Role of Patterns in Automated Task-Driven Grammar Refactoring*

Ján Kollár[1] and Ivan Halupka[2]

1 **Department of Computers and Informatics**
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`Jan.Kollar@tuke.sk`

2 **Department of Computers and Informatics**
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`Ivan.Halupka@tuke.sk`

Abstract

Grammarware engineering, and grammar-dependent software development has received considerable attention in recent years. Despite of this fact, grammar refactoring as a significant cornerstone of grammarware engineering is still weakly understood and little practiced. In this paper, we address this issue by proposing universal algorithm for automated refactoring of context-free grammars called mARTINICA, and formal specification language for preserving knowledge of grammar engineers called pLERO. Significant advantage of mARTINICA with respect to other automated refactoring approaches is that it performs grammar refactoring on the bases of user-defined refactoring task, rather then operating under some fixed objective of refactoring process. In order to be able to understand unified refactoring process of mARTINICA this paper also provides brief insight in grammar refactoring operators, which in our approach provide universal refactoring transformations for specific context-free grammars. For preserving of knowledge considering refactoring process we propose formalism based on patterns which are well-proven method of knowledge preservation in variety of other domains, such as software architectures.

1998 ACM Subject Classification F.4.3 Formal Languages (D.3.1)

Keywords and phrases Automated grammar refactoring, refactoring operators, grammar refactoring patterns, evolutionary algorithms, task-driven transformations

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.171

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 [3]. 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 [4], or it may influence, and in many cases limit, our choice of parser generator for use in implementing the syntactic analyzer [4].

* 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.*

The ability to transform one grammar to another equivalent grammar 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 is undecidable.

In our previous work [13], we addressed this issue by proposing an evolutionary algorithm for automated task-driven grammar refactoring. The algorithm is called mARTINICA (metrics Automated Refactoring Task-driven INcremental syntactIC Algorithm). The main idea behind this algorithm is to apply a sequence of simple transformation operators on a chosen context-free grammar in order to produce an equivalent grammar with the desired properties. Each refactoring operator transforms arbitrary context-free grammar into equivalent context-free grammar which may have different form than original grammar. Purpose of mARTINICA is to find sequence of refactoring operator instances that transforms specific context-free grammar into equivalent grammar whose form satisfies user-defined requirements. The current state of development of the algorithm requires that the grammar's production rules be expressed in BNF notation.

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

Pattern in general is a problem-solution pair in given context [14, 15]. Christopher Alexander argues that each pattern can be understood as an element of reality, and as an element of language [14]. Pattern as an element of reality is a relation between specific context, certain system of forces recurring in given context and certain spatial configuration that leads to balance in a given system of forces [14]. Pattern as an element of language is an instruction, which shows how certain spatial configuration can be repeatedly used in order to balance certain system of forces wherever specific context makes it relevant [14].

As such, patterns are means for documenting of existing, well proven design knowledge, and they support creation of systems with predictable properties and quality attributes [15]. In our view, role of patterns in the field of grammar refactoring is to preserve knowledge of language engineers about when and how to refactor context-free grammars and to support process of grammar refactoring by providing this knowledge. In order to incorporate patterns in the process of automated grammar refactoring we have coined new term – grammar refactoring patterns. Each grammar refactoring pattern describes a way in which a context-free grammar can be transformed, with preserving of language that it generates, specific situation in which this transformation is possible and consequences of this transformation on specific quality attributes of a context-free grammar. Description of situation in which transformation provided by specific pattern can be applied on specific grammar defines refactoring problem that pattern addresses. Grammar transformation provided by pattern defines solution of refactoring problem. Description of consequences of applying transformation provides context in which pattern should be used.

This paper is organized as follows. Section 2 provides motivation considering our research and briefly discusses possible domains of our approach's application. Section 3 discusses related work, while our refactoring algorithm is described in section 4. Section 5 presents some experimental results considering our refactoring approach, while grammar refactoring patterns are discussed in section 6.

