# Source Transformation, Analysis and Generation in TXL

James R. Cordy

School of Computing, Queen's University, Kingston, Canada

cordy@cs.queensu.ca

## Abstract

The TXL transformation framework has been widely used in practical source transformation tasks in industry and academia for many years. At the core of the framework is the TXL language, a functional programming language specifically designed for expressing source transformation tasks. TXL programs are self-contained, specifying and implementing all aspects of parsing, pattern matching, transformation rules, application strategies and unparsing in a single uniform notation with no dependence on other tools or technologies. Programs are directly interpreted by the TXL processor without any compile or build step, making it particularly well suited to rapid turnaround, test-driven development. In this paper we provide a practical introduction to using TXL in rapidly developing source transformations from concrete examples, and review experience in applying TXL to a number of practical large scale applications in source code analysis, renovation and migration.

*Categories and Subject Descriptors*   D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming;   D.1.2 [*Programming Techniques*]: Automatic Programming - *Automating analysis of algorithms, Program Modification, Program Synthesis, Program Transformations*;   D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement - *Restructuring, reverse engineering, and reengineering*;   D.3.3 [*Programming Languages*]: Language Constructs and Features - *Patterns*;   D.3.4 [*Programming Languages*]: Processors - *Interpreters, Parsing, Preprocessors, Translator writing systems and compiler generators*

*General Terms*   Algorithms, Design, Languages

*Keywords*   TXL, source transformation, translators, term rewriting, rule-based programming, rapid prototyping, test-driven development, software analysis, migration, re-engineering

## 1.   Introduction

TXL [7] is a programming language and rapid prototyping system specifically designed for implementing source transformation tasks. Originally designed in 1985 to support experiments in programming language design [10], it has matured into a general purpose source transformation solution that has been used in a wide range of practical applications, including applications in programming languages, software engineering, database technology and artificial intelligence.
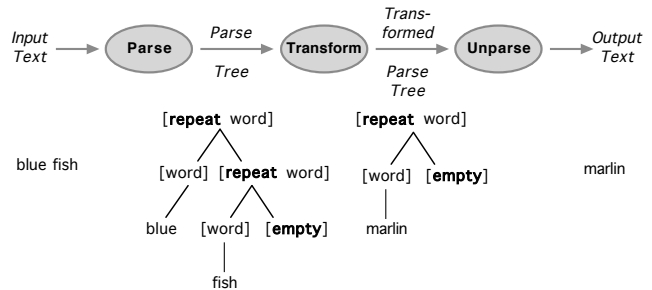
**Figure 1.** The TXL Paradigm

Other papers describe the history and design principles of TXL [7], its formal semantics [23] and its application to software engineering [12], artificial intelligence [32] and the semantic web [21] in some detail. In this paper we concentrate on the practice of developing solutions in TXL using its native test-driven development paradigm, and describe how this paradigm has been applied in a number of applications. We begin with a short introduction to the TXL language.

## 2.   The TXL Programming Language

The TXL paradigm consists of parsing the input text into a tree according to a grammar, transforming the tree to create a new output parse tree, and unparsing the new tree to create the output text (Figure 1). Grammars and transformation rules are specified in the TXL programming language, which is directly interpreted by the TXL processor (using an internal tree-structured bytecode for efficiency).

TXL programs are self-contained, in the sense that they depend on no other tools or technologies, and environment independent, so they are easily run on any platform. TXL is not itself intended to be a complete transformation environment or workbench, although it can be used as such directly from the command line. Rather, it is intended to serve as the engine embedded in such frameworks. Because they depend only on the standard input, output and error streams, TXL programs are easily configured as pipe-and-filters, live servers, embedded transformers and interactive applications as well as standalone batch transformation commands.

TXL programs normally specify a *base grammar*, which specifies the syntactic forms of the input structure, an optional set of *grammar overrides*, which modify the grammar to include new forms to be used or output by the transformation, and a rooted set of *transformation rules*, that specify how input structures are to be transformed to the intended output (Figure 2). The base grammar defines the lexical forms (tokens) and syntactic forms (nonterminal types) of the input language. The optional grammar overrides extend or modify the lexical and syntactic forms of the grammar to allow for the output and intermediate forms of the transformation.
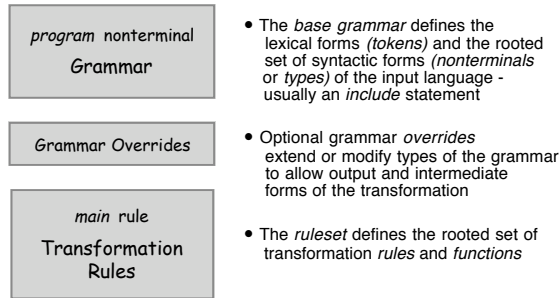
**Figure 2.** Anatomy of a TXL Program

```
tokens
    hexnumber  "0[Xx][\dABCDEFabcdef]+"
end tokens

% keywords of standard Pascal
keys
    program procedure function
    repeat until for while do begin 'end
end keys

compounds
    :=  <=  >=  ->  <->  '%=
end compounds

comments
    /*  */
    //
end comments
```

**Figure 3.** Specification of Lexical Forms

The ruleset defines the rooted set of transformation functions and rules to be applied.

## 2.1 Parsing and Grammars

TXL does not use a separate grammar and parsing technology, rather parser specification is an inherent part of the TXL program itself. As a result the TXL programmer has direct control over the interpretation of the grammar, and can easily modify parsing to suit the application. As in most other tools, specification of the language structure is in two parts: lexical forms and context-free syntactic forms.

Lexical forms describe how the input text is to be divided into the indivisible basic symbols (terminal symbols or *tokens*) of the input language. These form the basic types of the TXL program. Common basic lexical forms of the C / Java class of languages, such as identifiers, integer and floating point numbers, strings and character literals are predefined, and are often sufficient for the first version of a transformation. Other lexical forms of the input language are specified using regular expressions in the *tokens* and *compounds* statements and keywords can be distinguished from identifiers using the *keys* statement (Figure 3). Commenting conventions are specified using the *comments* statement (Figure 3). Comments, spaces and newlines are by default ignored but may optionally be explicitly parsed and transformed if desired.

