EBNF and SDT for GrGen.NET

Edgar Jakumeit

RapidSolution Software AG

Abstract. In this paper we present an extension of the Graph Rewrite Generator GrGen.NET by grammar rules which are similar to Extended Backus Naur Form grammar rules with embedded actions known from parser generators for string languages. The adaptation of this formalism to graph languages and graph rewriting – yielding Structure Directed Transformation – allows to process abstract syntax graphs concisely with a few grammar rules and declaratively without external control.

1 Motivation

The motivation for this paper and the graph rewriting language constructs introduced herein stems from the domain of formal languages and parser generators. The following listing shows an example ANTLR [13] grammar rule (from the grammar of the GrGen.NET rule language implementation, for matching terms), with terminals in upper case and nonterminals in lower case, with a Kleene star around a pattern to match it as long as possible and a bar separating several alternative cases; parameters are given in square brackets, semantic actions are given within curly braces. A `term` is translated to a tree node of type `Expr`, where a `term` consists of a factor, that may be followed by a sequence consisting of a `PLUS` or `MINUS` operator and a `factor`.

```
term[boolean inInit] returns [Expr res]
: left=factor[inInit] { res = left; }
  ( ( PLUS right=factor[inInit]
      { res = new PlusExpr(res, right); }
    | MINUS right=factor[inInit]
      { res = new MinusExpr(res, right); }
    )
  )*
;
```

In the aforementioned domains Backus Naur Form grammars built of rules like the example rule above are used to describe context free (string) languages. They consist of rules describing how nonterminals are to be replaced by one or multiple alternative sequences of terminal characters and further nonterminals. Backus Naur Form was extended with some form of regular expression notation to improve the handling of regular grammar constructs (esp. iteration) leading to Extended Backus Naur Form [21]. EBNF is a highly convenient notation [22], but notably it is brought to life by parser generators (as e.g. ANTLR [13]).

They read a declarative language specification in EBNF and generate a program capable of matching a sentence of the specified language. It is not only the language to match which is described and a parser generated for — additionally a transformation is specified with embedded actions, from the implicit concrete parse tree to an explicit abstract syntax tree. According to the mechanism of *syntax directed translation* [1] the entire input sentence is matched piece by piece along the grammar constructs and an output sentence is built piece by piece according to the embedded actions attached to the grammar constructs.

GrGen.NET has been extended with an equivalent to syntax directed translation – *structure directed transformation* – which specifies the rewriting of a graph language [1] to a graph language, assembled along the structure of the matching graph. The constructs known from *Extended Backus Naur Form* grammars decorated with embedded actions were transferred from string languages to graph languages; as string languages are different from graph languages the formalisms are not identical, but the analogy works well. The structure of this paper is as follows: after this motivation, an example domain is explained, followed by an introduction into GrGen.NET. Then the EBNF for graph rewriting constructs are introduced alongside an example, their semantics and implementation is sketched, and related work is discussed; finally we conclude.

2 Program Graphs

The example we will use to bring the argument forward originates from the domain of program graphs and program graph transformation; we want to transform an abstract syntax graph complying to a source model into a program graph complying to a target model. We will use two simplified models which are similar to the object oriented programming language graph models which can be found in the Refactoring case [6] of GraBaTs 2008 and the Reengineering [7] case of TTC 2011, just stripped down to the minimum needed for this paper. The source model is given below; the target model is structurally identical, notationally its elements have `T`- instead of `S`-suffixes.

```
node class ProgramS;
node class MethodS;
node class ConstantS {
    value:int;
}
node class AssignmentS;
edge class leftS;
edge class nextS;
```

```
node class ClassS;
node class AttributeS;
node class ExpressionS {
    operator:string;
}
edge class containsS;
edge class rightS;
edge class usedefS;
```

The abstract syntax graph is built from nodes representing the language elements; a program contains classes, a class contains attributes, constants or methods. A method contains assignment statements, an assignment contains a left and a right expression, and an expression is either a binary operator which

[1] derivable by star replacement (see section 5)

contains further left and right expressions, or a use-to-definition reference to a variable or constant which is read (or written when it appears on the left hand side). The assignment statements are linked with next edges into a list defining the order of execution. So structurally we have an abstract syntax tree along containment and left/right edges, which is extended to an abstract syntax graph with next and use-to-definition edges.