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 [1]. Grammar refactoring is a process that may occur in many fields of grammarware engineering, e.g. grammar recovery, evolution and customization [1]. In fact, it is one of five core processes occurring in grammar evolution, alongside grammar extension, restriction, error correction and recovery [5]. The problem is that, unlike program refactoring, which is well-established practice, grammar refactoring is little understood and little practised [1].

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 [5], 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 [5]. 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. Ability to algorithmically change form in which context-free grammar is expressed makes it in this context more abstract and widens the scope of grammars possible application.

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. Example of such situation would be development of left-recursive grammar which should be parsed by LL(k) parser, in which case form of the grammar would not satisfy requirements considering parser implementation. 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 domains for applying our algorithm, 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 [6]. 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 [7]. Grammar inference focuses on resolving issues of over-generality and over-specialization of the generated language [8], 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 [9].

Sequence of refactoring operator instances provides transformation from some context-free grammar into another equivalent context-free grammar, and thus this sequence provides unidirectional formal relation between two equivalent grammars. In this context, a set of refactoring operators forms a universal vocabulary of grammar refactoring. On the other hand grammar refactoring patterns can be viewed as problem-specific refactoring operators and as such they form more abstract, domain-specific vocabulary of grammar refactoring. Sequence of grammar refactoring pattern instances does not only preserve relation between two equivalent grammars, but also captures rationale behind each refactoring decision and thus enables us to more deeply analyze and understand specific refactoring process.

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 [9]. 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 [10] and are also more useful to human users [11] than iterative grammars.

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 [12]. This approach is further extended by Lohmann, Riedewald and Stoy [11], who present a technique for removing left-recursion in attribute grammars and semantic preservation while executing this procedure.

4 Background

In this section we discuss refactoring operators, as a basis for understanding of grammar refactoring patterns and core idea of our approach. We also discuss our refactoring algorithm as a background related to implementation of mARTINICA and interpretation of experimental results. This section also briefly introduces reader to method of describing properties of context-free grammar via formalism of objective function, which in context of our approach is used as a specification of refactoring objective.

4.1 Refactoring Operators

Formally, a grammar refactoring operator 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 operator parameters. We refer to each assignment of actual values to the required operator parameters of the specific grammar refactoring operator as refactoring operator instantiation, and an instance of this refactoring operator is referred to as a specific grammar refactoring operator with assigned actual values of its required operator parameters.

At this stage of development, we have experimented with a base of eight grammar refactoring operators (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 [2], while the others are proposed by us.

Nop is operator of identical transformation, and as such it does not impose any changes on context-free grammar. Unfold replaces each occurrence of specific non-terminal within some subset of production rules with right side of production rules whose left side is this non-terminal, and in BNF notation this transformation can lead to increase in number of production rules. Fold replaces some symbol sequences on the right side of some subset of grammar's production rules with specific non-terminal, whose right side is this sequence of symbols, and as such this operator provides inverse function to unfold operator. Remove operator removes specific non-terminal and all production rules containing this non-terminal on their right or left sides from grammar, but only in case when this transformation does not impose changes on language that grammar generates. Pack replaces specific sequence of symbols within right side of certain production rule with new-non-terminal, and creates production rule whose left side is this non-terminal and whose right side is equivalent to this sequence. Extend introduces new non-terminal, creates production rule whose left side is this non-terminal and right side is some other non-terminal, and replaces all occurrences of this other non-terminal within some subset of grammar's production rules with this new non-terminal. Reduce operator removes multiple production rules with equivalent right sides, but only in case when this transformation preserves language that grammar generates. In case when there are multiple production rules whose left side is specific non-terminal, split creates new grammar in which each of such rules will have different non-terminal on its left side. More detailed description of individual refactoring operators can be found in [2, 13].