Syntactic forms specify how sequences of input tokens are to be grouped into the structures of the input language. These form the structured types of the TXL program, and are specified using an (almost) unrestricted ambiguous ordered context free grammar in a BNF-like notation (Figure 4). In essence, every TXL program defines its own type system in this way, and the TXL program is strongly and statically typed in the type system specified by the grammar.

TXL inverts the usual BNF convention of quoting terminal symbols and instead uses unquoted bare terminal symbols while en-

```
define program              % goal symbol of input
    [expression]
end define

define expression
    [term]
  | [expression] + [term]   % arbitrary recursion &
  | [expression] - [term]   %  ambiguity supported
end define

define term                     define primary
    [primary]                       [number]
  | [term] * [primary]            | ( [expression] )
  | [term] / [primary]          end define
end define
```

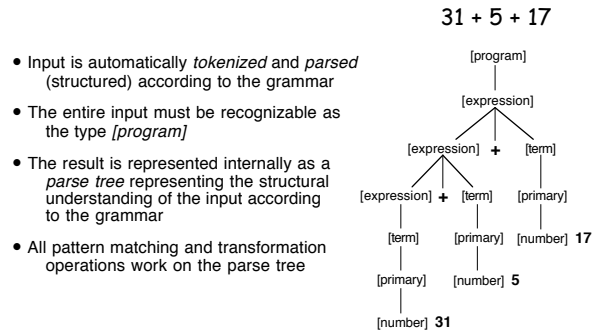**Figure 4.** Specification of Syntactic Forms (Grammar)



**Figure 5.** A TXL Parse Tree

closing nonterminal type names in square brackets, as in *[expression]*. This notation is used consistently throughout TXL, in grammars, patterns and replacements to give the language an example-like style. Each nonterminal is specified using a *define* statement, and the special nonterminal type *[program]* (the goal symbol of the grammar) describes the structure of the entire input. The usual standard set of BNF extensions is supported, including sequences *[repeat X]*, comma-separated lists *[list X]* and optional items *[opt X]*. Using these extended forms in preference to their recursive counterparts makes TXL grammars most convenient and natural for specifying patterns.

When a TXL program is run, its input is automatically tokenized and parsed (structured) according to the grammar using a direct top-down full backtracking functional interpretation, with heuristics to resolve left recursion. The entire input must be recognizable as the type *[program]*, and the result is represented internally as a parse tree representing the structural understanding of the program according to the grammar (Figure 5). All pattern matching and transformation acts on these parse trees.

The base grammar for the syntax of the input language is normally kept in a separate grammar file which is rarely if ever changed, and is included in the TXL program using an *include* statement. Input language variants and extra output or intermediate forms are added to the base grammar using grammar overrides, which modify or extend the base grammar's lexical and syntactic forms (Figure 6). In the figure, the notation "..." is not an elision, rather it is part of the TXL grammar notation meaning "whatever *[statement]* was before". In the example, the new form following the bar is appended to the definition of *[statement]* as the last alternative.

## 2.2 Rules and Functions

The actual input to output transformation is specified in TXL using a rooted set of transformation *rules* and *functions*. Each transformation rule specifies a *target type* to the transformed, a *pattern* (an example-like form of the particular instance of the type we are in-

```
% Base grammar for C++
include "Cpp.Grammar"

% Override to allow XML markup on statements
redefine statement
    ...                             % original forms
  | <[id]> [statement] </[id]>      % add new form
end redefine
```

**Figure 6.** Override to Allow XML Markup of Statements

```
% replace every 1+1 expression by 2
rule addOnePlusOne
    replace [expression]    % target structure type
        1 + 1               % pattern to search for
    by
        2                   % replacement to make
end rule
```

**Figure 7.** Simple Example TXL Rule

```
rule optimizeAddZero
    replace [expression]
        N1 [number] + 0
    by
        N1
end rule
```

**Figure 8.** Rule Using a Pattern Variable

```
rule resolveAdditions
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [add N2]
end rule
```

**Figure 9.** Rule Using a Subrule [add]

```
function main
    replace [program]
        EntireInput [program]
    by
        EntireInput [resolveAdditions]
                    [resolveSubtractions]
                    [resolveMultiplys]
                    [resolveDivisions]
end function
```

**Figure 10.** Example Main Rule

```
function resolveEntireAdditionExpression
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [add N2]
end function
```

**Figure 11.** A Simple Example TXL Function

```
function resolveFirstAdditionExpression
    replace * [expression]
        N1 [number] + N2 [number]
    by
        N1 [add N2]
end function
```

**Figure 12.** Simple Example TXL Function

terested in replacing), and a *replacement* (an example-like form of the result we would like when we replace such an instance). An example for recognizing and replacing expressions of the form *1+1* is shown in Figure 7.

TXL rules are both strongly and statically typed - that is, the replacement must be an instance of the same nonterminal type as the pattern. When the original target type does not include the form we want as replacement, grammar overrides are added to explicitly allow for it.

The pattern of a rule is an augmented source text example of the instances we want to replace, expressed in concrete syntax. Patterns consist of *tokens* (terminal symbols, such as *2* or *+*, which represent themselves) and *variables* (placeholders which match and bind to a name any instance of a given nonterminal type) (Figure 8). A pattern must be parsable as an instance (sentential form) of the target type.

When a pattern is matched, variable names are bound to the instances of their types in the match. Bound variable names can be used in the rule's replacement to copy their bound instance into the result. In the case of Figure 8, the *[number]* bound to *N1* in each match of the pattern is copied into the corresponding result. The replacement is also expressed as an augmented source text example of the desired result, consisting of tokens and variable references. Since rules are strongly typed, it must also be parsable as an instance of the target type.

