

Grammar Inference Algorithms and Applications in Software Engineering

Marjan Mernik, Dejan Hrnčič
Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
{marjan.mernik, dejan.hrnctic}@uni-mb.si

Barrett R. Bryant, Alan P. Sprague, Jeff Gray,
Qichao Liu, Faizan Javed
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, U.S.A
{bryant, sprague, gray, qichao, javedf}@cis.uab.edu

Abstract—Many problems exist whose solutions take the form of patterns that may be expressed using grammars (e.g., speech recognition, text processing, genetic sequencing). Construction of these grammars is usually carried out by computer scientists working with domain experts. In the case when there is a lack of domain experts, grammar inference can be applied. In this paper, two grammar inference algorithms are briefly described and their application to software engineering is presented.

Keywords—grammar learning; memetic algorithm; domain-specific language; metamodeling

I. INTRODUCTION

Grammatical inference (GI), often called grammar induction or language learning, is the process of learning a grammar from positive and/or negative sentence examples. Machine learning of grammars finds many applications in syntactic pattern recognition [1], computational biology [2], and natural language acquisition [3]. This paper discusses the application of GI to software engineering challenges. Software engineers usually want to recover a grammar from legacy systems in order to automatically generate various software analysis and modification tools. Often, the grammar can be semi-automatically recovered from compilers and language manuals [4]. However, grammars are often used as an efficient representation of artifacts that are inherently structural and/or recursive (e.g., neural networks, structured data and patterns). Semi-automatic grammar recovery [4] is not a feasible option in this case (compilers and language manuals do not exist) and grammars need to be extracted solely from artifacts represented as sentences/programs written in some unknown language.

One of the most important theoretical results in GI is Gold's theorem [5], which can be intuitively understood by recognizing the fact that using only positive samples could result in over-generalization (e.g., an automaton that accepts all strings). This implies the need for some restrictions or background knowledge on the generalization process. GI has been successful at inferring regular languages, where researchers have developed various algorithms (e.g., RPNI [6]) which can learn regular languages. The standard procedure is to construct the maximal canonical automaton from positive samples and generalize the automaton by using a state merging process (Fig. 1; originally presented in [7]). Merging states

results in generalization and an automaton is obtained that accepts a larger set of strings.

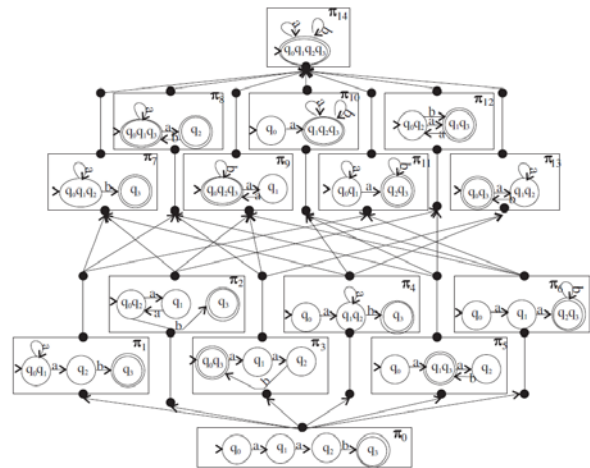


Figure 1. State merging process – lattice of automata

The search space of regular grammar inference depends on the total number of states in the maximal canonical automaton. Even for a modest number of states it is not practical to explicitly build the lattice (e.g., from 4 states 15 different automata can be obtained by merging states, while from 10 states the number of different automata is increased to 115,975). To overcome this problem, heuristic [8] or incremental methods [9] can be used.

The search space for context-free grammar (CFG) inference is even larger [7]. We need to find out how many different derivation trees exist for a given positive sentence. If we resort to Chomsky Normal Form (CNF), the number of all possible binary trees with n internal nodes is given by the n -th Catalan number [10], where internal nodes (nonterminals) need to be properly labeled, too. All possible labeling of nonterminals is defined by Bell numbers [10]. These together contribute to an immense search space. For example, a statement with 5 terminals (4 nonterminals) can be parsed by 210 different derivation trees, while this number increases to 1.9479161E9 for a statement with 11 terminals (10 nonterminals). Hence, a

different and more efficient approach to explore the search space is needed. Our GI approach applies an evolutionary algorithm, which is a search and optimization technique based on the principles and mechanisms of natural selection and survival of the fittest [11]. Evolutionary algorithms (e.g., genetic algorithms (GA), evolution strategies (ES), evolutionary programming (EP), and genetic programming (GP)) have been successfully used in a wide variety of combinatorial optimization problems as well as industrial problems [11]. However, many studies show evolutionary algorithms produce better results if they are augmented with hybrid techniques, such as local search and domain-specific knowledge [12].