In our approach, we use grammar refactoring operators as a tool for incremental grammar refactoring. We tend to keep the number of operators as small as possible, and we try to keep the refactoring operators as universal as possible. This is mainly because, as the base of refactoring operators grows, the state space of possible solution grammars also grows, and thus the size of the base of operators 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 base of operators, which means that it is a relatively trivial task to expand it or reduce it. In fact, the only refactoring operator required by the algorithm, which must reside at all times in the base of the operators, is the Nop operator.

In this work, we propose grammar refactoring patterns, as addition to base of refactoring operators. However, key difference between operators and patterns in this context is that growth in the number of refactoring patterns in base of refactoring operators does not have significant negative impact on calculation complexity of the algorithm, and in many cases opposite is the true. This is caused by their domain-specific orientation and relatively narrow scope of refactoring tasks for which individual patterns are applicable.

4.2 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 [3], 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 is (1).

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

4.3 Refactoring Algorithm

The main idea behind our grammar refactoring algorithm is to apply a sequence of grammar refactoring operators 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 operator instances that was used to create the post-grammar from the corresponding post-grammar of the previous generation. The number of refactoring operator 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 generation.

4.3.1 Refactoring Operators Instantiation

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

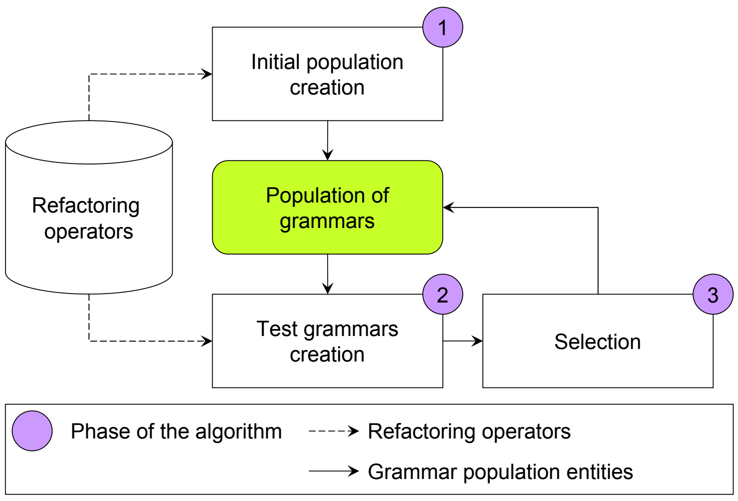

Figure 1 White-box view of mARTINICA.

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

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

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

4.3.2 Creating an Initial Population

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

The first step in this phase is to create grammar generation process chains for each grammar population entity. All operator instances of each process chain created in this phase of the algorithm are created in the random operator creation procedure, except for one, whose operators are all created in the identical operator 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 operator 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 operator instance in order to be able to generate the next operator instance of the process chain. We solve this issue by generating intermediate grammars after each random operator creation procedure by

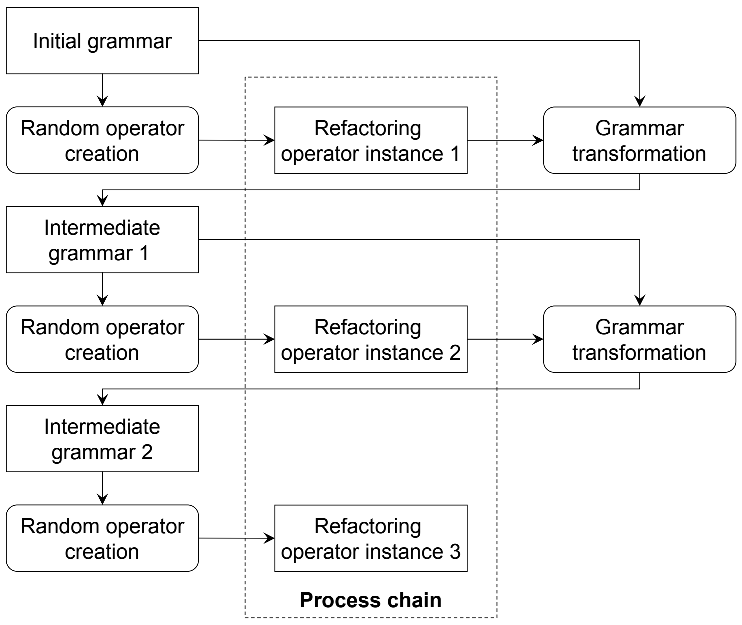

Figure 2 Creating a random process chain.

applying this operator instance on the grammar for which the random refactoring operator instance is being generated. We then generate the next random operator 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 operators 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.

