Incrementally Inferring Context-Free Grammars for Domain-Specific Languages

Faizan Javed
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, USA
javedf@cis.uab.edu

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

Alan Sprague, Barrett Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, USA
{sprague, bryant}@cis.uab.edu

Abstract

Grammatical inference (or grammar inference) has been applied to various problems in areas such as computational biology, and speech and pattern recognition but its application to the programming language problem domain has been limited. We propose a new application area for grammar inference which intends to make domain-specific language development easier and finds a second application in renovation tools for legacy software systems. We discuss the improvements made to our core incremental approach to inferring context-free grammars. The approach affords a number of advancements over our previous genetic-programming based inference system. We discuss the beam search heuristic for improved searching in the solution space of all grammars, the Minimum Description Length heuristic to direct the search towards simpler grammars, and the right-hand-side subset constructor operator.

1. Introduction

In [1], Kugel makes a case for programming computers the same way children learn – from examples. This idea is known as "programming by examples" and involves providing the computer with examples (or samples) from which a program which correctly classifies the input examples can be output by the computer without the programmer providing the computer a detailed algorithm on how to do so. To accomplish this, it would require the computer to *compute in the limit* - i.e., we would take the last output of the computer as its result without requiring it to announce when an output is its last. Our work on grammar induction, or "programming by examples", elaborates on these ideas; we provide language samples to our computer, which then uses the incremental learning algorithm to come up with a CFG which can classify or parse these language samples without over generalization or overspecialization.

One of the open problems in the area of domain-specific languages (DSLs) [2], also called little languages, is how to make domain-specific language development easier for domain experts not experienced in programming language design. van Deursen et al. [3] state a terse and apt definition for DSL's : "A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". The defining characteristics of DSLs are that they are usually declarative, and smaller in size than general-purpose programming languages. A few of the possible approaches to building a DSL are to use parameterized building blocks, or by grammar induction. Grammatical inference (grammar induction or GI), a subfield of machine learning, is the process of learning of grammar from training data. Machine learning of grammars finds application in diverse domains such as software engineering, syntactic pattern recognition and computational biology. Gold's theorem [4], one of the seminal results in this area, states that it is impossible to identify any of the four classes of languages in the Chomsky hierarchy in the limit using only positive samples (unless a statistical distribution over all the positive samples is used). However, using both negative and positive samples, the Chomsky hierarchy languages can be identified in the limit.

Our research focus is on designing DSLs using Context-Free Grammar (CFG) induction techniques. In the software engineering and programming language domain, such a technique would find application in cases where legacy DSLs have been running for many years and their specifications no longer exists to assist with evolution of their implementations (e.g., as was needed to solve the Y2K problem). A survey of programming language usage in commercial and research environments has shown that more than 500 general-purpose and proprietary programming languages are in use today [5]. The Y2K-like problems not withstanding, many commercial installations use in-house DSLs and a variety of situations can arise (e.g., software company went bankrupt) where source implementations either need to be recovered or translated to a different language dialect. We expect that DSLs would usually be small enough so that GI techniques may be applied in a tractable manner to recover the underlying grammar from sample sources. While semi-automatic techniques as in [5] can be used to recover grammar in situations where compiler sources and manuals are available, our technique is also applicable when such sources are not available, whether it be in or outside the domain of software engineering (e.g. neural networks, structured data and patterns) [6]. In such situations, the grammar needs to be extracted solely from artifacts represented as sentences/programs written in some unknown language.

The importance of CFGs is paramount in programming language syntactic design and various other domains. This, along with the fact that there has so far been no successful solution to the CFG induction problem (see section 2) makes for a promising research area to explore. In this paper, we briefly discuss our Genetic-Programming [7] approach to grammar induction in the next section, and then in section 3 we introduce some improvements to our incremental approach to inferring grammars (along with some initial results) for facilitating DSL development for domain experts not conversant with programming language development and for recovering legacy DSLs.