3 GrGen.NET

In this section we will have a look at the rule specification language defining the core of the general-purpose graph rewrite system GrGen.NET available from `www.grgen.net`; it builds the basis for the extensions of the following section.

Rules in GrGen.NET consist of a header and a body, with the body split into a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The header starts with the `rule` keyword, followed by the rule name, an optional list of input parameters, and optionally a list of output parameters. The pattern part is given within curly braces, as a list of graphlets, which are node and edge declarations or references, given with an intuitive syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. Attribute conditions can be given within `if`-clauses. The rewrite part is specified by a rewrite block nested at the end of the pattern part after the `modify` keyword. Graph elements declared in the rewrite-block are created, all other graph elements are kept, unless they are specified to be deleted within a `delete()`-statement. Attribute recalculations can be given within an `eval`-statement. These and a lot more language elements are described in more detail in the GrGen.NET user manual [2].

The following example rule is used for optimizing a program graph by replacing `a*2` with `a+a`; Figure 1 displays a snapshot from debugging this rule with the GrGen.NET components GrShell and yComp on an example graph, with the match highlighted.

```
rule optimize(e:ExpressionS) : (ConstantS)
{
   e -:leftS-> el:ExpressionS -u:usedefS-> c:ConstantS;
   e -:rightS-> :ExpressionS -:usedefS-> a:AttributeS;
   if { e.operator=="*" && c.value==2; }
   modify {
      delete(u);
      el -:usedefS-> a;
      eval { e.operator = "+"; }
      return(c);
   }
}
```

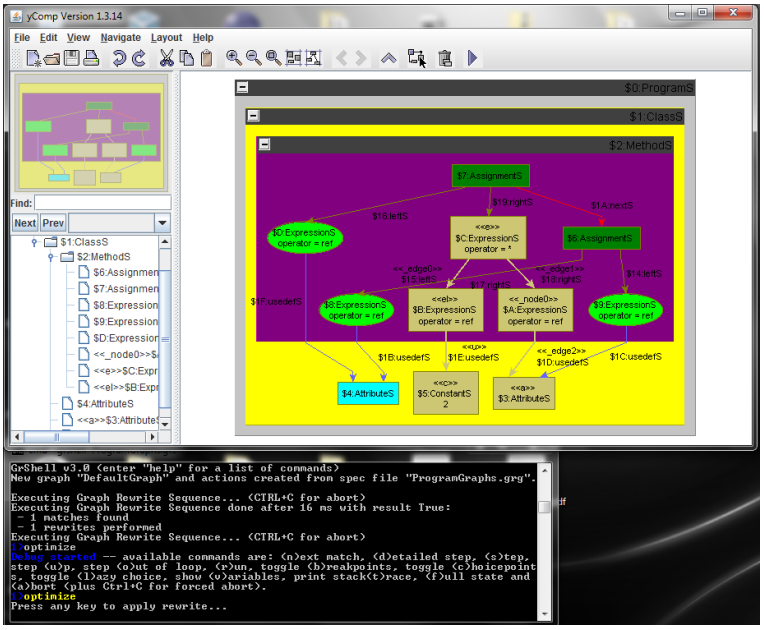

Fig. 1. Debugging the optimize rule

4 EBNF for Graph Rewriting

In the preceding section we had a look at the conventional rules with their fixed left hand side (LHS) patterns and right hand side (RHS) rewrites. In the following we will give an introduction to the structure directed transformation with EBNF like constructs and embedded actions as implemented in GrGen.NET, by matching a spanning tree of a program graph along the containment edges and creating a replicate of it complying to the destination model. This is only of limited use as such, but it is the backbone operation of program graph transformations, to which the functionalities needed for the tasks at hand are attached. While defining an exogenous transformation to showcase the ability to clone tree structures, one could easily define an endogenous transformation only modifying the found match instead.

The GrGen.NET equivalent of the grammar rules offered by parser generators are the subpatterns, introduced by the keyword `pattern`. They resemble the normal rules, with the key difference that they can not be called from the rule application control language of GrGen or from API level, but are fragments to be used from normal rules only. Subpatterns may declare parameters to define locations where matching should start, and they may declare rewrite parameters in the header to define locations where rewriting should continue. The graphlets known from the rules are used in specifying the pattern and rewrite part, too. In addition to the nodes and edges already known from the graphlets, subpattern occurrences can be declared by giving an optional entity name, then a colon

followed by the subpattern type name and a list of input (or output) parameters. In the rewrite block the rewrite parts of the matched subpatterns can get applied by using call syntax on the entity names of the matched subpatterns.