4.3.3 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 operator 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 operators as those on which original tested grammar was created, but these operators 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.

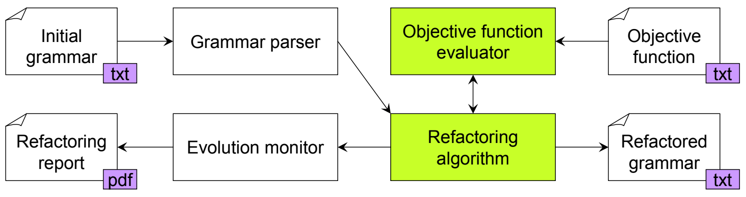

Figure 3 Architecture of Grammar Refactoring System.

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.

4.3.4 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.

5 Experimental Results

5.1 mARTINICA Implementation

In order to be able to 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 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

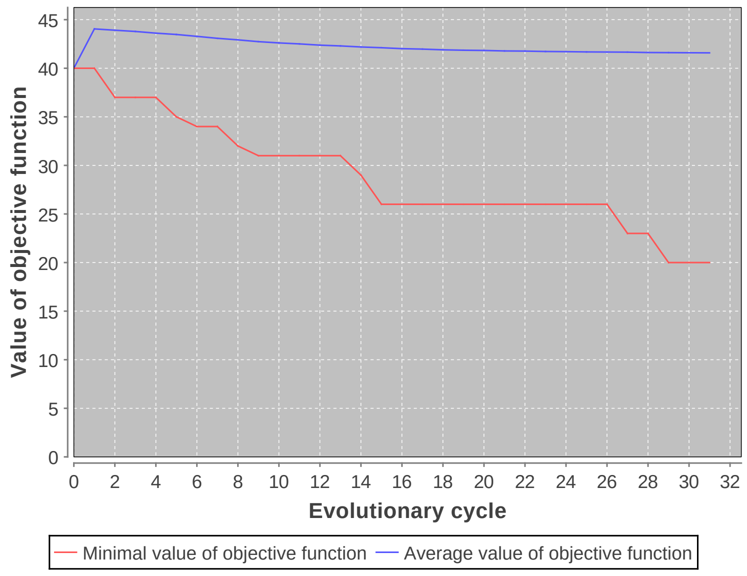

Figure 4 Values of the objective function through evolution.

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.

5.2 Refactoring Experiment

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. In this section we present results of refactoring experiment performed on context-free grammar generating a simple assignment language. This grammar consisted of 11 non-terminals, 13 terminals and 18 production rules expressed in BNF notation. In our experiment, the refactoring task was described by an objective function (1), while mARTINICA 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. 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. However, this interpretation is subjective, since in general there is no strict dividing line concerning reduction ratio at which refactoring task is fulfilled or not fulfilled, and this ratio may vary depending on aims of each individual refactoring process. The development of the value of the objective function through the evolutionary cycles is illustrated in fig 4.

We have chosen to present this experiment, because it explicitly illustrates the ability of mARTINICA to perform grammar refactoring tasks which have significant impact on values of grammar's size metrics. Other experiments however suggest that impact of our automated refactoring process on grammars structural metrics [3] is less significant. This is predominantly caused by incapacity of our refactoring operators to impose changes on grammar's structural metrics. On the other hand, refactoring patterns enable algorithm's users to extend base of refactoring operators, and create operators that suit their specific needs and requirements.