References to variables in a replacement can optionally be transformed by subrules (other transformation rules and functions), which transform only the copy of the variable's bound instance in the result (9). Subrules are applied to variable references using square bracket notation *X[f]*, which in standard functional notation would be written *f(X)*. Because all rules are strongly typed, a variable reference transformed by a subrule retains its type, guaranteeing a well formed result in the replacement. The result of a subrule application can itself be transformed by other subrules, as in *X[f][g]*, which denotes the composition *g(f(X))*.

When a rule is applied to a variable reference, we say that the copied variable's bound instance is the rule's *scope*. A rule application only transforms inside the scope that it is applied to. The distinguished rule named *main* is automatically applied with the entire input parse tree as its scope – other rules must be explicitly applied as subrules in order to have any effect. Often the main rule is simply a function that applies other rules (Figure 10).

TXL actually has two kinds of transformation rules, called *rules* and *functions*. The two are distinguished by whether they should transform one (for functions) or many (for rules) occurrences of their pattern. In all cases, TXL rules and functions are *total* – that is, if their pattern does not match, the defined result is their unchanged original scope.

By default, TXL *functions* do not search, but attempt to match only their entire scope to their pattern, transforming it if it matches (Figure 11). *Searching functions*, denoted by "*replace \**", search to find and transform the first instance of their pattern in their scope, but do not automatically reapply (Figure 12). *Rules* are just a shorthand for the compositional closure of the corresponding searching function. That is, a rule repeatedly searches its scope for the first instance of the target type that matches its pattern, transforms it in place to yield an entire new scope, and then reapplies itself to that entire new scope and so on until no more matches are found.

Rules and functions may be passed bound variables as *parameters*, which allow the values of variables captured in the pattern of an applying rule to the formal parameters of a subrule (Figure 13). Rule parameters can be used to contextualize sub-transformations as well as to build transformed results out of many disjoint parts of the original scope.

### 2.3 Patterns and Replacements

Both patterns and replacements are parsed using the same grammar as the input to create *pattern tree / replacement tree* pairs (Figure 14). TXL's ordered grammar interpretation insures that ambiguities are always resolved in the same way in both patterns and inputs,

```
rule resolveConstants
   replace [repeat statement]
      const C [id] = V [expression];
      RestOfScope [repeat statement]
   by
      RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [expression]
   replace [primary]
      ConstName
   by
      ( Value )
end rule
```

**Figure 13.** Using Rule Parameters in a Subrule

```
rule resolveAdditions
   replace [expression]
      N1[number] + N2[number]
   by
      N1 [+ N2]
end rule
```
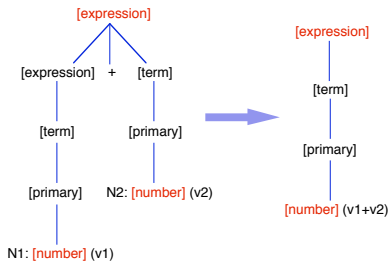


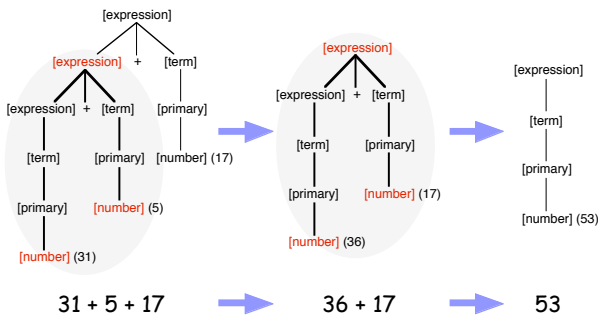**Figure 14.** Parse Tree Form of Patterns and Replacements



**Figure 15.** Rule Application by Tree Matching

reducing pattern matching a straightforward and efficient typed tree-matching problem.

Rules are implemented by searching the scope parse tree for tree pattern matches of the pattern tree, and replacing instances with corresponding instantiations of the replacement tree with subtrees copied for bound variable references. Each such replacement results in a new scope tree, which is then searched again for another match, and so on until no more matches can be found (Figure 15).

Patterns may refer to previously bound variables later in a pattern (i.e., may use nonlinear pattern matching [27]). This effectively parameterizes the the pattern with a copy of the bound variable, in order to specify that two parts of the matching instance must be the same in order to have a match (Figure 16). Patterns can be similarly paramterized by formal parameters of the rule or function or any other bound variables in order to specify that matching instances must contain an exact copy of the variable's bound instance at that point in the pattern. In TXL, references to a bound variable always mean a copy of the bound instance, no matter what the context.

```
rule optimizeDoubles
   replace [expression]
      E [term] + E
   by
      2 * E
end rule
```

**Figure 16.** Nonlinear Pattern Matching

```
rule optimizeFalseWhiles
   replace [repeat statement]
      WhileStatement [while_statement] ;
      RestOfStatements [repeat statement]
   deconstruct * [condition_expression] WhileStatement
      WhileCond [condition_expression]
   deconstruct WhileCond
      false
   by
      RestOfStatements
end rule
```

**Figure 17.** Using Deconstructors to Refine the Pattern

```
rule vectorizeScalarAssignments
   replace [repeat statement]
      V1 [variable] := E1 [expression];
      V2 [variable] := E2 [expression];
      RestOfScope [repeat statement]
   where not
      E2 [references V1]
   where not
      E1 [references V2]
   by
      < V1,V2 > := < E1,E2 > ;
      RestOfScope
end rule

function references V [variable]
   deconstruct * [id] V
      Vid [id]
   match * [id]
      Vid
end function
```

**Figure 18.** Using Where Clauses to Constrain the Match

(Although obviously the TXL interpreter maximally optimizes the implementation of this meaning to avoid copying whenever possible).

Patterns can be piecewise refined using *deconstruct* clauses in order to allow for more complex patterns (Figure 17). Deconstructors specify that the deconstructed variable's bound instance must match the given pattern - if not, the rule's entire pattern match fails and the search continues for another match. Deconstructors act like functions - that is, the variable's entire bound instance must match the deconstructor's pattern. As with functions, there is also a *searching deconstruct* ("deconstruct *") that searches within the variable's bound instance for a match.

