ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2011, Vol. 40, No. 4

EMBEDDING DSLS INTO GPLS: A GRAMMATICAL INFERENCE APPROACH *

Dejan Hrnčič
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor, Slovenia
e-mail: dejan.hrncic@uni-mb.si

Marjan Mernik
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor, Slovenia
e-mail: marjan.mernik@uni-mb.si

Barrett R. Bryant
The University of Alabama at Birmingham, Department of Computer and Information Sciences
Birmingham, AL 35294-1170, U.S.A.
e-mail: bryant@cis.uab.edu

crossref http://dx.doi.org/10.5755/j01.itc.40.4.980

Abstract. Embedding of Domain-Specific Languages (DSLs) into General-Purpose Languages (GPLs) is often used to express domain-specific problems using the domain's natural syntax inside GPL programs. It speeds up the development process, programs are more self-explanatory and repeating tasks are easier to handle. End-users or domain experts know what the desired language syntax would look like, but do not know how to write a grammar and language processing tools. Grammatical inference can be used for grammar extraction from input examples. A memetic algorithm for grammatical inference, named MAGIc, was implemented to extract grammar from DSL examples. In this work MAGIc is extended with embedding the inferred DSL into existing GPL grammar. Additionally, negative examples were also incorporated into the inference process. From the results it can be concluded that MAGIc is successful for DSL embedding and that the inference process is improved with use of negative examples.

Keywords: memetic algorithms, domain-specific languages, grammatical inference, embedding.

1. Introduction

Computer languages are a programmer's most basic tools [1]. With general-purpose languages (GPLs) we can address large classes of problems (e.g., scientific computing, business processing, symbolic processing, etc.), while domain-specific languages (DSLs) target one specific domain and their syntax and semantics is close to the domain's notation (e.g., aerospace, automotive, graphics, etc.). DSLs increase the productivity of developers and can also be used by end users [2, 3]. To speed up repeating tasks and to reduce the error rate inside written programs it is also useful to use syntax close to the problem domain [4]. This leads to language extensions and DSL embedding.

The term DSL embedding has been commonly used in the DSL community for two purposes and it is a source of much confusion. Firstly, DSL embedding has been used as a synonym for a DSL design phase where the DSL reuses already existing language constructs by extension, specialization, or the piggyback approach. Secondly, DSL embedding has been used as a synonym for a particular DSL implementation approach where the DSL is implemented by using the host language feature of function composition. This is embedding in a pure sense first introduced by Hudak [5]. To distinguish these two cases Mernik et al. [3] proposed for the DSL design phase two patterns: language exploitation (DSL design is based on an already existing language) and language invention (DSL design bears no relationship to an existing language), while embedding is only one out of many different implementation approaches (compiler/interpreter, application generator, preprocessing, embedding, extensible compiler/interpreter, COTS, and hybrid). Fowler [6] proposes classification into internal and external DSLs. On the other hand, Tratt [7] introduces new terms heterogeneous embedding and homogeneous embedding. The heterogeneous embedding can be implemented using hard

*This work was supported in part by United States National Science Foundation award CCF-0811630 and by Slovenian Research Agency award BI-US/11-12-031.

coded compilers/interpreters and preprocessing (e.g., source-to-source transformations) [7]. It is a suitable implementation approach when the DSL reuses an already existing language by extension, specialization, or piggyback. In other words, in the heterogeneous approach the programs written in a GPL and embedded DSL are first checked by a tool that extracts the DSL part and transforms it into the GPL code (e.g., SQLJ [8]), which is after the preprocessing normally compiled using the GPL compiler. The main advantage of heterogeneously embedded DSL is that the DSL syntax is close to the concrete syntax of the domain, which is especially important for the end user's productivity [9]. But such tools (e.g., compiler/interpreter, preprocessor) are usually implemented from scratch. Such an implementation of a DSL requires a formalization of its complete syntax. Homogeneously embedded DSLs can be implemented using macros, function composition, and libraries that provide domain-specific constructs [7]. A homogeneous embedded DSL is developed in a programming language, called a host language, into which it is embedded. The advantage of this approach is that no preprocessing tool is required. The main weakness of the homogeneous approach is that the DSL's concrete syntax is typically not close to the concrete syntax of the domain.