These constructs are employed in the listing given in Figure 2, which shows five grammar rules for matching a containment tree in the syntax graph. The first one, the subpattern `Program` matches the root node `p` of the abstract syntax graph and i) one or more ii) classes. The "i) one or more" is realized by the EBNF plus operator `()+`. The "ii) classes" here means: a node `c` of type `ClassS` linked to the program node, and a subpattern `cls` of type `Class` starting at the class node. The pattern nested inside the iteration construct `()+` is not sequentially following as in textual languages: it is matched anywhere in the graph — but normally nodes declared outside the pattern are referenced inside, these border nodes are then starting points of the search for pattern instances. A maximum match is sought for them, i.e. they are matched *eagerly* until all available pattern instances are covered. The modify parts of the `Program` subpattern create the program node `pt` of the target model, the classes `ct` of the target model, and request the execution of the rewrite part `cls` of the matched `Class` subpatterns, starting at `ct`. The `Class` subpattern matches from the class node `c` which is passed as parameter onwards zero or more methods or attributes or constants; the most interesting construct here is the alternative case distinction, with the alternative cases separated by bars. The `AssignmentList` subpattern matches recursively a list of assignments: each list element consists of an assignment with a LHS and RHS `Expression`, matching the list terminates at the end of the list when the optional (denoted by a question mark) following list element can not be found. The subpattern `Expression` finally matches recursively a binary tree: two nodes `el` and `er` of type `ExpressionS` as left and right children of the parent expression node `e`, including further `Expression` subpatterns starting at them, or the empty pattern. Here we could match use-to-definition edges going from the expression nodes to the attribute or constant nodes.

5 Semantics and Implementation

A formal semantics of the (E)BNF like language constructs of GrGen.NET is given in [9]. It is based on star grammars [3], cloning of nonterminals, and pair graph grammars [14]. The basic idea is to use a two-level graph rewrite process: first, a language of graph rewrite rules is derived by pair star substitution, then the host graph is rewritten with one of the resulting rules.

In this formalism rules contain stars, which are nonterminal nodes with their incident edges (their rays), leading to terminal border nodes. The stars are decorated by cardinalities, specifying lower and upper bounds on how often they may be cloned. Star rules describe how a star can be substituted by a graph, maybe containing further stars; this substituting graph is glued to the border nodes of the star. A rule application consists of i) cloning the star and its incident rays, producing cloned stars of a number in between the upper and lower bounds specified, ii) then applying a matching star rule on each clone. For each star

```
1  pattern Program {
2    p:ProgramS;
3    ( p -:containsS-> c:ClassS; cls:Class(c);
4      modify { pt -:containsT-> ct:ClassT; cls(ct); }
5    )+
6    modify { pt:ProgramT; }
7  }
8
9  pattern Class(c:ClassS) modify(ct:ClassT) {
10   ( ( c -:containsS-> m:MethodS; meth:Method(m);
11       modify { ct -:containsT-> mt:MethodT; meth(mt); }
12     | c -:containsS-> a:AttributeS;
13       modify { ct -:containsT-> at:AttributeT; }
14     | c -:containsS-> cs:ConstantS;
15       modify { ct -:containsT-> cst:ConstantT; }
16     )
17     modify { }
18   )*
19 }
20
21 pattern Method(m:MethodS) modify(mt:MethodT) {
22   m -:containsS-> a:AssignmentS; al:AssignmentList(a);
23   modify { mt -:containsT-> at:AssignmentT; al(at); }
24 }
25
26 pattern AssignmentList(a:AssignmentS) modify(at:AssignmentT) {
27   a -:leftS-> el:ExpressionS; expl:Expression(el);
28   a -:rightS-> er:ExpressionS; expr:Expression(er);
29   ( a -:nextS-> na:AssignmentS; al:AssignmentList(na);
30     modify { at -:nextT-> nat:AssignmentT; al(nat); }
31   )?
32   modify {
33     at -:leftT-> elt:ExpressionT; expl(elt);
34     at -:rightT-> ert:ExpressionT; expr(ert);
35   }
36 }
37
38 pattern Expression(e:ExpressionS) modify(et:ExpressionT) {
39   (
40     e -:leftS-> el:ExpressionS; exp1:Expression(el);
41     e -:rightS-> er:ExpressionS; exp2:Expression(er);
42     modify {
43       et -:leftT-> elt:ExpressionT; exp1(elt);
44       et -:rightT-> ert:ExpressionT; exp2(ert);
45     }
46   |
47     modify { }
48   )
49 }
```

Fig. 2. Subpatterns for replicating a program graph

all alternatives and all admissible clones are enumerated. This way, we derive a language of patterns that may be infinite (due to recursion or due to unlimited upper bounds). Finally, the resulting language it matched in the graph.

