the grammar parser, which creates a grammar model on the basis of which the refactoring is done. The objective function is parsed by the objective function evaluator, which calculates the values of the objective function for all grammars provided by the automated refactoring algorithm. It does not provide an automated refactoring algorithm with a refactoring objective, since the algorithm always assumes that the objective is minimization, and if this is false the objective function evaluator transforms the state function so that it is equivalent to the native state function, but with the objective of minimization.

The entire refactoring process is monitored by the evolution monitor, which creates a report containing some analytical data concerning the specific refactoring process.

program	::=	*PROGRAM IDENT BEGIN commandSequence END*
commandSequence	::=	*command COMMA commandSequence*
commandSequence	::=	*command*
command	::=	*IDENT ASSIGN expression*
command	::=	*VAR IDENT TYPE REAL*
command	::=	*VAR IDENT TYPE INTEGER*
expression	::=	*IDENT PLUS expression*
expression	::=	*IDENT MINUS expression*
expression	::=	*NUMBER PLUS expression*
expression	::=	*NUMBER MINUS expression*
expression	::=	*IDENT*
expression	::=	*NUMBER*

Table 2: Refactored grammar.

7.2 Refactoring experiment

We experimented with a context-free grammar generating a simple assignment language. The grammar itself contains 11 non-terminals, 13 terminals and 18 production rules, whose BNF notation is shown in Tab. 1.

Symbols starting with a lowercase letter represent nonterminals, while symbols starting with uppercase letters represent terminals. The start symbol of the grammar is a non-terminal *program*. In our experiment, the refactoring task was described by an objective function from example (1). We iterated through 30 evolutionary cycles, with a population of 500 grammar population entities, and the length of life of a generation was set to 4. After refactoring had been performed, we obtained the grammar shown in Tab. 2.

The value of the objective function evaluated for the initial grammar was 40, while the value of the objective function evaluated for the refactored grammar is 20, which means that mARTINICA managed to reduce the value of the objective function by 50 %, and thus fulfilled the refactoring task. The development of the value of the objective function through the evolutionary cycles is illustrated in Fig. 4. The upper line illustrates the development of the average value of the objective function in all grammars within the population of grammars, while the lower line shows the development of the value of the objective function in the best grammar found within the population of grammars.

8 Conclusion

In this paper, we have presented our algorithm and software system for automated refactoring of context-free grammars. The main advantage of the algorithm

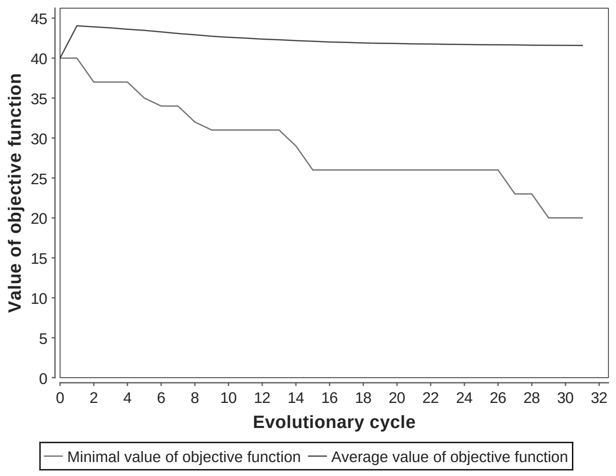

Figure 4: Values of the objective function through evolution.

is its relatively broad range of application, while its main disadvantages are relatively high computational complexity and slow propagation of positive changes within the population of grammars. In future, we will focus on resolving these issues and on expanding the implemented system to make it applicable for other aspects of grammar evolution, not only for refactoring.

Acknowledgements

The work presented in this paper was supported by SRDA Project of Slovak-Slovenian Research and Development No. SK-SI-0003-10 “Language Patterns in Domain-specific Languages Evolution”.

References

[1] T. L. Alves, J. Visser. A Case Study in Grammar Engineering. In *Proceedings of SLE’2008* (Eds. D. Gašević, R. Lämmel and E. Wyk), pp. 285–304. Springer-Verlag, Berlin-Heidelberg, 2009.

[2] J. Cervelle et al. On defining quality based grammar metrics. In *Proceedings of IMCSIT ’09. International Multiconference* (Eds. M. Ganzha and M. Paprzycki), pp. 651–658. IEEE Computer Society Press, Los Alamitos, 2009.

[3] A. D’ulizia, F. Ferri, P. Grifoni. A Learning Algorithm for Multimodal Grammar Inference. *Systems, Man, and Cybernetics, Part B: Cybernetics* **41**(6):1495–1510, 2011.

[4] M. Fowler et al. *Refactoring: improving the design of existing code.* Addison-Wesley, Boston, 1999.

[5] P. Klint, R. Läammel, C. Verhoef. Toward an Engineering Discipline for Grammarware. Transaction on Software Engineering and Methodology **14**(3):331–380, 2005.

[6] N. A. Kraft, E. B. Duffy, B. A. Malloy. Grammar Recovery from Parse Trees and Metrics-Guided Grammar Refactoring. *Software Engineering* **35**(6):780–794, 2009.

[7] R. Lämmel. Grammar Adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity* (Eds. J. Oliveira and P. Zave), pp. 550–570. Springer-Verlag, Berlin-Heidelberg, 2001.

[8] R. Läammel, C. Verhoef. Semi-Automatic Grammar Recovery. *Software: Practice and Experience* **31**(15):1395–1438, 2001.

[9] W. Lohmann, G. Riedewald, M. Stoy. Semantics-preserving Migration of Semantic Rules During Left Recursion Removal in Attribute Grammars. *Electron Notes Theor Comput Sci* **110**:133–148, 2004.

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

[11] M. Mernik et al. Grammar inference algorithms and applications in software engineering. In *Proceedings of ICAT 2009. XXII International Symposium* (Eds. A. Salihbegović, J. Velagić, H. Šupić and A. Sadžak), pp. 14–20. IEEE Computer Society Press, Los Alamitos, 2009.

[12] T. Mogensen. *Basics of Compiler Design.* University of Copenhagen, 2007.