End users often do not know how to develop a new language, how to extend an existing programming language or even how to write a grammar for the desired language syntax. To fill this gap, grammatical inference (GI) algorithms can be used. Many papers regarding use of GI in the field of programming languages exist [10–12]. Despite many preliminary studies that have been performed in this field, the results are still not satisfactory.

GI is a process of building a model from examples of some structures or with interaction of the learner by asking questions to an oracle [13]. By using examples of structures we distinguish between learning from text, where only positive examples are used and between learning from an informant, where positive and negative examples are given as input. The process of giving queries to an oracle is called active learning. The obtained models can be then used for recognizing, interpreting, generating or transducing data structures. GI attracts researchers from different fields (e.g., machine learning, formal language theory, structural and syntactic pattern recognition, computational linguistics, computational biology and speech recognition).

In software engineering, GI deals with a problem of finding the grammar from programs written in an unknown language. These programs can be written in some legacy programming language, where the specifications are lost and/or tools like the parser are missing. With GI the structural information from such programs can be restored and used for building tools for parsing and conversion to newer programming languages. On the other hand, GI can be used by domain experts or end users who have little or no knowledge about language development. They can write the desired programs and with the use of the GI approach the programming language tools can be automatically built.

We have developed a GI algorithm called MAGIc. It is a memetic algorithm, which uses evolutionary algorithm concepts combined with a local search technique. Evolutionary algorithms [14] are useful for solving realistic problems because they are robust, give optimal or semi-optimal solutions, and can easily be adapted for different problems [15–17]. MAGIc was developed for inference of DSLs, but can be extended also for DSL embedding, which is presented in this paper. In the previous version of our algorithm only positive examples were used. But some of the generated grammars can be overgeneralized, therefore in this paper an improved MAGIc is presented, that uses also negative examples to avoid overgeneralization.

The structure of this paper is as follows: in Section 2 we present an overview of DSL embedding and use of GI in software engineering. Section 3 describes the core of MAGIc and the complexity and performance of the algorithm. The extensions with DSL embedding and use of negative examples are presented in Section 4. The results are presented in Section 5. The paper is concluded with Section 6 where the brief overview and word about future work is presented.

2. Related work

Several approaches were developed to embed DSLs into GPLs. Dinkelaker et al. [18] used island grammars [19], where the developer needs to specify grammar parts of the host languages that are relevant to the embedded DSL. Only those parts of the embedded DSL syntax need to be specified that are not compatible with the syntax of the host language. Concrete syntax is defined using special annotations inside the code. With special preprocessing, the abstract syntax tree is converted into the syntax of the host language. Renggli et al. [20] developed the HELVETIA tool, which is a language workbench tool for defining embedded languages and for interpreting them into the host language. It offers also debug support for embedded DSL. Again, the complete syntax structure of the

DSL has to be known. Knoll and Mezini [21] developed a new language, based on patterns, that has the ability to extend itself semantically and syntactically. The embedding of DSL can be made only within the pattern language itself, and the development cost of such an approach is very high.

All approaches mentioned above have in common that an experienced developer is needed to incorporate the DSL syntax into the host language. In the case of end users, who have little or no knowledge about programming, this is hard to accomplish. A tool for automatic extension of the host language and to define syntax structure/grammar is needed. Here grammatical inference comes in place. Imada and Nakamura [11] presented an incremental inductive CYK algorithm, that uses positive and negative examples to infer ambiguous and unambiguous context free grammars (CFGs). Unlike MAGIc, the input example order influences the grammars inferred. Kraft et al. [22] recovered grammars from hard-coded parsers, where parse trees generated by parsing input examples are used to recover a grammar. DSL embedding cannot be made using this approach, because the grammar can be inferred using only existing and working parsers. A work dealing with grammar versioning is presented by Dubey et al. [10] where partially correct grammars or grammar dialects were used to handle grammar versions. An iterative technique with backtracking is introduced. New rules are generated using existing non-terminals as left-hand side (LHS) and right-hand side (RHS) are generated using output of the CYK algorithm. Their algorithm convergence is dependent on the input example order. The results of grammatical inference in programming languages are still premature and a need for a robust and efficient algorithm exists.