2. Related Work

Our research problem can be reduced to the problem of inferring CFGs from language samples. The grammar inference research community has so far been successful only in inferring regular languages. Various algorithms like RPNI [8] and EDSM [9] have been developed which can learn regular languages from positive (the set of strings belonging to the target language) and negative samples (the set of strings not belonging to the target language). In [10] a genetic-programming approach was used for inferring regular grammars and compared with the RPNI algorithm.
Successful learning of CFGs has proved to be more difficult than learning regular grammars. Despite various and differing attempts at solving the problem of CFG induction [11] [12] [13] [14] [15], there has been no one convincing solution to the problem as of now. In all the work cited so far, experiments were performed on theoretical sample languages instead of on real or even toy programming languages. Based on the work done in this area so far, we conclude that learning CFGs is still an unsolved problem.

Our previous work in this area focused on using a genetic-programming based system called GenParse to infer CFGs for DSLs. It used genetic operators and parameters, and encoded the grammar into a chromosome as a list of BNF production rules [16]. Chromosomes are evaluated using the LISA compiler generator [17] at the end of each generation by testing each grammar on a sample of positive and negative samples. The system was augmented with basic data-mining techniques such as frequent sequences [18] in order to approximately infer the sub-languages first. These improvements increased the inference capability of the GenParse system, and allowed bigger DSLs to be inferred. For more details and discussion on the GenParse system, please see [7][19]. In the area of Domain-Specific Modeling, we have applied grammar inference techniques to the problem of metamodel drift, which occurs when instance models in a repository are separated from their defining metamodel. The resulting system, called MARS [25], is a semi-automatic inference system which makes use of already existing tools along with new grammar inference algorithms to recover metamodels which correctly define the mined instance models.

3. Incrementally Inferring Grammars

In the current GenParse implementation, whenever a positive sample is not accepted (or a negative sample is accepted) by the current CFG, the induction engine takes these violating samples into account and infers a new CFG which then successfully accepts the positive sample (or rejects the negative sample). Adapting this behavior so that only the minimum number of new production rules are added to the existing CFG (which would enable the positive/negative sample to be accepted/rejected) would allow the search for a suitable CFG to be carried out in an incremental and more efficient way. So far, all attempts at using incremental grammar construction for CFG inference have only succeeded in inferring simple toy grammars [11][13]. In this section, we describe improvements to the incremental algorithm introduced in [19]. The approach makes use of positive samples only since the absence of negative examples often arises in practice. We first elaborate on the theoretical foundations of the approach, and then discuss the improvements made to the algorithm.

Input to the algorithm is a set of positive samples (or programs) *Pos = {S1, S_2, ,…, S_N },* and the following auxiliary functions are used:

- *Position(Token$_i$, S_i):* returns the position of Token$_i$ in sample S_i.
- *Prefix(S_i, k):* returns the first *k* tokens in sample S_i.
- *Suffix(S_i, k):* returns the last *k* tokens in sample S_i.
- *Length(S_i):* returns the length of sample S_i
- *Diff(S_i, j, k):* returns *k* tokens in program S_i starting from position *j.*

A ordering on the samples is described by the relation "simpler than" (<*) where $_{Si}$ is simpler than S_{i+1} (S_i <* S_{i+1}) iff the following holds:

- *All tokens in S_i are also in S_{i+1}*
- *If Position (Token$_x$, S_i) ≤ Position (Token$_y$, S_i), then Position (Token$_x$, S_{i+1}) ≤ Position (Token$_y$, S_{i+1}).*

The simplest case for incremental learning occurs when the following condition holds:

Case 1: Prefix(S_i, Length (S_i)) = Prefix (S_{i+1}, Length (S_i))

This statement denotes that the sample S_{i+1} is a continuation of sample S_i, i.e., S_{i+1} is sample S_i with some new tokens appended at the end. The initial grammar is constructed using the first sample and the algorithm described in [20].

The reason for this is to have a better grammar structure as a starting point for the incremental inference process as this can facilitate a better search process.

Case 2 describes the case when the new tokens aren't appended at the end of S_{i+1}, but rather somewhere in the middle. Case 3 is the most general of all the cases and describes the situation where the samples don't have any specific ordering on them.

