

# Towards the Automatic Evolution of Reengineering Tools

Massimiliano Di Penta\* and Kunal Taneja\*\*  
dipenta@unisannio.it, kunalt@iitg.ernet.in

\*RCOST - Research Centre on Software Technology  
University of Sannio, Department of Engineering - Via Traiano - 82100 Benevento, Italy

\*\*Indian Institute of Technology Guwahati, India

## Abstract

*Building reverse engineering or reengineering tools often requires parsers for many different programming languages. The diffusion of dialects and variants makes many available parsers almost useless. While manual grammar maintenance is feasible, it can be a long, tedious and expensive task.*

*This paper proposes to adopt genetic algorithms to evolve existing grammars inferring changes from examples written using the dialect. Applying grammar inference from scratch may lead to a useless grammar, while the proposed approach simply applies changes to the original grammar when needed, thus producing a meaningful grammar. The paper reports some preliminary results related to the evolution of a C grammar.*

**Keywords:** Grammar Inference, Genetic Algorithms, Source Code Analysis

## 1. Introduction

Source code parsing is an essential task for many different software engineering related activities, such as metrics extraction, reverse engineering or reengineering. Many new programming languages appeared in recent years, each one having its own peculiarities that need to be considered when developing a parser. Moreover, while some languages have been standardized, dialects have been developed by different providers. Much in the same way, some specific-purpose languages have been developed using lexicon and syntax similar (while not equal) to traditional languages (e.g., PHP, JSP, VBScript, or Javascript). As a consequence, while grammars and parsers are in theory available, in practice most of them are almost useless, since a large percentage of the existing source code is written using language variants that are not 100% compliant with the standard. For

example, a large part of the open source software is written using the GNU C dialect.

Another issue to be considered is represented by the evolution of programming languages. For example, the Java syntax has changed since Java 1.0, and ANSI C 99 introduced several changes in the C syntax. Different approaches can be followed to tackle the lack of existing robust parsers for a language dialect:

- *Using island or lake parsers:* if we are interested to deal with only some specific subsets of a language syntax (e.g., when extracting metrics or dealing with an embedded language such as SQL), we may adopt the *island parsing* technology [15]. Much in the same way, *lake parsers* may skip syntax differences introduced, for example, by language dialects. When we need to perform thorough analyses or transformations, however, this technology may not be as effective as needed.
- *Patching compilers:* the idea is to patch a compiler in order to rely on its own AST or, when possible (as for *gcc*) to enable AST dumping. An extensive discussion of this approach has been made by Antoniol et al. [3]. The main weaknesses are due to the verbosity of the AST produced and to the difficulties related to hacking the compiler's source code.
- *Writing the grammar by hand:* this is, of course, the obvious way for approaching the problem, although it can be long, tedious and expensive.

This paper proposes to use Genetic Algorithms (GAs) [9] to evolve an existing grammar towards a grammar able to parse a dialect or a language variant. As described in Section 3, automatic evolution needs to be complemented with a process encompassing some manual steps. The paper shows, evolving a C grammar, that the approach is feasible and that the obtained grammars are usable.

## 2. Related Work

Grammar inference [1, 6, 14, 16] has been successfully applied for years to different fields, such as Natural Language Processing (NLP) [7], bioinformatics [8], and data mining [13]. Despite that, only a few attempts have been made to apply evolutionary technique for inferring context-free grammars of general-purpose programming languages. The main cause of these failures should be found in the size and complexity of the grammars to be inferred. Also, while for NLP the way rules are built (i.e., the way verbs, nouns, articles are put together to form a sentence) may not be particularly relevant, when using a parser we need to understand the meaning of the inferred non-terminal nodes. For example, to build a Control Flow Graph we are interested to know which are the nodes related to control structures.

The difficulties related to obtaining grammars for the wide variety of existing languages has been highlighted in the work of Lämmel and Verhoef [11, 12]. They also experienced that, for large programming languages, a complete inference is not feasible, and suggested that process of recovering a grammar is, in general, semi-automatic. Our believe is that, while grammar inference is useless, a semi-automatic process for grammar recovery can benefit of automatic grammar evolution for some limited tasks, such as completing a grammar, obtaining a grammar for a dialect or for a language similar to the starting language.

An interesting approach to infer small grammars was proposed by Javed et al. [10]. We share with them the idea that restricted grammar can be successfully inferred. While they suggest, whenever possible, to start from a partial set of rules, we start from a partial grammar and use (see Section 3) differences as a penalty factor. Cyre [4] proposed an approach to acquire a language from a few rules, as well as to integrate a learning classifier system with GAs [5].

## 3. Approach Description

To benefit of the automatic grammar evolution, we defined a semi-automatic process, described below. Manual and automatic steps are distinguished in the process.