3. MAGIc

Memetic **A**lgorithm for **G**rammatical **I**nferen**c**e (MAGIc) is a memetic algorithm [23], which is a population-based evolutionary algorithm with local search. The combination of an evolutionary approach with local search techniques is often used because of better results. MAGIc is implemented to infer from positive examples CFGs, which are non-ambiguous and of type LR(1). Gold's theorem [24] states that inference of language grammar cannot be done from positive examples only without additional knowledge. Therefore differences between input examples are used in MAGIc. The initial population of grammars is not generated randomly, as this was proven to be insufficient for grammar inference [25], but is generated from input examples. By generating the initial grammars the Sequitur algorithm [26] was used. It detects repetition in an example and factors it out by forming grammar rules. Note, that each initial grammar parses only one input example.

The main MAGIc steps are local search, mutation, generalization, and selection. They are presented in Figure 1 and in more details discussed in the following subsections. It is important to notice, that MAGIc is not a typical genetic algorithm as it does not include a crossover step, where two parents are selected and offspring are produced from parts of both parents.

3.1. Local Search

Individuals in the population are grammars that parse at least one input example. The idea of the local search operator is to incrementally change the grammar in a way to parse more input examples.

The local search operator changes the selected grammar using only positive examples. To successfully change the grammar, a local search method needs one parsed (true positive) example, one not parsed (false negative) example and the difference between them. The difference between examples represents part(s) that need to be inserted or made optional in the grammar. Comparison of the examples has to be done at the token level, not at the character level. Therefore the lexical analysis phase [27] needs to be inserted at the beginning of the algorithm. To determine the difference between examples the *diff* command [28] is used. It returns three types of differences: ADD, REPLACE and DELETE. MAGIc is implemented to exploit those differences for changing the grammars. When the parts (tokens) that need to be added or made optional are determined, the location inside the grammar where to make the change needs to be identified. Here the LR(1) parser is used. When parsing the false negative example, the parser encounters an error and information about the parser stack and the LR(1) item set [27] is returned. This information is used to determine the location where to change the grammar. Successful change of the grammar based on the type of difference is done following the rules explained in [29].

3.2. Mutation

Mutation is used to make random changes in the grammar. Grammar symbols (nonterminals or terminals) can be made optional or iterative.

The frequency of mutation is dependent on the algorithm input parameter p_m, the probability of mutation. On every grammar symbol three types of change can be made: option, iteration$^+$, iteration*. For example, the changes of mutation on the production `Nx ::=` α_1 `Ny` α_2, where grammar symbol `Ny` is selected for mutation, can be:

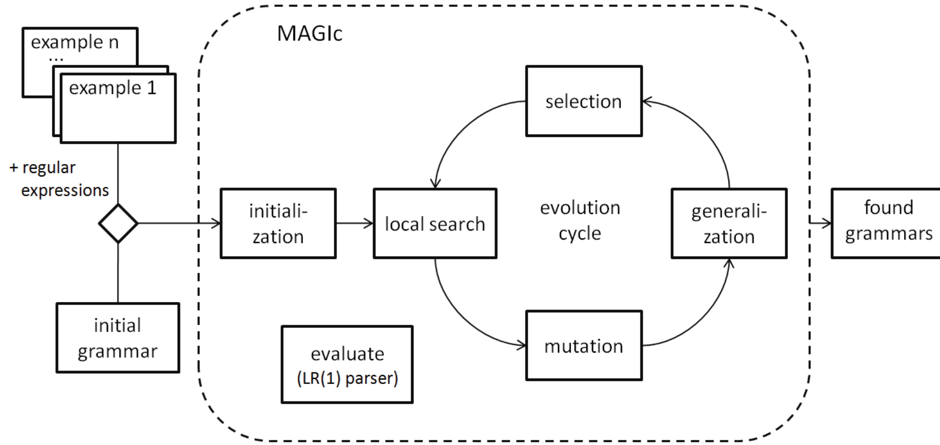

Figure 1. The Memetic Algorithm for Grammatical Inference

`Nx::=`α_1 `Nz` α_2		
Option	Iteration$^+$	Iteration*
`Nz::=Ny` `Nz::=`ε	`Nz::=Ny Nz` `Nz::=Ny`	`Nz::=Ny Nz` `Nz::=`ε

3.3. Generalization

Generalization is one of the most important parts of the algorithm. It checks for every grammar if it contains repetition or nested structures. Symbols `Nx`, `Ny` and `Nz` represent nonterminals, while α, β, $\gamma \in (N \cup T)^*$.