A short hint on the mapping of the syntactical to the semantical constructs: A subpattern maps to a star rule. An alternative with k cases maps to k star rules with the same left star. An EBNF iteration construct maps to a star rule with a non $[1;1]$ bound. A subpattern usage, an alternative, and an iteration are inserted as stars in their containing rule, with rays to the elements from the containing pattern which are referenced.

This holds for singular stars, stars which occur only in the pattern part. But as we are not only interested in matching a pattern but also in rewriting it, most of the time we work with pairs: a star in the pattern part and a star in the rewrite part linked together. They are substituted by pair star rules in lockstep, the pattern star is replaced by the pattern graph, and the rewrite star is replaced by the rewrite graph. This way we not only derive a language of pattern graphs, but a language of pattern and rewrite graph pairs, i.e. rules (according to SPO semantics, with common elements kept, LHS only elements deleted, and RHS only elements created).

Instantiating graph rewrite rules first, and matching them afterwards is only possible in theory – in practice, we have to interleave instantiation with matching, as it is done by the triple pushdown machine generated by GrGen.NET. This machine built from a call stack containing already matched patterns, an open tasks stack, and a result assembly stack is driven by recursive descent parsing with backtracking. For every subpattern first the terminal parts are matched, then the machine descends to the used subpatterns (stars). Alternatives cases are tried out one after another, until one leads to a match. Iteration patterns are matched until the upper bound is reached or matching fails. After a complete match was found, a match tree is constructed. It is the base for the rewriting, which is carried out during a depth first walk of the match tree. The implementation is explained in more detail in the user manual [2] and in [9].

6 Discussion

In section 4 a concise notation for specifying context free graph languages inspired by EBNF grammars for context free string languages was introduced alongside an example. The following Table 1 summarizes the correspondence between GrGen.NET language constructs and language constructs known from parser generators, like e.g. ANTLR.

Because there is no implicit first following character available as in string languages with their strict linear ordering defined by concatenation, the subpatterns must be given parameters to specify where to match (this allows to match into depth by handing in succeeding nodes); the iterated patterns are matched into breadth, starting at the nodes of the outside pattern referenced from inside the iterated pattern. In addition to these context free constructs known from context free grammar specifications, GrGen.NET supports context sensitive

Table 1. Corresponding notational and conceptual elements

Graph EBNF	String EBNF
graphlets	terminal characters
subpattern entities	nonterminals
subpattern definitions	grammar rules
`(.|.)` (alternative cases)	`(.|.)`
`(.)*` (zero or more times)	`(.)*`
`(.)+` (one or more times)	`(.)+`
`(.)?` (zero or one time)	`(.)?`
rewrite parts	semantic actions

negative and positive patterns (application conditions [4]) with the syntax `~(.)` and `&(.)`. The performance of the generated parser depends on the amount of backtracking which is required. Matching and rewriting syntax graphs with ten thousands of elements can be done with nearly no backtracking and is carried out in a fraction of a second. But running times explode for other tasks which require a lot of search and backtracking.

This is a follow up paper of [5] where the BNF like constructs were already introduced. The additions are: i) an introduction of an analogy of syntax directed translation to structure directed transformation, ii) the step from BNF to EBNF with the pattern iteration constructs and their concrete GrGen notation, and iii) a first implementation of these constructs. Similar BNF like graph pattern matching devices are available in the VIATRA2 [20] or Tefkat [10] tools, but constrained to matching and simple BNF only. More specialized language constructs are multinodes and iterated paths. More general mechanisms are triple graph grammars, bottom up graph parsers, and backtracking control for inverted graph rewrite rules to build a parser.