6 Grammar Refactoring Patterns

In our view, each grammar refactoring pattern provides equivalent transformation on context-free grammars and in this sense concept of grammar refactoring patterns is closely related to concept of refactoring operators. However there are some 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 that refactoring operators provide general transformations, whose usage is 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 our refactoring operators can be applied on arbitrary context-free grammar, including the situation when form of particular grammar does not allow specific transformation to occur, in which case original grammar form is returned as a result of a transformation. On the other hand, each grammar refactoring pattern prescribes some specific pre-conditions that context-free grammar must fulfill in order to be transformable by particular refactoring pattern.

In our approach, each grammar refactoring pattern is represented as specification consisting of three elements, which are context, problem and solution. Problem determines situation in which transformation provided by specific pattern can occur, context describes consequences of transformation on quality attributes of a context-free grammar, while transformation itself is specified in solution part of a pattern. In this notion of refactoring patterns, each refactoring operator is in fact refactoring pattern which lacks of explicit specification of a problem and a context. Problem part of a grammar refactoring pattern is described in the terms of grammar's quality attributes, and structural properties of grammar's production rules. Context part of a pattern is described in the terms of variations in values of grammar's quality attributes.

6.1 Specification of Grammar Refactoring Pattern

For purpose of expressing grammar refactoring patterns and in order to incorporate them in refactoring process of mARTINICA algorithm, we propose a language for formal specification of refactoring patterns called pLERO (pattern Language of Extended Refactoring Operators). Each refactoring pattern in pLERO is specified using schema in Listing 1, which consists of pattern name and three sub-specifications which describe context, problem and solution.

Context describes the effect that the grammar transformation provided by a solution has on chosen grammar metrics, in terms of increase or decrease in their values. Specification of context in pLERO is a set of metric-impact pairs, while each metric-impact pair describes impact of refactoring pattern on specific grammar metric. Purpose of context specification is to define a class of refactoring tasks to fulfillment of which can certain pattern contribute.

We can interpret Listing 2 as: Application of this pattern can lead to decrease in number of left recursive rules, and also increase in number of production rules

Problem defines structural properties that some production rules of a context-free grammar must and must not have, and also quality attributes that a context-free grammar must exhibit, in order to be transformed by instance of a refactoring pattern. By structure of grammar's production rule we mean order of specific terminal and non-terminal symbols on

Listing 1 Schema of grammar refactoring pattern specification.

```
PATTERN: [Pattern name]
        CONTEXT:
                [Context specification]
        END_CONTEXT
        PROBLEM:
                [Problem specification]
        END_PROBLEM
        SOLUTION:
                [Solution specification]
        END_SOLUTION
END_PATTERN
```

the right side of a production rule and occurrence of specific non-terminal on the left side of a production rule.

In order to specify this structure we have coined the term meta-structures of production rules. Each meta-structure of production rule is a structural model of specific sequence of arbitrary terminal and non-terminal symbols. Sequence of specific terminal and non-terminal symbols can be assigned to particular meta-structure of production rule only in case when this sequence exhibits structural properties prescribed by this meta-structure. To each assignment of sequence of specific terminal and non-terminal symbols to specific meta-structure of production rule we refer to as meta-structure instantiation, and to each meta-structure to which particular sequence of specific terminal and non-terminal symbols has been assigned we refer as to instance of meta-structure of production rule. We distinguish between two types of meta-structures of production-rules, namely primitive and composite meta-structures. Three kinds of primitive meta-structures of production rules are meta-non-terminals, meta-terminals and meta-symbols, while meta-non-terminal corresponds to arbitrary non-terminal symbol, meta-terminal corresponds to arbitrary terminal symbol and meta-symbol corresponds to arbitrary symbol of context-free grammar, regardless of the fact is this symbol terminal or non-terminal. Instance of meta-non-terminal is a specific non-terminal symbol, instance of meta-terminal is a specific terminal symbol and instance of meta-symbol is either a specific terminal symbol or a specific non-terminal symbol of a context-free grammar. Composite meta-structures are structural models made of primitive meta-structures. We propose one kind of composite meta-structure, which is meta-closure. Each meta-closure is a sequence of repeating primitive meta-structures, while instance of particular meta-closure is specific sequence of terminal and non-terminal symbols. Two meta-structure instances can be compared only if they were instantiated on the basis of a same meta-structure, and they are equivalent only when they refer to the same sequence of a specific terminal and non-terminal symbols, otherwise they are not equivalent.