Nested structures. All productions of the form

`Nx ::= Ny Ny` α `Nz Nz`

are generalized and replaced with productions

`Nx ::= Ny Nx Nz`
`Nx ::=` α

where `Ny` and `Nz` are grammar symbols (nonterminals or terminals).

Repeating symbols. If a sequence of repeating grammar symbols is found, it can represent iteration. For example, consider a sequence of repeating nonterminals:

`Nx ::=` α `Ny Ny` β
`Ny ::=` γ

Repeating nonterminals are replaced with iteration*:

`Nx ::=` α `Ny` β
`Ny ::=` γ `Ny`
`Ny ::=` ε

and iteration$^+$:

`Nx ::=` α `Ny` β
`Ny ::=` γ `Ny`
`Ny ::=` γ

Repeating RHS. If in some production the RHS of another production appears, it is replaced with the LHS of that production. In the following example, the β part of the `Nx` production represents also the RHS of the `Ny` production.

`Nx ::=` α β γ
`Ny ::=` β
`Ny ::=` δ

Therefore the grammar production `Nx` can be generalized to:

`Nx ::=` α `Ny` γ
`Ny ::=` β
`Ny ::=` δ

3.4. Selection

MAGIc's evolution cycle (Figure 1) ends with the selection step. Grammars generated in steps before selection are inserted into the current population and do not replace their parents. Therefore in each evolution cycle the size of the population grows and is not limited. The task of the final selection step is to choose the best *pop_size* grammars from the current population and generate a new population for the next generation. Currently the selection process is deterministic, grammars are ranked based on the following fitness function:

$$\phi^+ = \sum_{i=1}^{N} isParsed(i) \tag{1}$$

where N represents the number of input examples, $isParsed(i)$ function returns 1 if the i-th example can be parsed with the current grammar or 0 if not, and the function ϕ^+ sums all successfully parsed examples. The best *pop_size* number of grammars is selected and a new population is generated.

3.5. Complexity and performance

The MAGIc computational time is dependent not only on the number of input examples, but also on different algorithm parameters (population size, number of generations, and probability of mutation). The overall (worst case) complexity is explained in the next few lines, which refer to Figure 1.

The complexity of the *Initialization* phase is linear $O(N)$, because for each input example one initial grammar is generated using Sequitur [26]. After algorithm initialization, the evolutionary cycle begins with num_gen generations. In the local search step the algorithm has two *for* loops, one is for improving the grammars with pop_size steps and one for the evaluation of generated grammars with $N * \Delta pop_size$ steps. The total approximate number of steps in this phase is $(1 + \Delta)pop_size * N$. Note, that $pop_size + \Delta pop_size$ is abbreviated as $(1+\Delta)pop_size$. The *Mutation* phase has two nested *for* loops and $((1+\Delta)pop_size)*(size(G)*p_m)*N$ steps, where $size(G)$ is the number of nonterminals and terminals in grammar and p_m represents probability of mutation. The next two steps of the algorithm are *Generalization* with $((1+\Delta)pop_size)*N$ steps and *Selection* with pop_size steps (best pop_size grammars are selected into the next generation). The overall approximate number of steps in the algorithm is

$$\begin{aligned} & N + num_gen * (((1 + \Delta)pop_size * N)+ \\ & (((1 + \Delta)pop_size) * (size(G) * p_m) * N)+ \\ & (((1 + \Delta)pop_size) * N) + pop_size) \end{aligned} \quad (2)$$

which means that the complexity of the algorithm, based on the number of input examples is linear $\approx O(N)$. Due to this fact, increasing the number of input examples which are often needed to successfully infer the correct grammar does not significantly impact the MAGIc computation time.