Pattern matches can also be constrained using *where* clauses (Figure 18). Where clauses use a special kind of rule called a *condition* rule. Condition rules may be built-in, for example the ordering functions [<] and [>], implementing semantic less than and greater than respectively, or user-coded as subrules. Condition rules have only a (possibly very complex) pattern, but no replacement. They simply either succeed or fail to match their pattern, and the *where* clause invoking them succeeds or fails correspondingly. Because they are themselves TXL rules, they may use additional *deconstructs*, *constructs*, subrules, *where* clauses and so on, allowing for arbitrary computation in guards, including tests involving global or external information (Section 2.4).

Replacements can also be piecewise refined, using *construct* clauses to build results from several independent pieces (Figure 19). Constructors allow partial results to be bound to new variables,

```
rule addToSortedSequence NewNum [number]
   replace [repeat number]
      OldSortedSequence [repeat number]
   construct NewUnsortedSequence [repeat number]
      NewNum OldSortedSequence
   by
      NewUnsortedSequence [sortFirstIntoPlace]
end rule
```

**Figure 19.** Using Constructors to Build the Replacement

```
define symbol_table_entry
   [id] [type]
end define

function main
   export SymbolTable [repeat symbol_table_entry]
      % initially empty, filled in by subrules
   . . .
end function

rule translatePlusEqualIfString
   replace [statement]
      S1 [id] += E [expression];
   % This rule only applies if S1 is of type string
   import SymbolTable [repeat symbol_table_entry]
   deconstruct * [symbol_table_entry] SymbolTable
      S1 string
   by
      concatn (S1, E, 256);
end rule
```

**Figure 20.** Using a Global Table

allowing sburules to further transform them. Thus a rule can build a complex replacement from many different constructed pieces transformed by different sets of subrules.

## 2.4 Global Tables

Complex transformation tasks often use a symbol table to collect information which can then be used as a reference when implementing transformation rules. TXL provides *global variables* for this purpose. Globals in TXL are modelled after the Linda *blackboard* style of message passing[16]. In this style, bound local variables are *exported* to the global scope by a rule or function for later *import* by some other rule or function. Exported variables may be of any nonterminal type, including new types not related to the main grammar.

Globals have a great many uses in transformations, but the most common is the original use: symbol tables. Symbol tables in TXL are typically structured as an associative lookup table consisting of a sequence of *(key, information)* pairs. Both the key and the information can be of any nonterminal type, including new types defined solely for the purpose. Often the key is of type *[id]* (i.e., an identifier). TXL *deconstructs* are used to associatively look up the information given the key (Figure 20). Because they use pattern matching, table lookups are also two-way; if one wants to know the key associated with some information, the deconstructor can just as easily pattern match in the reverse direction.

## 3. Developing TXL Programs

We've now seen most of the basic features of TXL. Like most functional languages, TXL builds its power from the combination of this small set of general concepts rather than a large collection of more specific features. And like other functional languages, its power and expressiveness are not obvious from features themselves – rather one must look at the paradigms for using them. In this section we introduce three of the most basic paradigms for using TXL to solve practical problems: test-driven development, cascaded transformation, and aspect transformation.

```
include "Cobol.Grm"

redefine add_statement
      ...
   |  [pli_assignment]
end redefine

define pli_assignment
   [operand] = [expression];
end define

function main
   replace [program]
      OriginalCobol [program]
   by
      OriginalCobol [convertAddStatements]
                    [convertIfStatements]
                       (and so on)
end function
```

**Figure 21.** Main Program for Conversion of Cobol to PL/I

```
rule convertAddIJK
   replace [statement]
      ADD I TO J GIVING K    % COBOL
   by
      K = I + J;             % PL/I
end rule
```

**Figure 22.** Beginning with a Concrete Example

### 3.1 Test Driven Development

TXL is well suited to the test-driven development style of *eXtreme Programming* (XP) [3]. TXL applications are most easily developed beginning with an explicit set of test cases (example input-output pairs), which serve as the specification of the transformation. The transformation is then developed incrementally, as a sequence of successive approximations to the final result. TXL's interpreter is designed for rapid prototyping and fast turnaround, so partial transformations can be run against the test case inputs as we go in order to keep track of progress during development and catch problems early. In the usual XP development style, we leave tuning until later and concentrate on the simplest possible transformation rules to achieve the intended result. TXL programs optimize well once a correct solution has been implemented.

We'll demonstrate the process of developing a new transformation using a realistic but toy example - translation of Cobol *ADD* statements to PL/I code. The other basic rules of a Cobol to PL/I transformation would be developed similarly. For the purposes of the demonstration we'll ignore a number of details in this problem in order to concentrate on the method itself.

The transformation we have in mind takes Cobol as input, and therefore includes the Cobol base grammar. The first thing we need to do is to allow for the desired output by overriding the grammar. In Figure 21 we have done this by allowing an *[add_statement]* to be a *[pli_assignment]*. Rather than include the entire PL/I grammar, we have chosen to work incrementally, adding output forms to the grammar only as they are needed.

For the Cobol ADD statement to PL/I rule, we begin with a single explicit example pattern and replacement to cover exactly one particular statement in the test input, and generalize from there. Figure 22 shows this first approximation of our rule for transforming Cobol *ADD* statements to PL/I. It recognizes and converts only and exactly the Cobol statement "*ADD I TO J GIVING K*" and replaces it with the PL/I statement "*K = I + J;*". This rule only works if the variables in the statement have the exact names given. We test this rule on our input, see that the statement in question is transformed, and then move on to generalize it.

