Grammar Programming in TXL

Thomas R. Dean
Queen's University
Kingston, Canada
dean@cs.queensu.ca

James R. Cordy
Queen's University
Kingston, Canada
cordy@cs.queensu.ca

Andrew J. Malton
U. of Waterloo
Waterloo, Canada
malton@waterloo.ca

Kevin A. Schneider
U. of Saskatchewan
Saskatoon,Canada
kas@cs.usask.ca

Abstract

Syntactic analysis forms a foundation of many source analysis and reverse engineering tools. However, a single grammar is not always appropriate for all source analysis and manipulation tasks. Small changes to the grammar can make the programs used to accomplish these tasks simpler and more straightforward. This leads to a new paradigm of programming these tools: grammar programming. This paper discusses several industry proven techniques that can be used when designing base grammars and when making task specific changes to grammars.

1. Introduction

Syntactic analysis plays a large part in the analysis and manipulation of software systems. Syntax is the framework on which the semantics of most modern languages is defined. We use the syntax to deal with scoping of names and precedence of operators. Not surprisingly, syntactic analysis forms a foundation of many source analysis and reverse engineering tools such as ASF+SDF [3], Stratego[18], REFINE [19], Draco[15] and TXL [4,5].

When using the TXL language, most programmers use a common grammar for the language that is being analyzed or modified. TXL also allows the common grammar to be modified on an *ad hoc* basis by each of the programs. Small changes to the grammar can make significant impact on the simplicity and maintainability of the programs. Thus a new paradigm of programming is born: custom grammar modification.

2. Source Transformation in TXL

TXL is a programming language specifically designed to support structural source transformation. The structure of the source to be transformed is described using an unrestricted ambiguous context free grammar from which a parser is automatically derived. While based on a top-down approach, this parser has heuristics to resolve both ambiguities and left recursion.

The transformations are described by example, using a set of context-sensitive structural transformation rules from which an application strategy is automatically inferred. Many of the principles behind native patterns [16,17] also apply to TXL's by-example nature. Some examples of our experience in using TXL for software engineering problems are reported elsewhere [7].

2.1 The TXL Processor

Figure 1 shows the structure of the TXL language system. A TXL program which is comprised of a grammar and a set of rules is read by the TXL processor. A parser is derived from the grammar and the input is parsed based on that grammar. The resulting parse tree is passed to the transform engine which applies the rules transforming the input tree as they run. The last phase of the processor walks the parse tree and produces the transformed output.

A full description of the TXL language is beyond the scope of this paper. However, we will give a short overview for readers who are not familiar with the language. Figure 2 shows a subset of the grammar for a simple Pascal-like language. The special name 'program' defines the goal symbol of the grammar. Square brackets denote the use of a non-terminal in a production. Prefixing a reference to a non-terminal symbol with the keyword *repeat* indicates a sequence of the non-terminals. Thus a program in our sample language is the keyword "program", an identifier ([id]) and a semicolon followed by a sequence of definitions and a block, terminated by a period.

Vertical bars ('|') are used to indicate alternatives in the grammar productions. A definition in our sample language is a [constant_section], a [variable_section], a [type_section] or a [procedure_definition]. The *list* keyword is like *repeat*, except that each instance of the non-terminal is separated by a comma. Thus the [args] non-teminal is defined as a list of expressions separated by commas and surrounded by round brackets.

The terminals of the grammar are tokens, which are referred to in the same way that non-terminals are used. In our example subset grammar, the symbol [id] refers to the identifier token which defaults to a contiguous sequence of