To demonstrate the performance of the algorithm, a simple DSL named DESK [30] was used. DESK is a simple desk calculation language with statements of the form `PRINT <expression> WHERE <definitions>`. The average processing time for 12 DESK language examples with probability of mutation (p_m) 0.01, population size (pop_size) 40 and number of generations (num_gen) 30 was about 30 seconds [29]. For the DESK language, a parameter tuning was made, where p_m was [1%, 2% and 5%], pop_size was [20, 30, 40 and 50] and num_gen was [30, 50 and 70]. For each of the 36 parameter sets, 30 algorithm runs were made and based on the results the optimal parameter values were chosen. The results of parameter tuning are available in [29]. For experimental runs, we have used an Intel Core 2 Duo P8600 processor with 2.4 GHz.

To infer the correct grammar, the differences between input examples are also important. They have to be small and explicit enough that the *diff* algorithm detects the correct difference among examples. All 12 positive input examples for the DESK language are presented in [29]. The structure of negative examples is close to that of positive examples, emphasizing the part of the syntax that we do not want to be generalized.

The number of needed input examples depends on the size of the inferred language. The main rule for choosing the input examples is that they have to express each language construct.

4. MAGIc extensions

4.1. DSL embedding

As discussed in Section 1, using an embedded DSL is useful for several reasons. Development is easier, quicker and programs are easier to read, especially by end users. By embedding DSL into the host language, the developer has to deal with problems such as how the integration of DSL syntax is made into the existing GPL grammar and how to enable the DSL to use the domain's commonly used notation/syntax. MAGIc is useful for solving this problem, because the developer does not need to deal with either where to embed the productions of the DSL language into the existing grammar or with defining the formal syntax of embedded DSL. By looking into the parser stack trace, MAGIc determines the possible positions where to insert the DSL productions into the existing grammar. By looking into differences between input DSL examples, the structure of newly inserted productions is determined.

To determine the possible positions where to insert new productions, MAGIc uses the approach described in Section 3, which looks into the parser stack when an error in parsing the input example occurs. To define the structure of new grammar productions, MAGIc's *local search* and *generalization* methods are used. Method *mutation* may help to identify some DSL language concepts like iteration or option by mutating grammar elements (see Section 3). The extension made in MAGIc to make embedding of DSL

into GPL possible was the ability to begin the inference process from an already existing GPL grammar and not generated from input examples. The mutation and generalization steps have to leave initial GPL productions intact. This was accomplished by locking all the productions in the initial grammar. Locking the productions in the grammar means that no change to productions can be made, except in the *local search* step, where new nonterminals are inserted into current productions.

4.2. Negative examples

In the latest version of MAGIc negative examples were also included into the inference process. As mentioned before, MAGIc is able to infer correct grammar from positive examples alone, but in the inference process some grammars can be overgeneralized.

In the MAGIc extension, input examples are divided into positive and negative sets. The examples from the negative set have an impact on the fitness function of generated grammars, which is changed to:

$$\phi = \phi^{+} * ((\sum_{i=1}^{M} isParsed(i) > 0)?\ 0 : 1) \qquad (3)$$

where ϕ^{+} represents the old fitness and M represents the number of negative examples. The function sums all successfully parsed examples, but if at least 1 true negative example is successfully parsed, the output of the fitness function is 0. A grammar is correct, if it parses all true positive examples and rejects all true negative examples that were given as input to the algorithm.

As for now, there is no repair algorithm incorporated into MAGIc. Grammars that successfully parse negative examples are simply discarded. For further work we have an idea of using negative examples also for repairing inferred grammars. In Section 5 the influence of negative examples on MAGIc results is discussed and shown on inferred grammars.

5. Results

5.1. DSL embedding

To test the applicability of using MAGIc for DSL embedding, the ANSI C grammar [31] with approximately 200 productions was used. In Figure 2 only some productions are shown. These productions are important to show how MAGIc infers the new grammar with DSL syntax embedded. The DSL used is TinySQL and was used also for DSL embedding in [18]. Dinkelaker et al. have demonstrated the embedding into Java and Groovy code with the use of island grammars. The concrete DSL syntax is defined using special method annotation. From meta-data provided in those annotations the grammar of the embedded DSL is extracted and combined with the grammar of the host language. To combine the DSL grammar with the host language grammar, the language developer needs to specify those parts of the host language's syntax that are relevant when embedding DSL. The positions in the host language grammar where to insert new productions are determined automatically by MAGIc.