Case 2:

- S_i = Prefix(S_i, k) + Suffix (S_i, j) and Length(S_i)= k + j
- S_{i+1} = Prefix(S_i, k) + Diff(S_{i+1}, k +1, Length(S_{i+1}) –k-j) + Suffix(S_i, j)

Case 3:

- S_{i+1} = Prefix(S_i, k) + Diff(S_i + 1, k+1, Length(S_{i+1})-k-j) + Suffix (S_i, j)
- S_i = Prefix (S_i, k + k_1) + Suffix (S_i, j + j_1) and Length(S_i) = k + k_1 + j + j_1 , where k_1 >0 or j_1 > 0.

Our current work focus is on designing an improved algorithm for case 1. These improvements will also be valid for the other cases, and all of the cases will be incorporated into one cohesive algorithm. Table 1 details the improved algorithm. Due to space restrictions, we only provide a short description of these advancements to the algorithm. Our first improvement is the introduction of the beam search heuristic to the algorithm. At any point in time, a set (determined by the user specified variable þ) of suitable candidate grammars is stored and used for the next iteration of the grammars. Initially, the beam contains only the initial grammar.

We next introduce the use of the Right Hand Side (RHS) subset construction operator (step 2d in Table 2). If the new increment exists as a subset of a set of RHS terminals in the grammar rules, then those terminals are replaced by a new non-terminal and a new rule (NewNT → #subset) is appended to the grammar. The entire grammar is scanned for similar subsets, all of which are replaced by the new non-terminal. Table 1 shows an example of an increment (#item #price) which, when encountered, modifies grammar (a) to grammar (b). The increment is found as a subset in rule 2 of grammar (a). The subset is then merged into a new non-terminal and 2 new rules (NewNT→ #subset | ε) are added.

Table 1. RHS subset operator

NT1 → #stock NT2 NT3 NT2 → #item #price #qty NT2 | ε NT3 → #sales | ε (a)
NT1 → #stock NT2 NT3 NT2 → NT4 #qty NT2 | ε NT3 → #sales NT4 | ε NT4 → #item price | ε (b)

When only positive examples are used for inferring a grammar (as in our case), overgeneralization can be controlled by *either i) focusing on inferring a restricted class of formal languages (which have been proved to be learnable from positive examples only), or ii) using a heuristic.* For our inference algorithm, we follow the works in [21][22] and use the Minimum Description Length (MDL) [23] heuristic to direct the search towards compact grammars (i.e. few bits are required for encoding). The concept of MDL involves encoding a set of data, and then transmitting it to a receiver where it can be decoded. MDL compresses the grammar as well as encodes the training samples using that grammar. Thus, it offers a way to compare grammars and choose the one that is able to be compressed and encodes the examples using the least number of bits.

Our MDL heuristic is the sum of two components:

i) Grammar Description Length (GDL): the bits required to encode the grammar rules,

ii) Samples Description Length (SDL): the bits required to encode all examples using a grammar.

We would like to minimize the values of both the components. For computing the GDL, we assume that the total number of bits for encoding a rule is the sum of the bits required to encode the head of the rule, the body of the rule and the end of the rule. Since the rule size can be variable, a STOP non-terminal will be appended to each rule to indicate the end of the rule. If a grammar has U_{NT} unique non-terminals (excluding the STOP non-terminal), then the total number of bits required to encode a single non-terminal is given by: $\beta_{NT} = log\ (U_{NT} + 1)$.

For T unique terminals, the number of bits required is given by: $\beta_T = log\ (\ T\ )$.

Our approach involves dividing the rules into three subsets.

<u>*i) Start rules*</u>: these rules have the special property that they always have the same head, and as a result the following expression gives the number of bits required for encoding one rule R of the start subset:

$$\beta_R = (NT_R + 1)\ (log\ (U_{NT} + 1)) + (T_R\ )\ (log\ T).$$

NT_R and T_R are the number of non-terminals and terminals in the rule body, respectively.