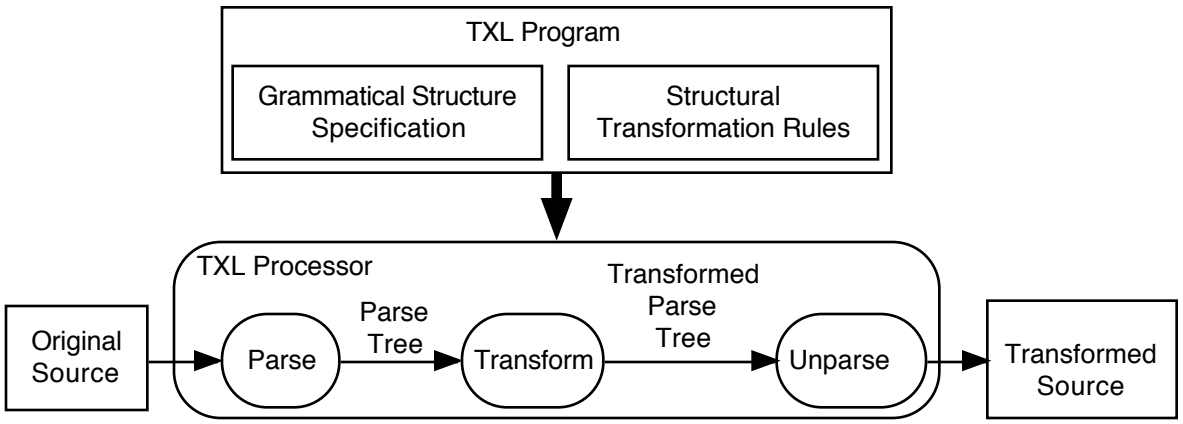

Figure 1. The TXL processor

letters, digits and underscores starting with a letter or underscore (i.e. a C identifier). The definitions of tokens can be changed by providing regular expression definitions, permitting the tokens conventions of other languages to be recognized. For example, COBOL identifiers (hyphens instead of digits) or Pascal style strings (‘’ instead of \’ for embedded quotes) can be defined as tokens.

2.2 Example: RSF from SQLj

Figure 3 shows a simple program used to extract a hypothetical Rigi Standard Format (RSF)[14] relation ‘MethodHostVar’ which identifies the host variables used in embedded SQL statements in each Java method. As a simple example, issues such as unique identification of methods and variables are ignored (they are addressed elsewhere [8, 13, 9]). This program shows three techniques commonly used in TXL programs.

The first line includes the Java base grammar providing the definition of the Java language. The second line includes the SQLj subgrammar. This grammar modifies the Java grammar to include SQLj, an embedded SQL extension to Java. The subgrammar links to the parent grammar through the use of redefinitions (explained next) to extend the base grammar.

The next three lines (ignoring blank lines) add a definition for RSF relations which are three identifiers on a line. The ‘[NL]’ non-terminal is a formatting instruction to TXL to insert a newline when writing out the result tree.

The next five lines redefine the definition of [host_variable] in the SQLj subgrammar to embed an RSF relation. The ‘...’ character sequence means ‘the previous definition’ of this non-terminal symbol. So the redefinition of host_variable is whatever a host variable was before or an RSF relation followed by a colon (‘:’) followed by an identifier (id).

The last grammar modification redefines the whole program to be whatever it was before (a Java program with embedded SQL) or a sequence of RSF relations.

All the main rule of the program does is invoke the rule annotateHostVars followed by the function replaceByRSF. The rule annotateHostVars visits each method once and the pattern of the rule separates the header (type,

```
define program
   program [id] ;
   [repeat definition]
   [block].
end define

define definition
      [const_section]
    | [variable_section]
    | [type_section]
    | [procedure_definition]
end define

define var_decl
    [id] : [typeName]
end define

define procedure_call
    [id] [args]
end define

define args
    '( [list expression] ')
end define
```

Figure 2. Subset of a Pascal-like language

```
include "Java.Grammar"
include "SQLj.Grammar"

define RSF_Relation
  [NL] [id] [id] [id]
end define

redefine host_variable
    ...
  | [RSF_Relation]
    : [id]
end redefine

redefine program
    ...
  | [repeat RSF_Relation]
end  redefine

function mainRule
  replace [program]
    P [program]
  by
    P [annnotateHostVars]
      [replaceByRSF]
end  function

rule annotateHostVars
  replace $ [method]
    T [type] Name [id] P [parms]
      B [body]
  by
    T Name P
      B [doEachHostVar Name]
end  rule

rule doEachHostVar MethName [id]
  replace [host_variable]
          : VarName [id]
  by
    'MethodHostVar MethName VarName
    : VarName
end  rule

function replaceByRSF
  replace [program]
    P [program]
  construct Rels [repeat RSF_Relation]
    _ [^ P]
  by
    Rels
end  function
```

Figure 3. Generating RSF relations

name and parameters) from the body. It calls the rule doEachHostVar on the body of the method with the method name as a parameter.