The first 207 productions are original ANSI C grammar productions. They are locked to prevent MAGIc methods from altering them, except for the local search method. Local search can insert new nonterminals into the existing grammar to define the connection between the host language and the embedded DSL.

To demonstrate the embedding of TinySQL into the ANSI C grammar consider the next three input examples:
- true positive example:

```
int main() {
   char str[][];
   int i;
   printf("Students:");
   for(i = 0; i < str.length; i++) {
      printf(str[i]);
   }
   return 0;
}
```

- two false negative examples:

```
int main() {
   char str[][] = { SELECT Name FROM
                         Students };
   int i;
   printf("Students:");
   for(i = 0; i < str.length; i++) {
      printf(str[i]);
   }
   return 0;
}
```

```
int main() {
   char str[][] = { SELECT Name, Surname
        FROM Students, Professors };
   int i;
   printf("Students and Professors:");
   for(i = 0; i < str.length; i++) {
      printf(str[i]);
   }
   return 0;
}
```

1. **translation_unit** ::= **external_decl**
2. **translation_unit** ::= **translation_unit external_decl**
3. **external_decl** ::= **function_definition**
4. **external_decl** ::= **decl**
6. **function_definition** ::= **declarator decl_list compound_stat**
9. **decl** ::= **decl_specs init_declarator_list** ;
10. **decl** ::= **decl_specs** ;
11. **decl_list** ::= **decl**
12. **decl_list** ::= **decl_list decl**
15. **decl_specs** ::= **type_spec decl_specs**
27. **type_spec** ::= int | long | ...
45. **init_declarator_list** ::= **init_declarator**
46. **init_declarator_list** ::= **init_declarator_list** , **init_declarator**
47. **init_declarator** ::= **declarator**
64. **enumerator** ::= id
65. **enumerator** ::= id = **const_exp**
67. **declarator** ::= **direct_declarator** ● ***NT1***
68. **direct_declarator** ::= id
69. **direct_declarator** ::= (**declarator**)
70. **direct_declarator** ::= **direct_declarator** ● [**const_exp**]
71. **direct_declarator** ::= **direct_declarator** [ ] ●
72. **direct_declarator** ::= **direct_declarator** ● (**param_type_list**)
73. **direct_declarator** ::= **direct_declarator** ● (**id_list**)
74. **direct_declarator** ::= **direct_declarator** ● ( )
88. **id_list** ::= id
89. **id_list** ::= **id_list** , id
90. **initializer** ::= **assignment_exp**
91. **initializer** ::= **initializer_list**
93. **initializer_list** ::= **initializer**
94. **initializer_list** ::= **initializer_list** , **initializer**
110. **stat** ::= **labeled_stat** | **exp_stat** | **compound_stat** | **selection_stat**
114. **stat** ::= **iteration_stat** | **jump_stat**
116. **labeled_stat** ::= id : **stat**
117. **labeled_stat** ::= case **const_exp** : **stat**
118. **labeled_stat** ::= default : **stat**
119. **exp_stat** ::= **exp** ;
120. **exp_stat** ::= ;
121. **compound_stat** ::= **decl_list stat_list**
125. **stat_list** ::= **stat**
126. **stat_list** ::= **stat_list stat**
127. **selection_stat** ::= if (**exp**) **stat**
129. **selection_stat** ::= switch (**exp**) **stat**
130. **iteration_stat** ::= while (**exp**) **stat**
131. **iteration_stat** ::= do **stat** while (**exp**) ;
132. **iteration_stat** ::= for (**exp** ; **exp** ; **exp**) **stat**
140. **jump_stat** ::= goto id ; | continue ; | break ; | return **exp** ;
145. **exp** ::= **assignment_exp**
146. **exp** ::= **exp** , **assignment_exp**
147. **assignment_exp** ::= **conditional_exp**
148. **assignment_exp** ::= **conditional_exp assignment_operator assignment_exp**
205. **const** ::= int_const | char_const | float_const
208. ***NT1*** ::= = SELECT id **NT2** FROM id **NT2** | ε
210. ***NT2*** ::= , id **NT2** | ε