Multinodes as available in e.g. PROGRES [17] are a limited version of pattern iteration; they correspond to an iteration of a node with its incident pattern edges. Several graph transformation tools have implemented a star operator over edges to define iterated paths (e.g. PROGRES [17], GROOVE [16], GReTL [8]); this can be seen as syntactic sugar for a recursive subpattern, matching an edge from a start node to a successor node, and then calling itself from the successor node on. An earlier, more general approach to structure directed transformation are Triple Graph Grammars [18], which allow to define the correspondence between two different models in a declarative way with a correspondence model in between. They can be made operational in either direction giving a transformation from the source to the target model or vice versa. In comparison, the EBNF with rewrite part constructs are limited to unidirectional transformations, pair graph relations, and context free graph languages derivable by star replacement; they are only used as fragments in defining more powerful rules. The backtracking recursive decent parsers generated from the EBNF constructs in GrGen.NET are similar to the graph parser combinators given in [11] — in contrast to the bottom up (CYK like) parser presented in [12] for a more general class of graph languages defined by adaptive star grammars, and

to the parsing algorithm introduced in [15] for context-sensitive layered graph grammars. A further more general approach to parsing is implemented in AGG [19], which offers backtracking control on a set of general graph rewrite rules. In a preliminary step the language generating grammar is transformed into a parsing grammar by exchanging the left hand sides and right hand sides of the rules. The parsing grammar is then executed under backtracking control until the given graph was reduced to the start graph or all reduction paths got stuck.

The EBNF and SDT constructs are more general and powerful than the language constructs cited at the beginning of the previous paragraph but much less powerful than the lastly cited general graph parsing or structure directed rewrite mechanisms. A *simple* means which allows an efficient handling of a limited, but *highly useful* class of context free graph languages; a declarative sublanguage employed to increase the expressiveness of the conventional graph rewrite rules, still used operationally.

7 Conclusions

The rule specification language of GrGen.NET was extended by constructs similar to the ones which can be found in EBNF grammars, giving a concise and convenient notation for expressing star-derivable context free graph languages, and for declaratively specifying transformations of such languages in a structure directed way with embedded rewrite parts. They are brought to life by backtracking recursive descent parsers which are generated out of the specifications by the graph rewrite generator. They allow to cut down on the number of rules and the size of the orchestrating control needed for complex graph transformation tasks, by shifting the rewriting of tree like sublanguages below graph rewrite rule level.

We want to thank the anonymous reviewers and Berthold Hoffmann for their valuable comments.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley (1986)
2. Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. Internal Report (2011), `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`
3. Drewes, F., Hoffmann, B., Janssens, D., Minas, M.: Adaptive star grammars and their languages. Theoretical Computer Science 411, 3090–3109 (2010)
4. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundamenta Informaticae 26, 287–313 (1995)
5. Hoffmann, B., Jakumeit, E., Geiß, R.: Graph Rewrite Rules with Structural Recursion. GCM 2008 (2008)
6. Hoffmann, B., Perez, J., Mens, T.: A Case Study for Program Refactoring (2008)

7. Horn, T.: Model Transformations for Program Understanding: A Reengineering Challenge (2011)
8. Horn, T., Ebert, J.: The GReTL Transformation Language (2011)
9. Jakumeit, E.: Mit GrGen.NET zu den Sternen (2008)
10. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. In: MoDELS Satellite Events. pp. 139–150 (2005)
11. Mazanek, S., Minas, M.: Parsing of Hyperedge Replacement Grammars with Graph Parser Combinators. In: Ermel, C., Heckel, R., de Lara, J. (eds.) GT-VMT 2008 (2008)
12. Minas, M.: Parsing of Adaptive Star Grammars. In: GraMoT (2006)
13. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages (2007)
14. Pratt, T.W.: Pair Grammars, Graph Languages and String-to-Graph Translations (1971), Journal of Computer and System Sciences, Volume 5, pages 560–595
15. Rekers, J., Schürr, A.: Defining and Parsing Visual Languages with Layered Graph Grammars. Journal of Visual Languages and Computing 8(1), 27 – 55 (1997)
16. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. AGTIVE 2003 (2004)
17. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2. pp. 487–550 (1999)
18. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, vol. 903, pp. 151–163 (1995)
19. Taentzer, G.: AGG: AGraph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J., Nagl, M., Bhlen, B. (eds.) AGTIVE 2003, vol. 3062, pp. 446–453. Springer (2004)
20. Varró, G., Horváth, A., Varró, D.: Recursive Graph Pattern Matching. In: Applications of Graph Transformations with Industrial Relevance, pp. 456–470 (2008)
21. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM 20, 822–823 (November 1977)
22. Wirth, N.: Compilerbau. B.G. Teubner (1986)