The structure of this paper is as follows. In Section II, domain-specific language development using GI is introduced and our newly developed GI algorithm called *MAGic* is described. In Section III, another application of GI to metamodel recovery is briefly mentioned. Related work is discussed in Section IV. Finally, we conclude in Section V.

II. DOMAIN-SPECIFIC LANGUAGE DEVELOPMENT

In this paper we explore a new application of GI in the area of software engineering, in particular the process of Domain-Specific Language (DSL) development [13]. In contrast to general-purpose programming languages (GPLs), where one can address large classes of problems (e.g., scientific computing, business processing), a DSL facilitates the solution of problems in a particular domain (e.g., aerospace, automotive, graphics). GPLs have been usually designed with great care and complying to language design principles. Unfortunately, this is not the case for DSLs, where design tends to be done in a more ad hoc manner. We have explored the applicability of GI to infer a DSL grammar from DSL programs. Such a scenario would be feasible when domain experts can provide complete DSL programs or excerpts of such programs. This is the case when domain notation is already established and the notation decision pattern [13] can be used. The results of GI, namely the inferred grammar, can be directly used to generate the DSL parser or be further examined by a software language engineer to enhance the design of the language.

A. *MAGic*

A Memetic Algorithm for Grammatical Inference (*MAGic*) infers CFGs from positive samples only. At the beginning, an initial population of grammars is generated. For generating initial grammars we used the Sequitur algorithm [14] that detects repetition in a sample and factors it out by forming grammar rules. Note that Sequitur does not generalize productions. The generated grammar by Sequitur only parses a particular sample. After the initial population is built we try to improve each grammar by local search, which is later described in more detail. Afterward, grammars undergo transformation through mutation. The mutation exploits the knowledge of grammars; namely, when they are specified in extended Backus Naur Form (EBNF), where grammar symbols often appear optionally or iteratively (option operator, iteration+ operator, and iteration* operator). For each grammar symbol that appears on the right-hand side of a production, a random

value is generated. If this value is smaller than the probability of mutation (p_m) then the grammar symbol is mutated. Option, iteration+, or iteration* operators are chosen randomly. After mutation an important step of generalization is performed. Because our goal is not to over-generalize grammars, only simple rules are applied. Note also, that we keep original grammars in the intermediate population. New grammars which are found by a generalization process do not replace them; instead, the new grammars are added to the current population. In the first step of the generalization, productions where the right-hand side (RHS) of another production appears in that production are replaced with the left-hand side (LHS) symbol. The next step searches for a repeated sequence of symbols. If such a sequence is found there is a possibility for iteration of symbols. At the end of the evolutionary cycle each grammar is evaluated on all positive samples. The number of positive samples that are parsed is the grammar fitness [11], which is used in the selection process. Deterministic selection is used where all grammars are ranked and only the best N grammars are selected into the next population. Note that during the evolutionary cycle, the population of grammars is not fixed and can grow but at the end of the evolutionary cycle only N grammars survive. A typical population change during the evolutionary run is shown in Fig. 2. The algorithm runs for M generations, where M is an input parameter of the algorithm. From our previous experience on EAs [15], we are aware of the fact that results of EAs heavily depend on a suitable parameter control mechanism (e.g., adaptive or self-adaptive parameter control).

The most important part of *MAGic* is local search. The implemented algorithm borrows the ideas from *GenInc* [16], our previous GI approach for DSL development, where GI is done by comparing ordered positive samples. *GenInc* relies on characteristic samples that are presented in a strict order allowing encoding of variances in the samples. *MAGic* works by comparing two random samples and changes the grammar based on the differences between them. Hence, strict ordering of samples is no longer needed and robustness of the inference process is enhanced in that the language designer does not need to conform to a certain order when creating input samples. After the differences among samples are identified, the objective is to change the grammar in such a manner that both samples can be parsed.