Figure 2. Excerpt of ANSI C grammar, extended with inferred DSL productions

Comparing the true positive example with first false negative example, the difference *diff* is:

```
= { SELECT Name FROM Students }
```

The positions where to insert new production

```
NT1 ::= = { SELECT Name FROM Students }
```

are returned from the LR(1) parser, when parsing the first false negative example and can be seen in Figure 2 marked with symbol ●. These positions can be used to insert new nonterminals and extend the current grammar with new rules of DSL.

The repetition of grammar symbols that can be seen in productions 208-210 is made when both false negative examples are incorporated into the grammar and by using the *generalization* method, which is described in subsection 3.3 and in detail in [29].

5.2. Negative examples

The generalization step can sometimes overgeneralize grammars and the resulting grammar can parse also true negative examples. It is important to identify such grammars and eliminate them to prevent them to advance to the next generation. Since the repair from true negative examples is under implementation, the grammars that parse at least one true negative example are simply discarded. As an example, consider the previous example of the embedded TinySQL language. Using the generalization step on the inferred grammar (Figure 2) the following change to productions 208-211 of the grammar is obtained:

```
208. NT1 ::= = { SELECT enumerator NT2
               FROM enumerator NT2 } | ε
210. NT2 ::= , enumerator NT2 | ε
```

The generalization step searches for the repeating RHS: Token `id` represents the RHS of the 64th production (**`enumerator`** `::= id`), therefore it is replaced with the LHS of that production (**`enumerator`**).

Using such an inferred grammar, due to production 65 the new grammar parses also negative example:

```
...{ SELECT Name = 3 FROM Students }...
```

Although this grammar parses all true positive examples it is not correct and is discarded.

6. Conclusions

This paper has presented an extension of MAGIc for embedding DSLs into GPLs. MAGIc is an incremental population-based algorithm comprised of initialization, local search, mutation, generalization and selection steps.

The extension to the algorithm was explained and tested on simple DSL TinySQL and GPL ANSI C. The results are similar to the original grammar of TinySQL used in [18]. The main advantage of our approach is that the grammar of the embedded DSL is inferred from input examples and the interaction between DSL grammar and host language grammar is determined automatically, hence the user does not have to be a language designer to be able to embed a DSL.

In the future we would like to further extend MAGIc to be able to use true negative examples also for repairing inferred grammars, which successfully parse true negative examples and are therefore discarded. We will also extend the selection process which needs to differentiate between neutral solutions. More extensive experimental work, which will include more DSLs, is also planned in the future.

References

[1] **C. A. R. Hoare.** Hints on programming language design. *Technical report, CS-TR-73-403, Stanford University, Stanford, CA, USA*, 1973.

[2] **T. Kosar, N. Oliveira, M. Mernik, M. J. Varanda Pereira, M. Črepinšek, D. da Cruz, P. R. Henriques.** Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. *Computer Science and Information Systems*, 2010, 7, 247–264.

[3] **M. Mernik, J. Heering, A. M. Sloane.** When and how to develop domain-specific languages. *ACM Computing Surveys*, 2005, 37(4), 316–344.

[4] **M. J. Varanda Pereira, M. Mernik, D. da Cruz, P. R. Henriques.** Program comprehension for domain-specific languages. *Computer Science and Information Systems*, 2008, 5(2), 1–17

[5] **P. Hudak.** Building domain-specific embedded languages. *ACM Computing Surveys*, 1996, 28.

[6] **M. Fowler.** Domain Specific Languages. *Addison-Wesley Professional*, 2010.

[7] **L. Tratt.** Domain Specific Language Implementation via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems*, 2008, 30(6), 1–40.

[8] **J. Melton, A. Eisenberg.** Understanding SQL and Java together: a guide to SQLJ, JDBC, and related technologies. *Morgan Kaufmann*, 2000.