The rule doEachHostVar adds the RSF relation MethodHostName to each host variable expression. The RSF relation has the method name and the variable name as arguments. The rule terminates when all host variables in that method have been annotated.

The rule replaceByRSF is applied to the entire program. It uses the built in extraction rule ('^') to retrieve all of the RSF relations in the program. The entire program is then replaced by the RSF relations.

Figure 4(a) shows a trivial snippit of java code with embedded SQL using SQLj. It is a method which returns the commission rate given the employee number for the salesperson. Figure 4(b) shows the code after the rule [annotateHostVars] has been run. Two RSF relations are now embedded in the source. The rule replaceByRSF will extract the two RSF relations and generate the following output:

```
MethodHostVar getComm result
MethodHostVar getComm empId
```

```
float getComm(int empId){
  float result;
  #sql { select Commission
      into :result


      from Salary where
      empNo = :empId }


  return result;
}
```

(a)

```
        String getComm(int empId){
  String result;
  #sql { select Commission
      into
      MethodHostVar getComm result
      :result
      from Salary where
      empNo =
      MethodHostVar getComm empId
      :empId }
  return result;
}
```

(b)

Figure 4. Example snapshots for the program in figure 3

The TXL program from Figure 3 illustrates two major paradigms of TXL programming. The first is the separation of sublanguages from the base grammar. This same technique can be used to handle embedded SQL in COBOL or the use of CICS in COBOL or C. In this way TXL is similar to the use of language modules in ASF+SDF [2].

The other, and more important, paradigm is the local changes to the grammar to simplify the program. Without these changes, the RSF relations would have to be accumulated in a (possibly global) variable and written to a separate file. The program would not be as simple, and in our experience, not nearly as clear.

The point of both this example, and of the paper, is that the grammar is as likely to be changed as a rule is to be written. TXL programmers colloquially refer to the practice as grammar programming, and the maxim underlying the this practice is to "let the parser do the work".

3. Grammar Programming

For us, grammar programming means writing a context free grammar which is designed or adapted to the particular task, be it analysis or transformation. Whereas the grammar which defines the front end of a compiler or other language processor is typically fixed and static, with grammar programming the programmer specifies or changes the parsing rules according to his needs. Within a collection of related tasks implemented using grammar programming, the same input language (e.g. C source code) may be parsed using many different grammars.

In our experience, this is most conveniently implemented by introducing a base grammar which defines the outline structure of the expected input language, and the standard non-terminal categories. The base grammar is often recovered directly from language manuals, by hand or in a manner similar to Lämmel and Verhoef's semi-automation [12]. A TXL source program includes the base grammar and then uses grammar overrides to redefine non-terminals or whole substructures of the base grammar, as required by the task.

Because the non-terminals of the base and subset grammar are referred to in the rules, it is important that the names of the non-terminals reflect the role that the non-terminal plays in the grammar. This is very similar to the generally accepted practice in procedural programming of giving constants and variables names that are representative of the real world concept they represent. The names of the non-terminals are typically those that are found in standard reference grammars. These observations are not new [16,17], but they are relevant to the discussion of the rest of the paper.

There is a difference between the typical grammar used in a compiler and the typical base grammar used for a TXL program. One of the purposes of the compiler grammar is to perform a syntactic check of the input, while a TXL grammar used for reverse engineering or design recovery is operating on code that has already been checked by a compiler. There are distinctions important when checking syntax that are unimportant when analyzing a program. For example, in standard Pascal, the order of constants, types variables and procedures in a block are defined and multiple instances are not permitted. An analysis grammar may allow any number of these sections in any order.

There are also distinctions that a compiler grammar may not make that might be important to an analysis program. For example, in the standard C reference and most C compiler grammars, the typedef keyword is treated as a declaration specifier. There is no distinction at the grammar level between the declaration of a type using typedef and the declaration of a variable. A analysis grammar might choose to separate the two cases.

The rest of this paper examines several grammar programming paradigms we have developed over years of TXL programming.

4. Grammar Programming Paradigms

There are several grammar programming paradigms that we have developed with experience in programming in TXL. These techniques are not limited to TXL and may be applied in other typed rewriting systems such as ASF+SDF. Because of limited space, all of the examples we examine in this paper are necessarily simplified from the form that would be used in real TXL applications. For example, references to variable names in programs may involve various qualifiers and modifiers. We also ignore most the details of the rules that use the techniques to solve a particular problem.