1. The first (*manual*) activity aims to identify the main differences between the source and the target language. This information can be obtained from reference manuals, by inspecting the source code, as well as from errors generated when parsing the target language with the starting language parser. The identified differences can be used to reduce GA search space, by properly defining examples and the starting grammar.
2. Especially when a grammar is huge and it is composed of several (almost) independent sections, it can be useful to (*manually*) partition it and let a single partition

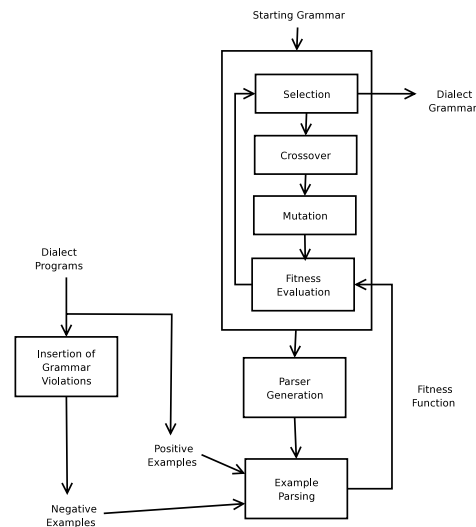


Figure 1. Grammar evolution using GA.

evolve. Although grammar partitioning may require some effort, whenever feasible it will greatly reduce the search space and improve the performance of the grammar inference task.

3. After, everything is ready to perform *automatic* grammar evolution through GA (see Figure 1). The fitness function is driven by parsing both *positive* and *negative* examples.
4. Then, the grammar partitions should be (*manually*) integrated.
5. Finally, the obtained grammar should be (*automatically*) tested over a conspicuous number of large source files.

The remainder of this section describes the approach details in terms of genome definition, GA operators and fitness function.

The genome needs to encode the production rules of a context-free grammar. Given a grammar  $G \equiv \{N, \Sigma, R, S\}$ , where  $N$  is a set of non-terminal symbols,  $\Sigma$  is a set of terminal symbols,  $R$  a set of production rules ( $R \subseteq N \times (N \times \Sigma^*)$ ), and  $S$  the root non-terminal. The most widely adopted approach [13] encodes each rule as an array of symbols, where the first item represents the Left-Hand-Side (LHS), and the remaining represent the Right-Hand-Side (RHS). A special symbol  $\epsilon$  denotes the end of the RHS and is also used to fill all the remaining empty slots. An example of simple context-free grammar (in the Bakus-Naur Form (BNF) [2]) is shown in Figure 2.

A *two-points* crossover [9] is used. It takes two parent genomes, breaks vertically in three pieces their RHS, and

Terminals		Non-Terminals	
0	a	3	A
1	b	4	B
2	c	5	ε

	LHS	RHS		
A → bA	3	1	3	5
B → ABa	4	3	4	0
B → c	5	1	5	5

**Figure 2. Genome encoding for a context-free grammar.**

Evolved-Grammar	
declaration_list	: declaration
	declaration_list declaration
declaration	: type_specifier init_declarator_list SEMI_COLN
	expression SEMI_COLN
statement_list	: statement
	statement_list statement
statement	: compound_statement
	declaration   expression SEMI_COLN
expression	: assignment_expression
	expression COMMA assignment_expression

**Table 1. Evolved C grammar with GNU declarations.**

mixes them to generate the offspring. We chose a crossover probability of 0.7. The mutation, performed with probability  $1/m$  (where  $m$  is the genome length), can work in two possible modes: i) it randomly selects a symbol in a rule, and replaces it with another (random) symbol (clearly, the LHS must be a non-terminal symbol); or ii) it randomly adds or removes a rule to/from the grammar.

As previously mentioned, we propose that, whenever available, an existing grammar (the *starting grammar*) is used as initial population, and that the complete grammar for the *target language* can be derived by evolving that grammar. Furthermore, the fitness function should penalize attempts to excessively disrupt the initial grammar, limiting the changes to what strictly necessary. The aforementioned approach can be applied i) when we need to parse a dialect; ii) when we have produced an incomplete grammar for a very complex language or iii) when we want to produce a grammar from one of a similar language (e.g., from C to PHP).

Our fitness function stems from what proposed by Javed et al. [10]. The basic idea behind grammar inference is to evolve a grammar so that it is able to parse the largest possible percentage (*cor\_positive*) of *positive examples* i.e., correct examples of the target languages. Moreover, it is also necessary that the grammar must be able to reject a reasonable percentage (*cor\_negative*) of *negative examples*, i.e., of programs containing syntactical errors. Negative examples are necessary, otherwise the GA could lead to a (useless) grammar that parses anything [8]. However, a grammar able to parse the 90% of positive examples source code lines, while not able to accept any complete source file, can be penalized despite being close to the target grammar. To this purpose, a third factor (*p\_total*) drives the fitness, accounting for the percentage of correctly parsed source lines.

Finally, since our approach aims to evolve an existing grammar, instead of creating a new one from scratch, the similarity between the obtained grammar and the starting one (*sim*) need to be kept sufficiently high. Overall, the obtained fitness function is defined as follows:

$$F = w_1 * (w_2 * cor\_positive + p\_total * w_3) * cor\_negative + w_4 * sim \quad (1)$$

where  $w_{1...4}$  are real, positive weighting factors.  $w_2$  and  $w_3$  are chosen such that  $w_2 * cor\_positive + p\_total * w_3$  tends to 1 when all the *positive examples* have been correctly parsed, while  $w_1$  and  $w_4$  are used to calibrate the influence of the similarity to the initial grammar. In our case studies  $w_{1...4}$  were set to 0.75, 0.75, 0.25 and 0.25 respectively. To measure the fitness of each individual, a *Yacc* parser is automatically generated from the genome and used against all positive and negative examples.