The second step involves generalizing by introducing TXL variables for the explicit names in the pattern, allowing for arbitrary names rather than just *I, J* and *K.*. We could simply allow for arbitrary identifiers (type *[id]* in TXL), but in the Cobol grammar it

```
rule convertAddGiving
   replace [statement]
      ADD I [operand] TO J [operand] GIVING K [operand]
   by
      K = I + J;
end rule
```

**Figure 23.** Generalizing Using Pattern Variables

```
rule convertAddNoGiving
   replace [statement]
      ADD I [operand] TO J [operand]
   by
      J = J + I;
end rule
```

**Figure 24.** Specializing to Other Cases

```
rule convertAdds
   replace $ [statement]                    % ($ = nonrecursive)
      AddStatement [add_statement]
   by
      AddStatement [convertAddOne]       % ADD 1 to Z
                   [convertAddNoGiving]  % ADD Y TO Z
                   [convertAddGiving]    % ADD X TO Y GIVING Z
                   [checkAddConverted]   % check we translated
end rule
```

**Figure 25.** Abstracting, Integrating and Prioritizing Cases

makes more sense to move up to the more general type *[operand]*, which allows for both identifiers and literal constant values (so that we match "*ADD 10 to X GIVING Y*" for example). Figure 23 shows this generalization. Our rule should now recognize all statements of the form "*ADD I TO J GIVING K*" for any arbitrary operands. Once again we run the rule on our test input to see that it is still working and correctly transforms all such statements.

In the next step we specialize by identifying, testing and generalizing special cases in the same way. For example, in Cobol the *GIVING* clause is optional. Figure 24 shows a specialization of our rule for that case. Again, we test each case as we go along to make sure that things are working properly. We continue in this way, developing separate rules for each special case, until we have a complete set.

Once we have developed working solutions for the general and each of the special cases, we are ready to integrate them by abstracting and prioritizing cases (Figure 25). Ruleset abstractions are built using a higher level rule to invoke each of the cases as a subrule. The order in which they are composed is important, because special cases such as *[convertAddOne]* (intended to optimize increments) may also be matched by *[convertAddNoGiving]*. Recall that rules that do not match simply return their original scope as result, so composed subrules simply pass on those cases that they do not match. To be certain that some case matched each instance, we add a special rule *[checkAddConverted]* that simply matches if the statement is still in Cobol once all other subrules have tried to transform it, and issues an error message if so. Once again, we test our entire ruleset on the test inputs.

### 3.2 Cascaded Transformation

The style of starting with partial solutions and gradually refining and combining them to make a sequence of successive approximations to the intended final result is typical of TXL development at every level. An entire Cobol to PL/I transformation would consist of developing translations for each of the statements and other forms of Cobol in similar fashion, largely independently of each other. The main transformation rule would then apply each of these rulesets in succession (possibly recursively). After each ruleset is applied, one more of the forms of the input language has been transformed to the output language, while other forms still remain
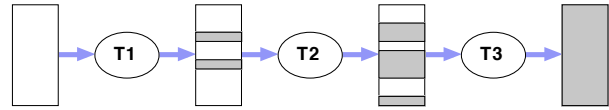


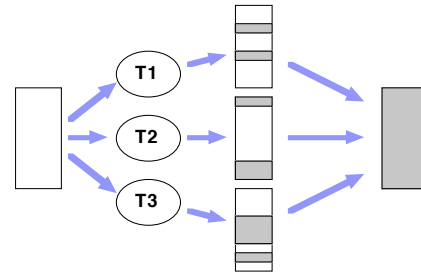**Figure 26.** Generic Cascaded Transformation Architecture



**Figure 27.** Generic Aspect Transformation Architecture

untranslated. The intermediate results are in a kind of *hybrid language* where some forms are in the original Cobol and some in the target PL/I. The grammar of this intermediate language is the base grammar as modified by the overrides for the translated forms.

In order to organize large scale transformations and make them easier to develop and maintain, TXL programmers normally separate them into a cascaded set of separate TXL programs, each of which attacks only one issue or set of issues. The output of each stage returns to a text file in the hybrid language which is then reparsed using the overridden grammar to the next stage. In this way, each stage can be developed and maintained independently, and each has a precise, easy to understand specification described by a test set that can be created (by hand) even in the absence of previous stages.

While this *cascaded transformation* style is most obviously applicable to language translation tasks, it is used in virtually all large-scale applications of TXL. A typical overall high-level application architecture is shown in Figure 26. Each stage in the sequence handles another aspect of the original input, until eventually everything has been handled. Often the stages in the sequence are not simply disjoint transformations, but rather depend on the analysis and transformation done by previous phases to support the next. For example, in software maintenance tasks, an early phase typically implements unique naming of variables [18] so that later phases can more efficiently implement scope-sensitive transformations.

In addition to suiting implementation as a sequence of TXL transformations, the cascaded style offers significant software engineering benefits. Interfaces between phases of an overall transformation are explicitly documented by the intermediate grammars (base grammar plus overrides) that describe the intermediate hybrid languages, and each subtransformation can therefore be specified, developed, tested and maintained independently of the others, allowing for parallel development and validation. Because intermediate forms are in explicit hybrid language text, tests for each phase can be crafted by hand even before previous phases are implemented.

A prototype translator from COBOL to Java developed at Legasys Corporation is perhaps the extreme demonstration of the advantages of the cascaded transformation paradigm. After two months of planning by the lead architect, a prototype translator consisting of 72 cascaded phases was built in less than two months by six developers, only three of whom were assigned to the project full time. Each phase intentionally handled only one small part

```
rule transformClasses
    replace [repeat declaration_or_statement]
        type ClassId [id] :
            class
                Imports [repeat import_list]
                Exports [repeat export_list]
                Fields [repeat variable_declaration]
                Methods [repeat procedure_delaration]
            'end ClassId
        RestOfScope [repeat declaration_or_statement]
    by
        module ClassId :
            Imports
            export DataRecord
            Exports
            type DataRecord:
                record
                    Fields
                'end record
            Methods [fixFieldReferences each Fields]
                    [makeConstructorMethod]
                    [addObjectParameterToMethods]
        'end ClassId
        RestOfScope [transformClassReferences ClassId]
    end rule
```

**Figure 28.** Rule to Implement Object Classes by Transformation to Turing Modules (Adapted from [9])

of the overall transformation. The parallel development provided by this approach paid large dividends in time and effort, quickly demonstrating automated translation of a 60,000 line Cobol application as part of the bid for a multi-million line contract.