Structure of right side of context-free grammar's arbitrary production rule can be described by a specific sequence of meta-structures, and each production rule can be viewed as sequence

Listing 2 Example of pLERO context specification.

```
CONTEXT
        minimizes countOfLeftRecursiveRules;
        maximizes prod;
END-CONTEXT
```

Listing 3 Example of pLERO declarations specification.

```
DECLARATIONS:
        Nonterminal1: NONTERMINAL;
        ArbitrarySequence1: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence2: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence3: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence4: CLOSURE ('ANY_SYMBOL', 0);
END-DECLARATIONS
```

of meta-structure instances on its right side and an instance of meta-non-terminal on its left side.

Specification of a problem in pLERO consists of a four parts, which are declarations, positive-match, negative-match and forces.

In declarations part all meta-structure instances and composite meta-structures of grammar production rules are specified. In meta-structure instances specification we always assume that two or more different, but comparable meta-structure instances are always not equivalent. This means that if we want to allow situation where more meta-structure instances specify same sequence of terminal and non-terminal symbols, we have also to specify the number of composite meta-structures that is equivalent to the count of such meta-structure instances, and assign each meta-structure instance to different meta-structure.

We can interpret Listing 3 as: Declaration of one meta-non-terminal instance called Nonterminal1, Declaration of four composite meta-structures which can be matched against any number (including zero) of any grammar symbols, and declaration of one instance of each composite meta-structure. These are the only meta-structures which can be used for specification of structural properties of a context-free grammar in entire problem part of pLERO specification.

In positive-match part of pLERO problem specification, structural properties that some subset of grammar's production rules must exhibit are defined. Positive-match is defined as a set of production meta-rules. Each production meta-rule defines a structure of one production rule, and it consists of label, left side and right side of production meta-rule. Label is unambiguous identifier of production meta-rule and enables us to manipulate with whole grammar's production rule whose structure is represented by given meta-rule, while this manipulation occurs in solution part of a pLERO specification. Left side of production meta-rule is some meta-non-terminal and right side of production meta-rule is some sequence of meta-structure instances.

In order to some grammar exhibit required structural properties it must contain production rules whose structural properties match with each of production meta-rules specified in positive-match. To matching of production rule to production meta-rule we refer as to production rule labeling. Each grammar's production rule can be labeled with at most one production meta-rule, and each production-meta rule can be used for labeling at most one production rule. This however can lead to two kinds of non-determinism, non-determinism in selecting of the production rule which will be labeled, and non-determinism in selecting of production meta-rule by which production rule should be labeled. In case when there are multiple production rules which match one production meta-rule and in case when there are multiple production meta-rules to which one production rules matches, sophisticated strategy for conflict resolution is required. Specific context-free grammar exhibits required structural properties only in case when all production meta-rules have been used for labeling, but in this case not all production rules have to be labeled.

Listing 4 Example of pLERO positive-match specification.

```
POSITIVE-MATCH:
[Rule1] Nonterminal1 :: Nonterminal1 ArbitrarySequence1;
[Rule2] Nonterminal1 :: ArbitrarySequence2;
END-POSITIVE-MATCH
```

Listing 5 Example of pLERO negative-match specification.

```
NEGATIVE-MATCH:
Nonterminal1 :: ArbitrararySequence3 Nonterminal1 ArbitrarySequence4;
END-NEGATIVE-MATCH
```

We can interpret Listing 4 as: Grammar must contain at least two production rules whose left side is the same non-terminal. Right side of one of these rules must also start with this non-terminal followed by arbitrary sequence of symbols, while right side of other rule is an arbitrary sequence of symbols. Rule1, and Rule2 are labels of production meta-rules.