The convergence fastness, as well as the usability of the obtained grammar, strongly depends on the examples used in the inference. If we need to evolve a C grammar because something is changed in the variable declarations, we should produce examples focused to that section, and let the GA to quickly “learn” the differences. The examples should contain a few other (different) statements to ensure that the obtained grammar will still be able to parse a complete source file.

When selecting (or producing) negative examples, two different cases should be considered: i) the target language is backward-compatible with the source language; or ii) the target language is a subset of the source language. In the first case, it is possible to use mutated versions of the positive examples, containing syntax error uniformly distributed, with respect to both statements and type of errors. In the second case, producing negative examples is simpler, since they can just be starting language programs, that are not correct for the target language.

Evolved-Grammar	
function_definition	: type_specifier declarator compound_statement
declarator	: IDENTIFIER L_SBR R_SBR
declaration	: type_specifier init_declarator_list SEMI_COLN
init_declarator_list	: init_declarator
init_declarator	: init_declarator_list COMMA init_declarator
assignment_expression	: VARID
assignment_expression	: VARID QUALS assignment_expression
assignment_expression	: VARID QUALS assignment_expression
assignment_expression	: additive_expression

**Table 2. Inference of PHP variable declarations and uses**

## 4. Case Studies

To perform a preliminary assessment of the approach's effectiveness, we applied it to the ANSI C grammar (prior to the ANSI 99 standard), containing more than 200 production rules.

First, the grammar was evolved to support some GNU extensions, such as nested functions and the possibility of mixing code and declarations. To this aim, the grammar was properly partitioned. It was quite easy to extract a partition of 35 production rules dealing with declarations and with the general statement structure. Then, 110 generations were needed to produce a result in agreement with our requirements (i.e., parse all the positive code). An excerpt of obtained grammar is reported in Table 1 (differences from the starting grammars are highlighted with boxes). Some decisions (e.g., a declaration can be an expression) appear as counterintuitive; however the obtained grammar is rather usable.

The second experiment can be thought of as a necessary step to evolve a C grammar towards a PHP grammar. PHP uses the same C convention for functions names, while identifiers beginning with a "\$" for all the other variables (except a few other cases). To this aim, first it was necessary to add the new token (VARID) in the lexer. Then, we let the grammar evolve. The grammar excerpt in Table 2 shows how changes were automatically made when needed while, for instance, the *function declarator* correctly remained unchanged.

## 5. Conclusions and work-in-progress

This paper described how genetic algorithms can be used to evolve an existing grammar towards another, similar

one. While attempts made in the past to infer a complete general-purpose language grammar failed, we experienced that grammar evolution can be, at least, a promising tool in a semi-automatic grammar recovery process. Larger case studies and cost-benefit analysis are in-progress to better assess the benefits and limitations of the approach.

## References

- [1] *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*. Springer-Verlag, 1996.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles Techniques and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1985.
- [3] G. Antoniol, M. Di Penta, G. Masone, and U. Villano. Compiler hacking for source code analysis. *Software Quality Journal*, (12):383–406, 2004.
- [4] W. Cyre. Evolutionary language acquisition. In *IASTED International Conference on Artificial Intelligence & Soft Computing*, pages 146–151, Banff, Canada, July 2002.
- [5] W. Cyre. Learning grammars with a modified classifier system. In *Proc. 2002 World Congress on Computational Intelligence*, pages 1366–1371, Honolulu, Hawaii, USA, May 2002.
- [6] C. De La Higuera. Current trends in grammatical inference. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*, Aug 2000.
- [7] G. Dulewicz and O. Unold. Evolving natural language parser with genetic programming. *Abraham, A., Koppen, M. (eds.) Advances in Soft Computing. Hybrid Information Systems*, pages 361–377, 2002.
- [8] P. Dupont. Inference from positive and negative samples by genetic search: the GIG method. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 21–23, Sep 1994.
- [9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.
- [10] F. Javed, B. Bryant, M. Crepinsek, M. Mernik, and A. Sprague. Context-free grammar induction using genetic programming. In *ACMSE '04*, Huntsville, Alabama, USA, Apr 2004.
- [11] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, pages 78–88, Nov-Dec 2001.
- [12] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software - Practice and Experience*, 12(1), 2001.
- [13] M. Lankhorst. *Genetic algorithms in data analysis*. University Library Groningen, 1996.
- [14] S. Lucas. Context-free grammar evolution. In *First International Conference on Evolutionary Computing*, pages 130–135, 1994.
- [15] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 13–22, Oct 2001.
- [16] P. Wyard. Context free grammar induction using genetic algorithms. In *In: Belew, R.K. Booker, L.B. (eds.) Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 514–518, San Diego, CA, USA, 1991.