<u>*ii) Epsilon rules:*</u> these rules have a fixed length of 1, that is, the LHS non-terminal only. We don't encode the epsilon symbol, and there is no need for a STOP symbol since all the rules are of the same size. The number of bits to encode an epsilon rule is given by:

$$\beta_R = \ log\ (U_{NT} + 1).$$

Table 2. The Incremental Learning Algorithm

```
1. Create initial Grammar $G_1$ from the first sample $S_1$ using the algorithm described in [20].
2. While more samples exist, do :
    a. Generate an LR(1) parser for all the grammars ($G_i$'s) in the Beam and try to parse the next sample. If þ or more than þ grammars successfully parse the samples (where þ is the maximum number of grammars to be held in a beam), then skip steps b-e.
    b. If $G_i$ fails to parse a sample $S_j$, reconstruct $G_i$ by making the use of the increment in tokens from $S_{j-1}$ to $S_j$ and the following sub-steps (c,d and e).
    c. If the increment already exists as the right hand side (RHS) of a rule in the grammar with non-terminal Nx as left hand side (LHS), then append Nx to the rule and add the rule Nx → epsilon to the grammar. In some cases, this will introduce recursion in the grammar.
    d. Else, if the increment does not exist as the RHS of a rule, perform the RHS subset construction procedure on $G_i$, if possible. If the procedure succeeds, try the appending of the new non-terminal to existing rules as described in the latter part of sub-step e.
    e. Else, add the rule Nx → #increment to the grammar. Append a new non-terminal Nx to the first rule and then create an LR(1) parser from the grammar and then try to parse all the samples up to and including the violating sample. This process is repeated, and Nx appended to all rules in succession (except the epsilon rules) until all the samples (up to and including the violating sample) are successfully parsed. Successful grammars are stored in the beam.
    f. Calculate the MDL scores of all the grammars in the Beam, and choose þ grammars with the highest scores.
3. If all samples are parsed successfully, output the grammar with the highest MDL score. Else, indicate failure and output the sample which can't be parsed.
```

iii) All other rules : All other rules are rules which have a non-terminal on the LHS and any number of non-terminals or terminals on the right hand side. The following equation gives the number of bits to encode a rule of this type:

$$\beta_R = (NT_R + 2)\,(\log\,(U_{NT} + 1)) + (T_r\,)\,(\log\,T).$$

For computing SDL, we need to estimate the derivation power of the grammar. This can be done by counting all the sentences which can be generated by a grammar. However, this is not possible since a grammar usually generates an infinite language. To overcome this, we use a calculation method similar to the variability calculation of feature diagrams [24], with additional rules to handle recursion. The number of all possible different sentences is calculated by the rules in Table 3, where *NT* stands for non-terminals, *a* is a terminal symbol, ***B*** denotes a non-terminal or terminal symbol, and *Var* stands for *Variability*.

$$SDL = Log\ (Var\ (G)), \text{ where } G \text{ is the grammar.}$$

Table 3. Rules for SDL calculation

Var (NT ::= $\mathbf{B}_1 \ldots \mathbf{B}_n$)	= Var($\mathbf{B}_1$) * … * Var($\mathbf{B}_n$)	
Var (NT ::= $\mathbf{B}_1$ | … | $\mathbf{B}_n$)	= Var($\mathbf{B}_1$) + … + Var($\mathbf{B}_n$)	
Var (NT ::= a)	= 1	(single terminal)
Var (NT ::= eps)	= 1	(empty production)
Var (NT ::= NT **B**)	= Var(**B**)	(left recursion)
Var (NT ::= **B** NT)	= Var(**B**)	(right recursion)

An experimental run of the DESK DSL is show in Table 4. The second column shows the 3 samples used to infer the grammar, and the third column shows the incrementally constructed grammar after each iteration. The grammar inferred after sample 3 successfully parses all samples and provides a reasonable generalization of the language being inferred.

Table 4. Experimental Run on the DESK DSL

i	S_i	G_i
1	print %b	NT1 → #print #id
2	print %b + %b	NT1 → #print #id NT2 NT2 → ε NT2 → #oper+ #id
3	print %b + %b where %b = 20	NT1 → #print #id NT2 NT2 → #oper+ #id NT3 | ε NT3 → ε NT3 → #where #id #oper= #int

In table 4, the increment during iteration 3 is (#where, #id, #oper=, #int). At this point, two suitable LR(1) grammars can be created and stored in the beam.