### 3.3   Aspect Transformation

Many applications require radically different transformations to be made to different parts of the input, or have outputs that require a number of different transformations of the same or overlapping parts of the input to generate the output. For example, this is typically the case when the representation of particular aspects of the input must be "*woven*" quite differently in the output language than in the input one.

Such nonlinear transformations are handled in TXL using a different transformation architecture, in which the same entire input is fed to several different transformations, each of which transforms one aspect of the input to the output. The results are then reassembled using a final transformation or weaver. Figure 27 demonstrates this transformation architecture. Once again, the style has real software engineering advantages due to its clear separation of concerns. Transformations for the different aspects can be developed in parallel and maintained largely independently.

This *aspect transformation* architecture is also very common in production TXL applications, both alone and in combination with the cascaded paradigm. For example, design recovery applications typically use a set of parallel aspect transformations to recognize different design aspects of the program. TXL programmers find that designing applications as cooperating sets of independent subtransformations yields both the obvious engineering advantages and technical advantages such as the ability to use different grammar overrides tailored to each transformation (called *grammar programming* or *agile parsing* [15]).

One can of course find both of these decomposition styles in every software architecture book. What is different here is their fine-grained application to software transformation tasks. Transformations that seem complex or challenging as single programs are often straightforward when decomposed into cascaded or aspect steps.

## 4.   Practical Experiences with TXL

TXL is a mature source transformation technology and has been used in a wide range of applications in academia and industry over the past 20 years. In this section we outline a number of typical applications and how they have exploited the test-driven, cascaded development paradigm.
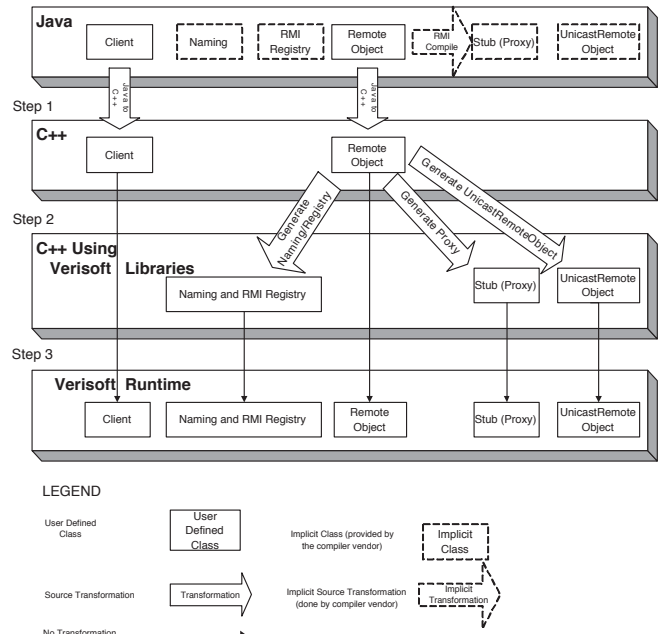


**Figure 29.** Java RMI to C++ / VeriSoft Transformation Architecture (Adapted from [6])

### 4.1   Language Dialects

TXL's original design goal was to support experiments in programming language design in the context of the Turing programming language project [20]. The idea was that proposed new language features and variants could be implemented by source transformation to the original language in order to provide a rapid prototype of the new language that could be used and experimented with right away. By using TXL source transformations instead of modifying the Turing language implementation directly, proposals could typically be evaluated in a day or two rather than several weeks.

In the late 1980's many variants of the Turing language were prototyped in this way using TXL, including competing proposals for the addition of object-oriented features to Turing [9]. Figure 28 shows one of the main rules in the TXL rapid prototype of a Turing dialect very close to the eventual winner, known as Object-Oriented Turing. Even at this early stage in TXL, we can see the clear influence of test-driven development in the generalized example style of the pattern and replacement in the rule.

### 4.2   Language Translation

TXL has been used in many language translation tasks, such as the Cobol to Java one mentioned in Section 3.2. Most recently it has been used to extend the capabilities of the VeriSoft modelling framework [17] to handle Java RMI (remote method invocation) programs by automatically transforming them to semantically equivalent C++ / VeriSoft programs [6].

Translation of Java RMI to C++ / VeriSoft is a nontrivial transformation task, involving both integration of widely separated information in the original Java into local contexts in C++ (*global-to-local* transformation) and separation of local information in Java into widely separated contexts in the generated C++ / VeriSoft components (*local-to-global* and *coupled* transformations). In particular, three quite separate results, the *Registry*, the *Proxy Stub* and the *Unicast Remote Object*, have wide dependency on many parts of the original program.
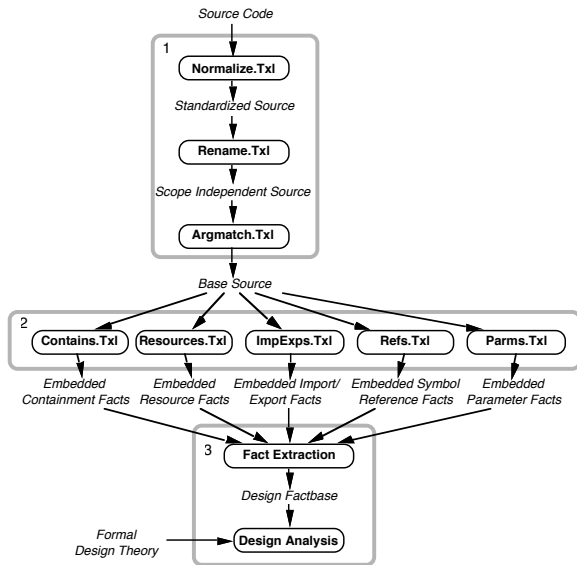
**Figure 30.** Design Recovery by Source Transformation (Adapted from [12])



**Figure 31.** Implementation of the $\mu$* Metaprogramming System (Adapted from [12])

```
\    struct {
        char *name;
        int (*addr)();
    } func[] =
    {
        $AllEntries,
        {"",0}
    };
\
    where AllEntries
        \ {$X,$Y} \ [list init]
    each function (F [id])
    where X \ "" \ [string] [" F]
    where Y \ mpro \ [id] [_ F]
\\
```

**Figure 32.** Example $\mu$* C Metaprogram (Adapted from [12])

together with the C++ transformations of the original Java RMI components to yield the final C++ / VeriSoft program.