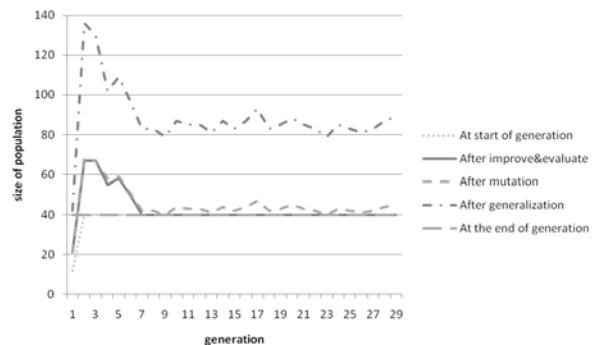


Figure 2. Population change during evolutionary run

In contrast to GenInc, MAGIc exploits the parsing history (an LR(1) parser is used). When the parser fails to parse the sample, information about the parser stack and the LR(1) item set are returned to the evaluation function. Each state of the LR(1) automaton is composed of LR(1) items [17]. Because the parser puts the processed states on the stack, it is also possible to examine all previous states. The change to the grammar is done based on the type of difference between samples. Those types can be ADD, REPLACE or DELETE. Due to page limitations, only one example of local search is given below.

Case ADD

Assume true positive sample, $s_1s_2\dots s_j$, is recognized but the false negative sample, $s_1s_2\dots s_ka_1\dots a_n s_{k+1}\dots s_j$, is not. The difference between them, $a_1\dots a_n$, is part of sample two. In order to successfully recognize the second sample, the difference has to be added into the grammar. By parsing the false negative sample, three types of errors can be returned by the parser:

1. (Case 1) The parser did not recognize any token from the difference and stopped at token a_i ; the difference that needs to be added is defined as $a_{i+1}\dots a_n$, where i is the number of recognized tokens from the difference; in this case $i = 0$
2. (Case 2) The parser recognized i tokens from the difference, where $i = 1\dots(n-1)$; the start of the difference that has to be added to the grammar is moved to the first symbol of the unrecognized difference a_{i+1}
3. (Case 3) The parser recognized the whole difference, but stopped at s_{k+1} ; $i = n$, hence $a_{i+1} = s_{k+1}$

(Case 1 and 2) The state's configurations on the LR(1) parser stack where an error occurs can be of the three forms below:

$$\begin{aligned} N_x &\rightarrow \alpha_1 \cdot \alpha_2 \\ N_y &\rightarrow \beta \cdot \\ N_z &\rightarrow \cdot \gamma \end{aligned}$$

If configuration $N_x \rightarrow \alpha_1 \cdot \alpha_2$ is returned, then it is checked if $s_{k+1} \in \text{FIRST}(\alpha_2)$ [17]. If true then the next change to the grammar is made, as shown below:

$$\begin{aligned} N_x &::= \alpha_1 N_1 \alpha_2 \\ N_1 &::= a_{i+1}\dots a_n \\ N_1 &::= \epsilon \end{aligned}$$

If $s_{k+1} \notin \text{FIRST}(\alpha_2)$, then it is checked if $s_{k+1} \in \text{FOLLOW}(N_x)$ [17]. If true, then an alternative to the last part of the production is made:

$$\begin{aligned} N_x &::= \alpha_1 N_1 \\ N_1 &::= \alpha_2 \\ N_1 &::= a_{i+1}\dots a_n \end{aligned}$$

If $s_{k+1} \notin \text{FIRST}(\alpha_2) \wedge s_{k+1} \notin \text{FOLLOW}(N_x)$ then the grammar cannot be changed at this configuration and the algorithm goes to the next configuration returned from the parser.

The next form of parser configuration, that can also be returned, is $N_y \rightarrow \beta \cdot$. In this case we check if $s_{k+1} \in \text{FOLLOW}(N_y)$. If this is true, then the next change to the grammar is made, as follows:

$$\begin{aligned} N_y &::= \beta N_1 \\ N_1 &::= a_{i+1}\dots a_n \\ N_1 &::= \epsilon \end{aligned}$$

If $s_{k+1} \notin \text{FOLLOW}(N_y)$ then change to the grammar at this configuration cannot be made and the next one is processed.