In negative-match part of pLERO problem specification, structural properties that some subset of grammar's production rules must not exhibit are defined. Negative-match is defined similarly as positive-match, and it is represented by a set production meta-rules, however in this case all production meta-rules are without labels. In order to grammar exhibit structural properties that prevent transformation provided by a pattern solution, similarly as in positive-match all production meta-rules in negative-match must be used for labeling.

Labeling of production rules with production meta-rules specified in positive-match, and labeling of production rules with production meta-rules specified in negative-match are two separate but interlinked processes. This means that some production rule can be labeled with one production meta-rule of positive-match and at the same time this production rule can be labeled with one production meta-rule of negative-match. However some meta-structure instance that occur in arbitrary production meta-rule of positive-match and same meta-structure instance occurring in arbitrary production meta-rule of negative-match represents in both cases same sequence of same symbols.

We can interpret Listing 5 as: Grammar does not contain rule whose left side is instance Nonterminal1 and whose right side containts this instance.

In forces part of pLERO problem specification additional quality attributes that grammar must posses are defined. This quality attributes are expressed using relational expressions, whose variables are grammar metrics, and logic operators between these expressions. In this part of specification meta-structure instances can also be used, as arguments of grammar metrics. From a global point of view pLERO specification of forces is one logic expression, which if evaluated as true, indicates that grammar possesses required quality attributes, and if evaluated as false, shows that grammar does not exhibit required quality attributes.

We can interpret Listing 6 as: Grammar contains exactly two production rules whose left side is non-terminal matched against meta-non-terminal instance Nonterminal1.

Solution in pLERO provides transformation on context-free grammar, in case that this

Listing 6 Example of pLERO forces specification.

```
FORCES:
        RulesLeftSide(Nonterminal1) = 2
END-FORCES
```

Listing 7 Example of pLERO solution specification.

```
SOLUTION:
INTRODUCE_NONTERMINAL (Nonterminal2);
REMOVE_PRODUCTION (Rule1);
REMOVE_PRODUCTION (Rule2);
INTRODUCE_PRODUCTION ([New1]
                 Nonterminal1:: ArbitrarySequence2 Nonterminal2);
INTRODUCE_PRODUCTION ([New2]
                 Nonterminal2:: ArbitrarySequence1 Nonterminal2);
INTRODUCE_PRODUCTION ([New3] Nonterminal2:: EMPTY_SYMBOL);
END-SOLUTION
```

grammar possesses structural properties defined in positive-match of problem specification, exhibits quality attributes defined in forces specification and does not posses structural properties defined in negative-match of problem specification. This transformation can be only related to meta-structure instances used in positive-match specification, and production rules which have been labeled. To specify this transformation we use relatively simple imperative language, which manipulates with meta-structures and production meta-rule labels as with constants, and allows execution of some operations on context-free grammar. Since transformation provided by solution is in fact refactoring operator, here we will not discuss it in detail.

We can interpret Listing 7 as: Introduce new non-terminal of grammar and match it against meta-non-terminal instance Nonterminal2, remove productions with labels Rule1 and Rule2 and introduce three new production rules. Along with Listing 2, Listing 3, Listing 4, Listing 5, Listing 6 this solution specifies a method of removing left-recursion in case when one grammar's production rule is left recursive and other is not.

7 Conclusion

In this paper, we discussed universal algorithm for grammar refactoring, as well as unified method for preservation and automated application of language engineer's knowledge. As such, our refactoring approach presents appropriate basis for creation of new theory concerning automated task-driven grammar refactoring, while provided experimental results explicitly demonstrate correctness and effectiveness of this approach. However, achievement of this goal also requires deeper understanding and intensified research in refactoring operators, as well as quality-based grammar metrics. Crucial part of this research are grammar refactoring patterns, since they operate with knowledge derived from experience of language engineers and thus they present appropriate tool for converging of state-of-art and state-of-practice in the field of grammar refactoring. Possibility to simply enlarge base of refactoring operators and grammar refactoring patterns greatly benefits to adaptability of our algorithm and also contributes to significant degree of our approach's flexibility.