4.1 Rule Abstraction

Even if the base grammar is generalized as suggested in Section 3, there may be distinctions that are necessary in general but unimportant for a particular application. A simple approach would be to write separate rules for each case. Alternatively, one can remove the distinctions by overriding the grammar. One example of this type of problem is identifying those variables that are used in arithmetic contexts (i.e. addition,subtraction, etc.) in the COBOL language.

The COBOL base grammar has the normal multiple levels of precedence in the expression grammar, and the

```
define arithmetic statement
    [add_statement]
  | [subtract_statement]
  | [mult_statement]
  | [divide_statment]
  | [compute_statement]
end  define

redefine statement
    [repeat RSF_relation]
    [arithmetic_statement]
   | ...
end  redefine
```

```
rule annotateArithStatements PN[id]
 replace [statement]
    Arith [arithmetic_statement]
 construct ArithIds [repeat id]
    _ [^ Arith]
 construct FS[repeat RSF_relation]
        _ [buildRSF 'Arith PN
                          each ArithIds]
 by
    FS
    Arith
end rule
```

Figure 5. Rule abstraction: recognizing arithmetic statements

non-terminal [statement] derives *all* of the different statements. Without grammar modification, we must write separate rules that target each level of precedence and each type of arithmetic statement (e.g. ADD statement). By modifying the grammar, we can significantly reduce the number of rules needed to extract the information. Figure 5 shows the grammar overrides to handle arithmetic statements and the single rule that is used to extract uses of identifiers in arithmetic statements.

The grammar modifications, shown on the left hand side of the figure, accomplish two things. The first is that the five COBOL arithmetic statements are grouped under a single non-terminal called [arithmetic_statement]. The second is that the [statement] non-terminal is changed to recognize arithmetic statements before other statements. This grammar is ambiguous since the five statements characterized as arithmetic statements can be also reached through the other alternative of the non-terminal [statement]. The TXL parser resolves the ambiguity by choosing the first matching alternative.

The right hand side of Figure 5 shows how this modified grammar is used by the rule annotateArithStatements. The rule is called with the COBOL program name as a parameter (PN) and visits each arithmetic statement extracting the identifiers from the rule. The function buildRSF is used to build an RSF relation Arith for each of the identifiers. The changes to the grammar allow us to write one rule where the base grammar would require five separate rules. The RSF relations generated by annotateArithStatements would be gathered together and output by a rule similar to replaceByRSF from Figure 3.

4.2 Grammar Specialization

Grammar productions, like procedures are often reused when the same concept is reused in the grammar. For example, the same non-terminal may be used for all references to names of variables. While this may be useful for most analysis and manipulation tasks, for some tasks, some distinctions may be necessary.

Figure 6 shows an example of how the grammar can be used to distinguish between the declaration and use of variables in a Pascal-like language. We only show the subset of the grammar involving variable declaration, and variable and function reference. In the locations of the grammar where a name is declared, the [decl_name] non-terminal is used, while where a name is referenced, the [ref_name] non-terminal is used. Both of these symbols derive the same non-terminal, [name], which in turn derives whatever a name is in the language. This permits rules to separately target declarations and references of names. This particular case is often used when building base grammars.

```
define var_decl
   [list decl_name] : [type]
end  define

define factor
    'not [factor]
  | [ref_name] [opt arguments]
end  define
```

```
define decl_name
  [name]
end  define

define ref_name
  [name]
end  define
```

Figure 6. Grammar specialization example

```
define get_ref
    [ref_name]
end  define

define put_ref
    [ref_name]
end  define
```

```
redefine assignment_statement
    [put_ref] := [expression]
end  redefine

define factor
    'not [factor]
  | [get_ref] [opt arguments]
end  define
```

Figure 7. Example GetRef/PutRef grammar overrides

Another example of this technique that would be used *ad hoc* for a specific TXL program would be to use overrides to further refine the grammar to distinguish between references that modify variables (i.e. left hand side of assignment) and references that do not modify variables [11, 20]. Figure 7 shows some of the grammar overrides for such a program. We define the non-terminals [get_ref] and [put_ref], both of which derive the non-terminal [ref_name], which is defined as it was in Figure 6. Uses of [ref_name] in the grammar are redefined to use the appropriate non-terminal. For example, the assignment statement, which modifies the left hand side of the assignment operator is redefined to use the [put_ref] non-terminal and factor, which represents a read access to a variable as a get reference ([get_ref]). The grammar treats a function call as a get_ref of the function.