In case of configuration $N_z \rightarrow \cdot \gamma$ we first check if $s_{k+1} \in \text{FIRST}(\gamma)$. If this check is true then the grammar is changed to:

$$\begin{aligned} N_z &::= N_1 \gamma \\ N_1 &::= a_{i+1}\dots a_n \\ N_1 &::= \epsilon \end{aligned}$$

If $s_{k+1} \notin \text{FIRST}(\gamma) \wedge s_{k+1} \in \text{FOLLOW}(N_z)$ is true, then a new alternative to the grammar is added:

$$\begin{aligned} N_z &::= \gamma \\ N_z &::= a_{i+1}\dots a_n \end{aligned}$$

In case that $s_{k+1} \notin \text{FIRST}(\gamma) \wedge s_{k+1} \notin \text{FOLLOW}(N_z)$ then the grammar at this configuration cannot be changed and the next configuration returned from the stack is processed.

(Case 3) The parser recognizes the whole difference and returns an error at symbol s_{k+1} . Let us first examine the configuration $N_x \rightarrow \alpha_1 \cdot \alpha_2$. Here, $s_{k+1} \notin \text{FIRST}(\alpha_2)$, because if it were, the parser would recognize s_{k+1} . The algorithm checks if $s_{k+1} \in \text{FOLLOW}(N_x)$ and makes the following change to the grammar:

$$\begin{aligned} N_x &::= \alpha_1 N_1 \\ N_1 &::= \alpha_2 \\ N_1 &::= \epsilon \end{aligned}$$

If $s_{k+1} \notin \text{FIRST}(\alpha_2) \wedge s_{k+1} \notin \text{FOLLOW}(N_x)$ then the grammar at this configuration cannot be changed and the algorithm goes to the next configuration.

The next possible configuration is $N_y \rightarrow \beta \cdot$. Here the algorithm cannot change the grammar, because $s_{k+1} \notin \text{FOLLOW}(N_y)$ (otherwise the parser would recognize it).

The last possible configuration is $N_z \rightarrow \cdot \gamma$. First, $s_{k+1} \notin \text{FIRST}(\gamma)$, so the next check is if $s_{k+1} \in \text{FOLLOW}(N_z)$. If true, then an alternative of N_z production is made:

$$\begin{aligned} N_z &::= \gamma \\ N_z &::= \epsilon \end{aligned}$$

The REPLACE and DELETE type of differences are similar to the described ADD type.

We have tested MAGIc on several examples (e.g., DESK [16], FDL [16], WHILE [16], HyperTree [18]). MAGIc is able to infer CFGs, which are non-ambiguous and of type LR(1). We have also performed extensive control parameter tuning for

the DESK language where the following settings gave the best results: $p_m=0.01$, $pop_size=40$, $num_gen=30$. To find out which process (local search, mutation, generalization) is most crucial to MAGIc, we also performed a small experiment on the DESK language. Results are shown in Table 1, which confirms that local search is indeed the most important part of MAGIc, because without local search the solution was not found, but with this operator the solution was found in all 30 runs for each combination.

TABLE I. Influence of local search, mutation and generalization on MAGIc. SR – success rate; AES – average number of evaluations; GS – average number of generations to first solution; ANN – Average number of different nonterminals; ANP – average number of productions; ARHS – average size of right hand side in inferred grammars; ANS – average number of samples needed to solution.

local search	mutation	generalization	SR	AES	GS	ANN	ANP	ARHS	ANS
no	no	yes	0	n/a	n/a	n/a	n/a	n/a	n/a
no	yes	no	0	n/a	n/a	n/a	n/a	n/a	n/a
no	yes	yes	0	n/a	n/a	n/a	n/a	n/a	n/a
yes	no	no	1	3712.20	7.47	8.13	16.25	3.50	9.04
yes	no	yes	1	3731.03	4.63	7.48	14.27	3.39	8.07
yes	yes	no	1	4014.47	7.60	8.27	16.46	3.48	8.95
yes	yes	yes	1	4206.27	4.70	7.45	14.07	3.32	7.56

B. Case Study

The capability of using grammatical inference techniques in DSL design, in particular the MAGIc tool, has been tested on a real example from the computer graphics domain. In [18], Strnad and Guid developed a method for modeling trees with hypertextures, a method for describing three-dimensional shapes and textures. The method is based on a volumetric representation of trees generated by three-dimensional variation of an Iterated Function System (IFS), which is a technique for fractal generation. Using this method, a tree is a fractal object described by nonlinear and nondeterministic IFS where a combination of linear transformations (scalings, translations and rotations) and nonlinear shears are used. Scaling, translations, and rotations are used to describe fractal subparts (size, position, and orientation), while nonlinear shears are used to bend them in two coordinate directions. Nondeterminism of transformations is achieved by randomly chosen values of transformation parameters (e.g., angle of rotation). Random parameters allow the same set of transformations to produce visually different hypertrees with similar basic structure.