### 4.3 Design Recovery and Analysis

The Advanced Software Design Technology (ASDT) project was a joint project involving IBM Canada and Queen's University in the early 1990's. The global goal of the project was to study ways to formalize and better maintain the relationship between design documents and actual implementation code throughout the life cycle of a software system (an approach perhaps now better known as *model-driven software engineering*). Early on it was realized that if such an approach is to be introduced into practice, it must be applicable to handle existing large scale legacy systems whose design documents have been long ago lost or outdated. Thus *design recovery*, the reverse engineering of a design database from source code artifacts, became an important practical goal of the project [22].

At the time design recovery did not at first seem like a good application for source transformation. However, it is clearly a problem well suited to generalized pattern matching, and using a three stage source annotation approach (Figure 30), a completely automated design recovery system was implemented in TXL. The approach involves several TXL transformations, beginning with a cascaded transformation to prepare the source code for easy analysis (step 1). In this phase syntactic and positional variation is reduced by reordering and simplification in order to reduce the number of cases for the static design analysis phases. Identifiers are uniquely named according to their declaration scope and all references renamed to match in order to factor out scope dependence.

The second step is an aspect transformation of the normalized source consisting of five static analysis transformations, each of which searches for a set of source patterns corresponding to a particular design relationship. Each analysis transformation annotates the source with design facts for its recognized relationships in Pro-
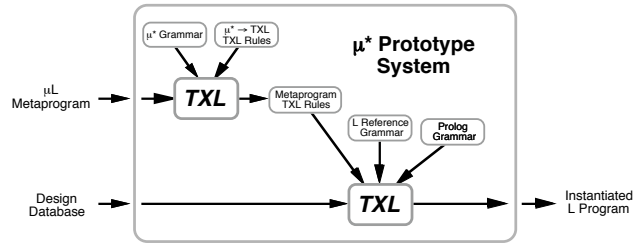
log notation. Finally, in the third step these "embedded" design facts are extracted (using *grep*) and merged to create a Prolog design database for the entire program, which can then be analyzed using Prolog or other graph-based analysis tools (step 3).

This application involves a combination of both cascaded transformation paradigm (vertically) and the aspect-oriented method (horizontally) to achieve its result. The ASDT project served as a proof of concept for design recovery using TXL, and its basic architecture has been used in a number of projects since, most spectacularly in LS/2000 (Section 4.5).

### 4.4 Generative Programming

The ASDT project also concerned itself with the other half of maintaining the relationship between design documents and implementation: the generation of original code from design documents, often called *generative programming* or *metaprogramming* [11]. In metaprogramming, the generation of code from the design model is guided by a set of code templates which are instantiated in response to queries on the design database. For example, a procedure header template may be instantiated once for each *procedure* fact in the design database.

$\mu$* (pronounced "mew star") is a family of metaprogramming languages sharing a common notation and implementation. In $\mu$*, templates are written as example source in the target programming language, which may be one of many, including C, Prolog, Pascal and others, and are instantiated under direction of metaprogramming annotations added to the template source. The metaprogramming annotations specify the design conditions under which the parts of the template apply (Figure 32).

Implementation of $\mu$* uses a two-stage source transformation system implemented in TXL (Figure 31). In the first stage, metaprograms are transformed using TXL into a TXL ruleset that implements their meaning as a source transformation of a design database represented as Prolog facts, and in the second stage this ruleset is run as a TXL source transformation of the design database for the system to be generated. In a sense, this is a second-order
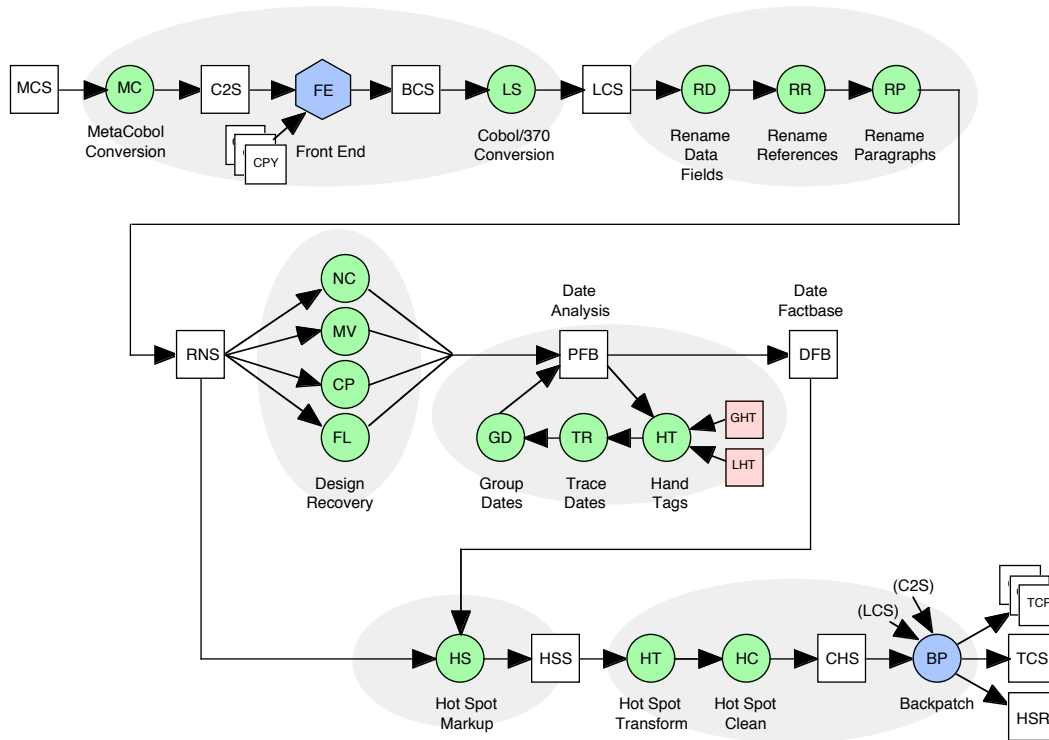
**Figure 33.** The LS/2000 Process Architecture

cascaded transformation, in which the first stage transforms its input into a second transformation.