In future we would like to focus on resolving some known issues concerning our approach, such as relatively slow propagation of positive changes within the population of grammars, which is indicated by the fact that gap between relative quality of a best found grammar and average quality of grammars within the population grows in each evolutionary cycle, which is the phenomena that can be clearly observed on Fig. 4. We would also like to focus on achieving greater abstraction power of pLERO language, since currently count of production rules on which refactoring pattern operates is limited by number of production meta-rules contained in positive-match part of pLERO problem specification.

However, our vision goes even far, since mARTINICA currently covers only one aspect of grammar adaptation, e.g. grammar refactoring, while ultimate goal is creation of universal approach covering other processes concerning grammarware engineering e.g. grammar construction and grammar destruction.

References

1 P. Klint, R. Lämmel and C. Verhoef. *Toward an engineering discipline for grammarware*. ACM Transactions on Software Engineering Methodology, Vol.14, No.3, 2005, pp. 331-380.

2 R. Lämmel.*Grammar Adaptation*. In Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01), 2001, J. Oliveira and P. Zave (Eds.). Springer-Verlag, London, UK, pp. 550-570.

3 J. Cervelle, M. Crepinsek, R. Forax, T. Kosar, M. Mernik and G. Roussel. *On defining quality based grammar metrics*. In Proceedings of IMCSIT '09. International Multiconference (IMCSIT '09), 2009, M. Ganzha and M. Paprzycki (Eds.). IEEE Computer Society Press, Los Alamitos, USA, pp. 651-658.

4 T. Mogensen. *Basics of Compiler Design*. University of Copenhagen, Copenhagen, DK, 2007.

5 T.L. Alves and J. Visser. *A Case Study in Grammar Engineering*. In Proceedings of 1st International Conferenceon Software Language Engineering (SLE' 2008), 2008, D. Gašević, R. Lämmel and E. Wyk (Eds.). Springer-Verlag, Berlin-Heidelberg, pp. 285-304.

6 M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, USA, 1999.

7 M. Mernik, D. Hrncic, B.R. Bryant, A.P. Sprague, J. Gray, L. Qichao and F. Javed. *Grammar inference algorithms and applications in software engineering*. In Proceedings of ICAT 2009. XXII International Symposium (ICAT 2009), 2009, A. Salihbegović, J. Velagić, H. Šupić and A. Sadžak (Eds.). IEEE Computer Society Press, Los Alamitos, USA, pp. 14-20.

8 A. D'ulizia, F. Ferri, and P. Grifoni. *A Learning Algorithm for Multimodal Grammar Inference*. Trans. Sys. Man Cyber. Part B, Vol.41, No.6, 2011, pp. 1495-1510.

9 N.A. Kraft, E.B. Duffy and B.A. Malloy. *Grammar Recovery from Parse Trees and Metrics-Guided Grammar*. Software Engineering, Vol.35, No.6, 2009, pp. 780-794.

10 R. Läammel and C. Verhoef. *Semi-Automatic Grammar Recovery*. Software: Practice and Experience, Vol.31, No.15, 2001, pp. 1395-1438.

11 W. Lohmann, G. Riedewald and M. Stoy. *Semantics-preserving Migration of Semantic Rules During Left Recursion Removal in Attribute Grammars*. Electron. Notes Theor. Comput. Sci., Vol.110, 2004, pp. 133-148.

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

13 I. Halupka, J. Kollár. *Evolutionary algorithm for automated task-driven grammar refactoring*. In Proceedings of International Scientific Conference on Computer Science and Engineering (CSE 2012), 2012, pp. 47-54.

14 C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, USA, 1979.

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