A hypertree consists of branches which are like smaller trees. Branching structure is generated by condensation transformation [18]. Moreover, the ideal structure of hypertrees can be distorted with noise perturbation. The whole description of a hypertree consists of resolution, number of iterations, fractal depth, number of branch levels, the description of the generator (POINTINIT or LINEINIT), the coloring scheme (DEPTHCOLOR) and transformations. In Fig. 3, an excerpt from a DSL program is given, while in Fig. 4 a generated hypertree is displayed. Strnad and Guid [18] have no experience with language engineering and they implemented the language from scratch. The parsing algorithm was hard coded and not based on a grammar. Because of a lack of knowledge in language engineering the underlying grammar

was not identified. If it were, the parsing code could be automatically generated by parser generators. On the other hand, maintenance of the parsing code is now extremely difficult if this language evolves in the future.

```

RESOLUTION 300 400 300
ITERATIONS 3000000
POINTINIT 0 0 0
TREEDEPTH 5
BRANCHDEPTH 1
HYPERVOLUME -0.6 0.6 -1 0.6 -0.6 0.6

DEPTHCOLOR 0-1 0.7+/-0.0 0.7+/-0.0 0.5+/-0.0
DEPTHCOLOR 2-5 0.25+/-0.25 0.75+/-0.25
0.25+/-0.25
TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SHEAR (0,0,0) (0.5,0.5,0.5) (2,2,2) SHEAR_XZ
SCALE (0.3,0.3,0.3) (0.4,0.4,0.4) (0.3,0.3,0.3)
ROTATE (-80,-80,-80) (0,0,0) (0,0,0)
ROTATE (0,0,0) (45,45,45) (0,0,0)
TRANSLATE (0,0,0) (-0.72,-0.72,-0.72) (0,0,0)

TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.6,0.6,0.6) (0.6,0.6,0.6) (0.6,0.6,0.6)
ROTATE (0,0,0) (50,50,50) (0,0,0)
TRANSLATE (0,0,0) (-0.4,-0.4,-0.4) (0,0,0)

TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.8,0.8,0.8) (0.8,0.8,0.8) (0.8,0.8,0.8)
ROTATE (0,0,0) (150,150,150) (0,0,0)
TRANSLATE (0,0,0) (-0.8,-0.8,-0.8) (0,0,0)

CONDENSATION 1
CONE -1.0 0.5 0.02 0.0 CONE_Y

```

Figure 3. Excerpt of domain-specific program for hypertree generation

The purpose of this section is to show that current advances in grammatical inference are now able to infer grammars for simple DSLs, such as presented in Strnad and Guid [18]. We have run the MAGIc tool on a sample of programs provided by Strnad and Guid [18]. After 30 runs (note that MAGIc uses a stochastic algorithm) the following measures on the memetic algorithm were collected: AES = 8071 (Average Evaluations to



Figure 4. Sample of generated tree

Solution), SR = 1.0 (Success Rate), while the inferred grammars have the following characteristics: ANT = 27.0 (Average Number of Terminals), ANN = 10.6 (Average Number of Nonterminals), ANP = 21.2 (Average Number of Productions), ARHS=6.4 (Average number of Right-Hand Side of production). We used an Intel Core 2 Duo P8600 processor at 2.4 GHz. The final inferred grammar found in generation 15 is shown in Fig. 5.

```

NT1 -> #resolution NT2 #iterations #num NT3 NT2
      #treedepth #num #branchdepth #num
      #hypervolume NT2 NT2 #condensation #num
      #cone NT2 #num #coney
NT2 -> #num #num #num NT4
NT3 -> #pointinit
NT3 -> #lineinit #num #num #num #num
NT4 -> #depthcolor #range #bpp #bpp #bpp NT4
NT4 -> #epsilon
NT4 -> #name #programe NT4
NT4 -> #scale #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #rotate #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #translate #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #transform #num #num NT4
NT4 -> #shear #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #rpar #shearxz NT4
NT4 -> #perturb
      #lpar #num #comma #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #comma #num #rpar
      #lpar #num #comma #num #comma #num #comma #num #rpar NT4

```

Figure 5. Inferred grammar

III. METAMODEL RECOVERY

In the programming language domain, a CFG defines the programming language’s syntax. However, in the modeling language domain metamodels are used for the same purpose. Because of this distinction the former domain is usually referred to as grammarware and the latter as modelware. Again, we are interested in domain-specific modeling (DSM) [19] which allows a higher level of abstraction than general-purpose modeling languages (e.g., UML). DSM involves the construction of a metamodel that defines the key elements of a domain and their relationships. Instances of the metamodel, called instance models, represent specific configurations of the domain and are in this respect equal to programs written in some programming language.