Table 5. MDL calculations for the DESK DSL

U_{NT} = 2 (excluding NT1 and STOP) , T = 6

Separator = $\log(U_{NT} + 1)$

NT1 → #print #id NT2 NT3
NT2 → #oper+ #id | ε
NT3 → ε
NT3 → #where #id #oper= #int

GDL = 1 start rule + separator +
2 epsilon rules + separator + 2 other rules

MDL = GDL + SDL = 38.09 + $\log_2(4)$ **= 40.096 bits**

(a)

NT1 → #print #id NT2
NT2 → #oper+ #id NT3 | ε
NT3 → ε
NT3 → #where #id #oper= #int

GDL = 1 start rule + separator +
2 epsilon rules + separator + 2 other rules

MDL = GDL + SDL = 38.09 + $\log_2(3)$ **= 39.68 bits**

(b)

Both the grammars contain 1 start rule, 2 epsilon rules and 2 rules of other kinds, but they differ in the location of non-terminal NT3. The SDL calculation shows that grammar (a) has a higher derivation power than grammar (b) because it can generate one more sentence and is more general than grammar (b). Grammar (b) however is compressed better, as the MDL score reveals. Overall, the difference in the MDL score between the two grammars isn't vast. However for bigger experiments where a larger number of grammars would need to be scored, a bigger variability in the score can occur.

We are also currently experimenting with an optimal value for the beam size. For the DESK DSL, a beam size of 1 is suitable since it is a small DSL However, for bigger DSLs where the search space would be expansive with a bigger set of possible solution grammars, a larger beam size value might be required.

4. Future Work and Conclusion

Solving the problem of CFG inference can lead to solutions to many programming language related problems such as renovation problems and development of domain-specific languages. In this paper we describe and discuss improvements to our incremental learning algorithm. Our focus was on case 1 of the learning algorithm, and we augment that algorithm with the beam search heuristic to better search in the solution space, as well as the MDL simplicity heuristic to direct the beam search towards simpler grammars to control overgeneralization. A RHS subset constructor operator is also introduced.

Our future work involves continuing work on the incremental learning algorithm, specifically developing an all inclusive algorithm which can handle all the cases described, and further investigating the MDL heuristic for bigger DSLs. We are also experimenting with the size of the beam to observe its affects on the quality of the inferred grammars for DSLs of varying size.

Since CFGs are widely used in many other domains, the results of this work will be directly applicable in many different fields such as software system renovation, development of domain-specific languages, syntactic pattern recognition, computational biology and natural language acquisition.

5. REFERENCES

[1] Kugel, P. It's time to think outside the computational box. *Communications of the ACM* 48(11): pp. 32-37 (2005).

[2] Mernik, M., Heering, J., and Sloane, A.M. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):pp. 316-344, December 2005.

[3] van Deursen, A., Klint, P., and Visser, J. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26-36, 2000.

[4] Gold, E. M. Language Identification in the Limit. *Information and Control*, 10: pp. 447-474, 1967.

[5] Lämmel, R., and Verhoef, C. Semi-automatic grammar recovery. *Software –Practice & Experience*, 31(15):1395-1438, December 2001.

[6] Mernik, M, Črepinšek, M., Kosar, T., Rebernak, D., and Žumer, V. Grammar-based systems: Definition and Examples, *Informatica*, 28(3): pp. 245-255, 2004.

[7] Črepinšek, M., Mernik, M., Bryant, B., Javed, F., and Sprague, A. Inferring Context-Free Grammars for Domain-Specific Languages. *In Proceedings of the*

Fifth Workshop on Language Description, Tools and Applications (LDTA 2005), J. Boyland, G. Hedin (Eds.), pp. 64 - 81, 2005.

[8] Oncina, J., and Garcia, P. Inferring regular languages in polynomial update time. In N. Perez de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis, volume 1 of Series in Machine Perception and Artificial Intelligence*, pp. 49 – 61. World Scientific, Singapore, 1992.

[9] Lang, K.J., Pearlmutter, B. A., and Price, R. A. Results of the Abbadingo One DFA Learning Competition and a new Evidence-Driven State Merging Algorithm, *Fourth International Colloquium on Grammatical Inference, Lecture Notes In Computer Science,* Vol. 1433, pp. 1-12, Ames, IA, July 1998, Springer-Verlag.