4.3 Grammar Categorization

One example of the categorization paradigm is the typedef problem in C that we discussed previously. The grammar provides separate grammar non-terminals and productions for variable declarations and for type definitions using typedef. In this case the grammar is unambiguous. On branch of the grammar requires the keyword typedef, while the other branch does not permit typedef to occur.

Another example is shown in Figure 8. In this example, an ambiguity is deliberately introduced in the grammar in method call. As mentioned in section 4.2, the TXL parser resolves ambiguities by choosing the first listed choice that matches. Thus functions with jdbc names will be parsed as a jdbc method call. This is an somewhat simplistic example in that any method with the same name will be classified as a jdbc function call. But it suffices to illustrate the technique.

4.4 Union Grammars for Translation

When automatically translating between two languages, the grammar for the conversion program must be able to hold both languages. If the grammars are similar, they can be combined at each level where they match. For example, when translating C to Pascal, both languages are block/statement/expression based languages. We could combine the grammars at the global declaration level, the procedure level, the statement level and the expression level. As mentioned earlier, we assume the input is correct since the mixed grammar will allow mixed programs as input.

Figure 9 shows how the grammar can be combined at several of these levels. We redefine the non-terminal [program] to be a Pascal or a C program. The two grammar are rejoined at the non-terminal [decl] since a C program is a sequence of declarations and a Pascal program is a sequence of declarations followed by a block (the main program).

The non-terminals [begin_or_brace] and [end_or_brace] handle the minor differences between Pascal and C blocks as does the minor difference of the presence or absence of the then keyword in the if statement The new definition of the non-terminal [block] is a merge of the Pascal and C definitions which allows local declarations.

```
define method_call
    [jdbc_call]
  | [id] [arguments]
end  define

define jdbc_call
   [jdbc_name] [arguments]
end  define
```

```
define jdbc_name
    createStatement
  | prepareStatement
  | executeUpdate
  | executeQuery
  | getRow
end  define
```

Figure 8. Grammar categorization example

```
define program
    [pascal_program]
  | [c_program]
end  define

define pascal_program
  'program [id] [file_header]
    [repeat decl]
    [block] '.
end  define

define c_program
  [repeat decl]
end  define

define begin_or_brace
  'begin | '{
end  define

define end_or_brace
  'end | '}
end  define

define block
  [begin_or_brace]
  [repeat decl]
  [repeat statement]
  [end_or_brace]
end  define

define if_statement
  'if [expression] [opt 'then]
  [statement]
  'else [statement]
end  define
```

Figure 9. C/Pascal union grammar

When the languages are farther apart, an alternate technique must be used. In this approach, the translation is broken up into multiple independent programs each of which performs a small part of the overall translation. The main difference is that the first program will use the grammar for the input language and only provide overrides for the changes it will make. The second program will use the overrides for the first program and adds several overrides of its own. At some point, the number of overrides becomes a significant overhead. At this time an intermediate base grammar is written which excludes the features of the input language that have already been translated and includes the features of the target language that has been used. The next several passes apply changes to this intermediate grammar, yielding a new intermediate grammar. One initial prototype for translating COBOL to Java had seventy–two passes.

The technique was later modified [10] and used to transform the graphs generated by the GNU C compiler to graphs compatible with the Datrix [1] schema. In this case, a single mixed schema (graph equivalent of grammar), and multiple independent passes were used.

4.5 Markup

For several transformation projects we found it convenient to separate the identification of the code features to be transformed from the actual transformation. For example, in LS/2000 [8], the identification of Y2K sensitive code was separated from the actual remediation of the code. The identification phase was very aggressive, since it was important that no Y2K bugs escape identification, and a reasonable number of false positives (i.e. Y2K safe code) could be tolerated. The transformation phase was conservative, and the client could manually fix any Y2K problems that were not transformed as long as they were identified. It also meant that any of the false positives identified would not be remediated and the client would not have to undo any of the changes made by the tool. The identification program communicated through the use of markup.