DSM represents a paradigm shift recognized by many researchers and is also gaining acceptance in industry. But, this wide acceptance has resulted in an increasing number of renovation problems. As a metamodel evolves, each new version captures some change in the modeling language and the instance models that are dependent on the metamodel definition need to be updated. Most existing solutions are more suited to model migration tasks and are not applicable when metamodels do not exist (e.g., after time has passed and a metamodel has evolved to the point that earlier instances cannot be loaded in the new metamodel). When the metamodel is no longer available for an instance model, the instance model will fail load into the modeling tool.

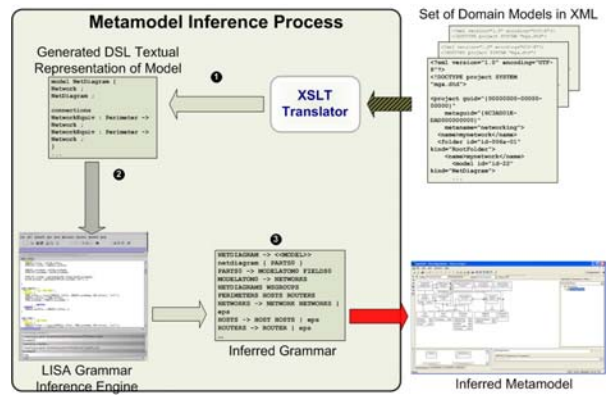


Figure 6. Overview of MARS

A. MARS

We have developed MARS (Metamodel Recovery System) [20], a semi-automatic grammar-driven system which uses GI techniques to recover metamodels from instance models. An overview of MARS is presented in Fig. 6 (originally presented in [20]). MARS consists of three main steps. In the first step, MARS takes as input a set of models exported as XML files, a capability provided by most modeling tools. These XML files are too verbose to be used as input to GI. Hence, they are mapped to an intermediate Model Representation Language (MRL), a concise DSL that describes the domain models in a form that can be used by a GI engine. An MRL program is a textual representation of the various metamodel elements (e.g., models, atoms and connections). The core of MARS is step 2 where two-way transformation from a metamodel to a CFG is defined. This transformation is thoroughly described in [20]. In the last step, the inferred metamodel is exported to an XML file that can be used to load the instance models into the modeling tool.

MARS is able to infer metamodel elements, generalizations, aggregations and connections. An example of an original metamodel, its particular model, and the inferred metamodel are presented in Figs. 7 - 9 (originally presented in [20]). The inferred metamodel (Fig. 9) was obtained from 8 instances, which exhibit all possible constructs. We are currently working on generalization of our approach with the aim to infer multi-tiered metamodels, which allows users to create models with different viewpoints in the domain.