[10] Dupont, P. Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG method. In Rafael C. Carrasco and Jose Oncina, editors*, Proceedings of the Second International ICGI Colloquium on Grammatical Inference and Applications*, of *Lecture Notes in Artificial Intelligence*, vol. 862 , pp. 236-245, Berlin, September, Springer-Verlag, 1994.

[11] Sakakibara, Y. Learning Context-Free Grammars using Tabular Representations. *Pattern Recognition* 38 (2005): pp. 1272-1383.

[12] Laxminarayana, J. A., and Nagaraja, G. Inference of a Subclass of Context-Free Grammars using Positive Samples. In *ECML/PKDD 2003 Workshop on Learning Context-Free Grammars*, 2003.

[13] Nakamura K., and Ishiwata, T. Synthesizing Context-Free Grammars from Sample Strings based on Inductive CYK Algorithm. In *Proceedings of Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000*, *Lecture Notes in Artificial Intelligence*, vol. 1891, pp. 186-195, Lisbon, Portugal, September 11 – 13, 2000, Springer-Verlag.

[14] Oates, T., Armstrong, T., Harris, J., and Nejman, M. Leveraging Lexical Semantics to Infer Context-Free Grammars. In *ECML/PKDD 2003 Workshop on Learning Context-Free Grammars*, 2003.

[15] Nakamura, K., and Matsumoto, M. Incremental learning of Context Free Grammars based on Bottom-Up Parsing and Search. *Pattern Recognition* 38 (2005): pp. 1384-1392.

[16] Wyard, P. Representational Issues for Context Free Grammar Induction Using Genetic Algorithm. *Proceedings of the 2nd International Colloquium on Grammatical Inference and Applications*, *Lecture Notes in Artificial Intelligence,* vol. 862, pp. 222 -235, Springer-Verlag, 1994.

[17] Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V., LISA: An Interactive Environment for Programming Language Development, *Proceedings of the 11th International Conference on Compiler Construction, CC'2002, Lecture Notes in Computer Science*, vol. 2304, pp. 1 – 4, 2002, Springer-Verlag.

[18] Jiawei, H., and Kamber, M., Data Mining: Concepts and Tehniques, *Morgan-Kaufmann Publishers*, 2001.

[19] Črepinšek, M., Mernik, M., Bryant, B., Javed, F., and Sprague, A. Context-Free Grammar Inference for Domain-Specific Languages, *submitted to Science of Computer Programming (invited). Technical Report UABCIS-TR-2006-0301-1,* UAB, 2006.

[20] Nevill-Manning, C. G., and Witten, I. H. Compression and Explanation using Hierarchical Grammars. *The Computer Journal*, 40:103-116, 1997.

[21] Langley, P., and Stromsten, S. Learning Context-Free Grammars with a Simplicity Bias. In *Proceedings of Machine Learning: ECML 2000, 11th European Conference on Machine Learning*, *Lecture Notes in Artificial Intelligence*, vol. 1810, pp. 220-228, Barcelona, Catalonia, Spain, May 31 – June 2, Springer-Verlag, 2000.

[22] Petasis, G., Paliouras, G., Spyropoulos, C. D. and Halatsis, C. eg-GRIDS: Context-Free Grammatical Inference from Positive Examples using Genetic Search". *In Proceedings of the 7th International Colloquium on Grammatical Inference (ICGI 2004), Lecture Notes in Artificial Intelligence,* vol.3264*,* pp. 223 – 234, Springer, 2004.

[23] Rissanen, J. *Stochastic Complexity in Statistical Inquiry*, World Scientific Publishing Co., Singapore, 1989.

[24] Kang, K.C., Cohen, S.G., Hess, J.A., Novak W.E., and Peterson, A.S. Feature-Oriented Domain Analysis (FODA) Feasibility Study, *Technical Report, CMU/SEI-90-TR-21, ADA 235785*, Software Engineering Institute, CMU, Pittsburgh, PA, 1990.

[25] MARS: MetAmodel Recovery System: http://www.cis.uab.edu/softcom/GenParse/mars.htm