[9] **T. Kosar, P. E. Martínez López, P. A. Barrientos, M. Mernik.** A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. *Information and Software Technology*, 2008, 50(5), 390–405.

[10] **A. Dubey, P. Jalote, S. K. Aggarwal.** Learning context-free grammar rules from a set of program. *IET Software*, 2008, 2(3), 223–240.

[11] **K. Imada, K. Nakamura.** Towards Machine Learning of Grammars and Compilers of Programming Languages. *In: Proceedings of European conference on Machine Learning and Knowledge Discovery in Databases - Part II, ECML PKDD '08, Springer-Verlag, Berlin, Heidelberg*, 2008, 98–112.

[12] **F. Javed, M. Mernik, B. R. Bryant, A. Sprague.** An Unsupervised Incremental Learning Algorithm for Domain-Specific Language Development. *Applied Artificial Intelligence*, 2008, 22(7-8), 707–729.

[13] **C. de la Higuera.** A bibliographical study of grammatical inference. *Pattern Recognition*, 2005, 38(9), 1332–1348.

[14] **Z. Michalewicz.** Genetic Algorithms + Data Structures = Evolution Programs, Third Edition. *Springer-Verlag, New York, NY, USA*, 1996.

[15] **E. Babkin, M. Petrova.** Application of Genetic Algorithms to Increase an Overall Performance of Neural Networks in the Domain of Database Structures Synthesis. *Information Technology and Control*, 2006, 35(3A), 285–294.

[16] **N. Goranin, A. Čenys.** Genetic Algorithm Based Internet Worm Propagation Strategy Modeling. *Information Technology and Control*, 2008, 37(2), 133–140.

[17] **M. Paulinas, A. Ušinskas.** A Survey of Genetic Algorithms Applications for Image Enhancement and Segmentation. *Information Technology and Control*, 2007, 36(3), 278–284.

[18] **T. Dinkelaker, M. Eichberg, M. Mezini.** Incremental Concrete Syntax for Embedded Languages. *In: Proc. of ACM SAC 2011, Taiwan*, 2011, 1309–1316.

[19] **L. Moonen.** Generating Robust Parsers using Island Grammars. *In: Proceedings of 8th Working Conference on Reverse Engineering, IEEE Computer Society Press*, 2001, 13–22.

[20] **L. Renggli, T. Girba, O. Nierstrasz.** Embedding Languages Without Breaking Tools. *In: Proceedings of 24th European Conference on Object-Oriented Programming (ECOOP 2010), Springer-Verlag*, 2010, 380–404.

[21] **R. Knöll, M. Mezini.** π - A Pattern Language. *ACM SIGPLAN Notices*, 2009, 44(10), 503–522.

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

[23] **P. Moscato.** On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. *Technical report, Concurrent Computation Program 158-79, California Institute of Technology, Pasadena, CA, USA*, 1989.

[24] **E. M. Gold.** Language Identification in the Limit. *Information and Control*, 1967, 10(5), 447–474.

[25] **M. Črepinšek, M. Mernik, V. Žumer.** Extracting grammar from programs: brute force approach. *ACM SIGPLAN Notices*, 2005, 40(4), 29–38.

[26] **C. G. Nevill-Manning, I. H. Witten.** Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 1997, 7, 67–82.

[27] **A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman.** Compilers: Principles, Techniques, and Tools, Second Edition. AddisonWesley, 2007.

[28] **J. W. Hunt, M. D. McIlroy.** An Algorithm for Differential File Comparison. *Technical report, CSTR 41, Bell Laboratories, Murray Hill, NJ*, 1976.

[29] **M. Mernik, D. Hrnčič, B. R. Bryant, F. Javed.** Applications of Grammatical Inference in Software Engineering: Domain Specific Language Development. *In: Carlos Martín-Vide (ed.), Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory - Vol. 2, Scientific Applications of Language Methods, Imperial College Press, London*, 2010, ch. 8, 421–457.

[30] **J. Paakki.** Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 1995, 27(2), 196–255.

[31] **J. Degener.** ANSI C Yacc grammar. *http://www.lysator.liu.se/c/ANSI-C-grammar-y.html*, 1995 (accessed October 29, 2010).

Received November 2010.