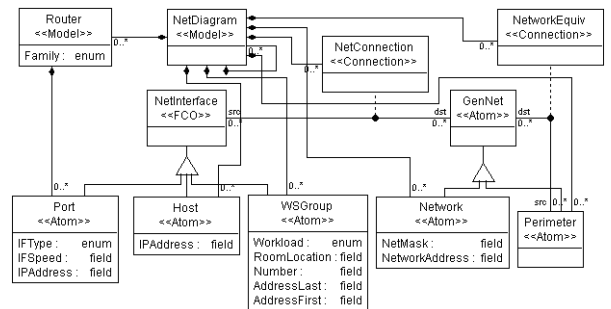


Figure 7. Original metamodel for the network domain

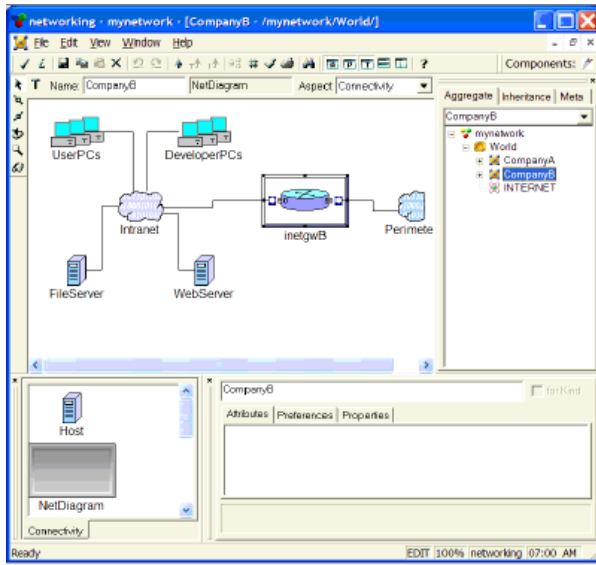


Figure 8. An instance of Network metamodel

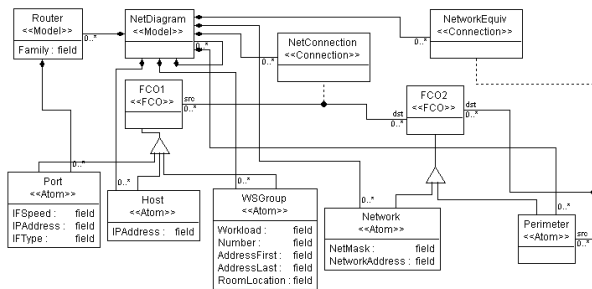


Figure 9. Inferred metamodel for the Network Domain

IV. RELATED WORK

While grammatical inference techniques have been applied to problems in diverse domains, most research is concentrated in two main areas. Many grammatical inference efforts are focused on natural language learning. EMILE [21] falls under the PACS learning paradigm and uses clustering characteristic expressions and contexts to infer natural language grammar from a corpus. The Alignment-based learning (ABL) [22] algorithm uses the principle of substitutability of constituents to compare all the sentences in a corpus in order to learn a grammar. Each sample is then compared to every other sample during a single learning phase and the differences are noted. ADIOS [23] uses a statistical method to extract hierarchical structure and learns a complete syntax from a language corpus as well as generates grammatically novel sentences in an unsupervised manner. The main difference between these research efforts and MAGIC is their focus on learning natural language; consequently, most of these algorithms assume existence of certain syntactic categories or word classes (part-of-speech tags like nouns and verbs).

To a lesser extent, grammatical inference has also been applied to the area of programming languages and software engineering. Synapse, an incremental inductive Cocke-Younger-Kasami (CYK) algorithm that learns simple CFGs from positive and negative examples, is discussed in [24]. Improvements to the Synapse system in the form of a process called bridging (to generate rules to bridge an incomplete parse tree) and serial and global search methodologies to find the minimum set of rules are described in [25]. Compared to Synapse and its various extensions, MAGIC only uses positive examples as input, a more extreme scenario of learning. Synapse also expects an ordered presentation of both positive and negative samples for efficient learning while MAGIC is not sensitive to order effects in input. Preliminary work in using evolutionary algorithms blended with local search is presented in [26] and tested on Tomita's simple regular languages. Another related work, eg-GRIDS [27], uses several operators suited to construction of CFGs (such as optional omission of a nonterminal, and attempts to detect patterns like $a^i b^j$). MAGIC uses similar operators, and in addition uses "local search" that examines differences in parsing of samples, one successful and one unsuccessful.

V. CONCLUSION

This paper presented several algorithms and application of GI in software engineering. We emphasized the newly developed memetic algorithm MAGIC which improves current results in grammar inference of DSL grammars from example DSL programs. The result is improved tool support for DSL development, assisting domain experts and software language engineers in developing a DSL and its implementation. A case study was presented that uses the approach of a real DSL for expressing hypertrees in the computer graphics domain. Metamodel recovery, yet another application of grammar inference, was briefly described as well. MARS, a semi-automatic grammar-driven system which uses GI techniques to recover metamodels from instance models was introduced and explained on a Network metamodel.

In the future, we would like to extend the experimental part with more DSL examples and enhanced local search with the information from a grammar repository (collection of GPL and DSL grammars), as well as from negative samples.

ACKNOWLEDGMENT

This material is based upon work partially supported by the National Science Foundation under Grant No. 0811630.