Figure 10 shows a markup of code identifying a condition that leads to an abnormal termination. Such a markup is useful when tracking down unexpected errors.

In TXL, markup is accomplished in two ways. Figure 10 shows an example of grammar based markup. In this method, we override the base grammar to allow markup symbols on those elements of the grammar that we are interested in. Our previous example (from figure 10) required that we be able to identify expressions of interest. The right hand side of the figure shows a sample rule that might use the markup. The skipping statement in TXL prohibits the rule from searching inside of subtrees rooted at the given non-terminal. In this case, the skipping statement prevents the rule from going into an infinite loop.

```
f = fopen(filename, "r");
if (<ERROR_CONDITION>f == NULL</ERROR_CONDITION>) {
    fprintf(stderr,"could not open %s for input\n",filename)
    exit(ENOFILE)
}
```

Figure 10. Example code markup

```
include "C.Grammar"

define startmark
    < [id] >
end  define

define endmark
    </[id]>
end  define

redefine expression
      ...
    | [startmark][expression][endmark]
end  redefine
```

```
rule annotateExpression
    skipping [expression]
    replace [expression]
        E [expression]
    where
        E [meetsSomeCondition]
    by
        <interesting>E</interesting>
end  rule
```

Figure 11. Grammar and rules for markup

The rule replaces an expression that meets some condition with a marked up expression containing the same expression. By prohibiting the rule from matching inside of expressions, the new subexpression is not matched again.

The information used to identify which features to markup (e.g. which expression) can come from several places. It can come from analysis of other features in the program, or it can be communicated in fact from from global analysis done over several programs [8]. This technique proved to be so useful, that it was implemented in a higher level language, HSML [6].

Markup was one of the reasons that the polymorphic non-terminal [any] was introduced into TXL. Figure 12 shows a modified version of figure 11, based on the TXL manual [5]. Instead of overriding expression to permit markup, the [any] token is used to indicate that any non-terminal is permitted as part of markup.

There are two changes to the rule annotateExpression. The first is that it calls the function doMarkup to add the markup to the input. The second is that the skipping clause now indicates that the non-terminal markup is not to be traversed when looking for a match.

The function doMarkup is where the [any] tag is used within the rule. As a pattern, it matches any non-terminal. In this case it matches the expression and allows us to replace the expression non-terminal with a markup non-terminal. Polymorphic rules are generally discouraged in TXL programming since they intentionally violate the type constraints imposed by the grammar.

```
include "C.Grammar"

define startmark
    < [id] >
end define

define endmark
    </[id]>
end  define

define markup
   [startmark]
     [any]
   [endmark]
end  redefine
```

```
rule annotateExpression
    skipping [markup]
    replace [expression]
      E [expression]
    where
      E [meetsSomeCondition]
    by
      E [doMarkup 'interesting]
end  rule

function doMarkup Tag [id]
  replace [any]
    Any [any]
  construct Markup [markup]
    '< Tag '> Any '</ Tag '>
  deconstruct Markup
    MarkupAny [any]
  by
    MarkupAny
end  function
```

Figure 12. Polymorphic grammar and rules for markup

5. Industrial Experience

All of these techniques were applied in the LS/2000 [8] and LS/AMT tools developed at Legasys Corp. in Kingston. Together these two tools have analyzed and transformed more than 4.5 billion lines of COBOL, PL/I and RPG. While individual projects are confidential, some of the subjects of these projects were

- Automated Language Translation (Fortran to Java, Cobol to Java)
- Automated migration from a character terminal environment to an enterprise messaging environment
- Automated migration from a character terminal environment to a three tier web based environment
- Performance analysis of a mutli-step mainframe batch program
- Analysis of decision points leading to abnormal termination.

Since these tools were being used in an industrial setting, performance of the tools was a concern. Our profiling of the time spent in these processes indicates that parsing was never a significant fraction of the time. This leads to the second maxim of TXL programming, which is that "parsing is free". As a matter of fact, significant performance gains in the runtime of our tools were experienced almost every time the rule sets were simplified using one of the techniques we have presented in this paper.

6. Conclusions

TXL's flexible grammar definition capability and efficient parser make it easy for programmers to customize a grammar to the problem. This leads to a new paradigm for programming in rule based systems. The grammar system in TXL is flexible enough that grammar changes are as likely to be written as rules when solving a source code analysis or manipulation problem. These techniques are not limited to TXL, and can be used in any term rewriting system that uses an underlying grammar to guide the rewriting.

As rewrite systems such as TXL and ASF+SDF are used for more industrial projects involving analysis and transformations of large legacy software systems, techniques that make the best use of the strengths of these systems becomes more important.

Bibliography

[1] Bell Canada, *Datrix Abstract Semantic Graph: Reference Manual, version 1.4*, Bell Canada Inc., Montreal Canada, May 01, 2000.

[2] vand den Brand, M., Sellink, A., Verhoef, C., "Current Parsing Techniques in Software Renovation Considered Harmful", *Proc. 6th International Workshop on Program Comprehension (IWPC 98)*, Ischia, Italy, June 1998, pp. 108–117.

[3] van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, C., and Visser, J., "The ASF+SDF Meta-Environment: a component-based language development environment", *Compiler Construction 2001 (CC 2001), Lecture Notes in Computer Sicence,* R. Wilhelm, ed., Vol 1827, Springer Verlag, 2001, pp. 365–370.

[4] Cordy, J.R., Halpern, C.D., Promislow, E., "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Languages*, 16(1), January 1991, pp. 97-107.

[5] Cordy, J.R., Carmichael, I.H. and Halliay, R., *The TXL Programming Language - Version 10*, Queen's University at Kingston and Legasys Corporation, Kingston, January 2000 (65 pp).

[6] Cordy, J., Schneider, K., Dean, T., Malton, A., "HSML: Design Directed Source Code Hot Spots", *Proc. 9th International Workshop on Program Comprehension*, Toronto, Canada, May 2001, pp. 145–154.

[7] Cordy, J., Dean, T., Malton, A., Schneider, K., "Software Engineering by Source Transformation – Experience with TXL", *Proc 1st International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001, pp. 168–178.

[8] Dean, T., Cordy, J., Schneider, K., Malton, A., "Experience Using Design Recovery Techniques to Transform Legacy Systems", *Proc. ICSM 2001 - IEEE International Conference on Software Maintenance*, Florence, November 2001, pp. 622-631.

[9] Dean, T. Cordy, J., Schneider, K., Malton, A., "Unique Naming In Reverse Engineering", in preparation.

[10] Dean, T., Malton, A., Holt, R., "Union Schemas as a Basis for a C++ Extractor", *Proc 8th Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001, pp. 59–67.

[11] Lamb, D., Schneider, K., "Formalization of Information Hiding Design Methods", *Proc. CASCON '92, IBM Center for Advanced Studies Conference*, Toronto, Canada, November, 1992, pp. 201-214.

[12] Lämmel, R., Verhoef, C., "Semi-automatic Grammar Recovery", *Software Practice & Experience*, 31(15), December 2001, pp. 1395-1438.

[13] Malton, A.J., Schneider, K.A., Cordy, J.R., Dean, T.R. et al., "Processing Software Source Text in

Automated Design Recovery and Transformation", *Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension*, Toronto, May 2001, pp. 127-134.

[14] Müller, H., Klashinsky, K., "Rigi – A System for Programming-in-the-Large", *Proc. 10th International Conference on Software Engineering (ICSE 88)*, pp. 80-86.

[15] Neighbors, J., The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, 10(5), September 1984, pp. 564-574.

[16] Sellink, A., Verhoef, C., "An Architecture for Automated Software Maintenance", *Proc. 7th International Workshop on Program Comprehension (IWPC 99)*, Pittsburgh, Pennsylvania, May 1999, pp. 38–48.

[17] Sellink, A., Verhoef, C., "Native Patterns", *Proc. 5th Working Conference on Reverse Engineering*, Honolulu, Hawaii, October 1998 1998, pp. 89-103.

[18] Visser, Eelco, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System description of Stratego 0.5", *Rewriting techniques and Applications (RTA '01), Lecture Notes in Computer Science*, A Middeldorp, ed. Springer-Verlag, May 2001, pp. 357–361.

[19] Reasoning Systems, *Refine Uses Manual*, Palo Alto, California, 1992.

[20] Lethbridge,T., Plödereder, E., Tichelaar, S., Riva, C., Linos, P., *The Dagstuhl Middle Model (DMM)*, Version 0.003, June 6, 2001.