REFERENCES

- [1] K. S. Fu, Syntactic Methods in Pattern Recognition, Academic Press, 1974.
- [2] Y. Sakakibara, Grammatical Inference in Bioinformatics, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27, no. 7, pp. 1051-1062, 2005.
- [3] Manning, C.D. and H. Schutze, Foundations of Statistical Natural Language Processing, MIT Press, 1999.
- [4] R. Lammel and C. Verhoef, Semi-automatic grammar recovery. Software—Practice & Experience, vol. 31, no. 15, pp. 1395–1438, 2001.
- [5] E. M. Gold, Language identification in the limit. Information and Control, vol 10, pp. 447–474, 1967.

- [6] J. Oncina and P. Garcia, Inferring regular languages in polynomial update time, In *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pp 49–61. World Scientific, 1992.
- [7] M. Črepinšek, M. Mernik, and V. Žumer, Extracting Grammar from Programs: Brute Force Approach, *ACM SIGPLAN Notices*, vol. 40, no. 4, pp. 29–38, 2005.
- [8] S. Savchenko, Regular expression mining, *Dr. Dobb's Journal*, vol. 29, pp. 46–48, 2004.
- [9] B. Parekh and V. Honavar, An incremental interactive algorithm for grammar inference, In *Proceedings of the Third International Colloquium on Grammatical Inference*, ICGI-96, pp. 238–249, 1996.
- [10] R. P. Stanley, S. Fomin, *Enumerative Combinatorics Volume 2*. Cambridge University Press, 2001.
- [11] T. Back, D. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*, University of Oxford Press, 1996.
- [12] C. Grosan, A. Abraham, and H. Ishibuchi, *Hybrid Evolutionary Algorithms*, volume 75 of *Studies in Computational Intelligence*, Springer, 2007.
- [13] M. Mernik, J. Heering, and A. Sloane, When and how to develop domain-specific languages, *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.
- [14] C. G. Nevill-Manning and I. H. Witten, Identifying hierarchical structure in sequences: A linear-time algorithm, *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.
- [15] J. Brest, S. Greiner, B. Boskovič, M. Mernik, and V. Žumer, Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems, *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 6, pp. 646–657, 2006.
- [16] F. Javed, M. Mernik, B. R. Bryant, and A. Sprague, An unsupervised incremental learning algorithm for domain-specific language development. *Applied Artificial Intelligence*, vol. 22, no. 7, pp. 707–729, 2008.
- [17] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2007.
- [18] D. Strnad and N. Guid, Modeling trees with hypertextures, *Computer Graphics Forum*, vol. 23, no. 2, pp. 173–187, 2004.
- [19] J. Gray, J-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema and J. Sprinkle, Domain-specific modeling, *Handbook on Dynamic System Modeling*, Chapter 7, CRC Press, 2007.
- [20] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, MARS: A MetaModel Recovery System Using Grammar Inference, *Information and Software Technology*, vol. 50, no. 9–10, pp. 948–968, 2008.
- [21] P. Adriaans, M. Trautwein, and M. Vervoort, Towards high speed grammar induction on large text corpora, In *Proceedings of 27th Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'00*, pp. 173–186, 2000.
- [22] M. van Zaanen, Implementing alignment-based learning, In *Proceedings of 6th International Colloquium on Grammatical Inference, ICGI '02*, pp. 312–314, 2002.
- [23] Z. Solan, D. Horn, E. Ruppim, and S. Edelman, Unsupervised learning of natural languages. In *Proceedings of National Academy of Sciences*, pp. 11629–11634, 2005.
- [24] K. Nakamura and M. Matsumoto, Incremental learning of context free grammars based on bottom-up parsing and search, *Pattern Recognition*, vol. 38, no. 9, pp. 1384–1392, 2005.
- [25] K. Nakamura, Incremental learning of context free grammars by bridging rule generation and search for semi-optimum rule sets, In *Proceedings of 8th International Colloquium on Grammatical Inference, ICGI '06*, pp. 72–83, 2006.
- [26] E. Rodrigues and H. Silverio Lopes, Genetic programming with incremental learning for grammatical inference, In *Proceedings of 6th International Conference on Hybrid Intelligent Systems, HIS 2006*, pp. 47–50, 2006.
- [27] G. Petasis, G. Paliouras, C. Spyropoulos, and C. Halatsis, eg-GRIDS: Context-free grammatical inference from positive examples using genetic search, In *Proceedings of 7th International Colloquium on Grammatical Inference, ICGI '04*, pp 223–234, 2004.