The $\mu*$ system has been used for tasks such as generating the several kinds of C and Prolog "glue" code necessary to allow Prolog programs access to C library interfaces described by a formal interface design specification. This method was applied to generate the code to make the GL graphics library available to Prolog programmers.

### 4.5 Maintenance Hot Spots and the Year 2000 Problem

Maintenance *hot spots* [13] are a generalization of code performance hot spots to any kind of design or source code analysis activity. Sections of source code are labeled as *hot* when a design or source analysis looking for sensitivity to a particular maintenance issue, such as the Year 2000 problem, has identified them as relevant. Maintenance hot spots can be used either by human maintainers to focus their maintenance and testing efforts, or by automated reprogramming tools as targets for reprogramming templates.

LS/2000 [14] was a production software analysis system developed at Legasys Corporation to assist in the Year 2000 conversion of billions of lines of Cobol, PL/I and RPG source code. Using a variant of the ASDT design recovery technique followed by design analysis and a "*hot spot markup*" for the Year 2000 problem, LS/2000 produced hot spot reports for every module of an application that had any potential Year 2000 risks embedded in it, and automatically reprogrammed the majority of hot spots according to a large set of transformation templates. Clients of LS/2000 reported a 30-40 fold increase in Year 2000 conversion productivity using automated hot spot identification and reprogramming. Time to examine and convert a source code module of tens of thousands of lines of source code was reduced from a few hours to less than five minutes, and tested accuracy of conversion was increased from about 75% to well over 99%.

LS/2000 (Figure 33) is a classic instantiation of the generic cascaded transformation architecture of Section 3.2, involving a pipeline of successive approximations, each with a single independent specification. Like ASDT, it also exploits an aspect transformation architecture in its design recovery component, in which several transformations recognize different design aspects of the same source code artifacts. This architecture yielded enormous practical benefits in the high pressure environment of the Year 2000 project since different parts of the architecture could be corrected, refined and reengineered largely independently of one another. All phases of LS/2000 were initially implemented and debugged as TXL transformations, allowing for rapid development and deployment. As production efficiency pressures grew, some components, such as *Date Analysis*, were replaced by more efficient hand-coded C implementations, using the TXL versions as precise specifications and validation comparators.

At the core of LS/2000 were the *Hot Spot Markup* and *Hot Spot Transform* phases of the process (Figure 4.5). *Hot Spot Markup* took as input each source module of the software system being analyzed along with the set of Year 2000 date relationships inferred by design analysis for the module. In order to implement the markup process as a pure source to source transformation, the inferred date relationships were represented as Prolog source facts prepended to the module source and parsed into a TXL global table in the main rule. Potentially Year 2000 sensitive operations in the source were marked as hot by a set of TXL rules using source patterns guarded by pattern matches of the Prolog source facts in the global table. Figure 34 shows one such markup rule.

*Hot Spot Transform* then took as input the resulting marked-up source and used a set of TXL source transformation rules to search for marked hot spots that were instances of a large set of reprogramming templates based on a "windowing" solution to Year 2000. Because the *Hot Spot Markup* phase had explicated the

```
rule markupDateLessThanYYMMDD
   import DateFacts [repeat fact]
   replace $ [condition]
      LeftOperand [name] < RightOperand [name]
   deconstruct * [fact] DateFacts
      Date ( LeftOperand, "YYMMDD" )
   deconstruct * [fact] DateFacts
      Date ( RightOperand, "YYMMDD" )
   by
      {DATE-INEQUALITY-YYMMDD
        LeftOperand < RightOperand
      }DATE-INEQUALITY-YYMMDD
end rule
```

**Figure 34.** An Example LS/2000 Markup Rule

kind of potential risk in the markup label of each hot spot, these templates could be applied very efficiently. Figure 35 shows an example of a TXL hot spot transformation rule for reprogramming one kind of Year 2000 hot spot.

The hot spot transformation rules of LS/2000 were developed using test-driven development from explicit examples of each Year 2000 risk for each different date format. Example conversions were exhaustively tested to validate both that their behavior would remain consistent with the original unconverted code until the Year 2000 turnover, and to validate that (unlike the original) it would continue to behave correctly afterwards. Each of these examples was then encoded in a separate transformation rule as a TXL pattern / replacement pair, generalized and conditioned as outlined in Section 3.1. The result was a Year 2000 transformation that was highly robust and accurate.

### 4.6 Other Applications of TXL

TXL has been used in a wide range of other tasks in software engineering in addition to those above [12]. However, in recent years it has also been applied to a much wider range of applications in other areas, including recognition of hand-written mathematical fromulae [32], document analysis for the semantic web [21] and security analysis of network protocols [28].

## 5. Related Work

Many other practical source transformation systems address a similar problem domain to TXL, but each has its own particular approach. Like TXL, Stratego [30] is a programming language specifically designed for expressing source transformations. Stratego's strengths lie primarily in its emphasis on abstraction and reusability. Unlike TXL, which is designed to maximize independence between programs and provides reuse only at the whole transformation level, Stratego is designed to allow for fine-grained reuse of transformation strategies and rules, providing the ability to program generic strategies on top of pure term rewriting. Stratego/XT [31] embeds Stratego in a toolset framework to support development of complex transformation systems.

ASF+SDF [4] is a powerful general toolkit for end-to-end implementation of complex transformations and other language processing tasks. ASF+SDF's "meta-environment" [5] provides a sophisticated and powerful workbench for developing transformations. Unlike TXL, both Stratego and ASF+SDF are based on a Generalized LR (GLR) [29] parser. ANTLR [24], on the other hand, like TXL is based on generalized top-down (LL) parsing. ANTLR provides a user-configurable framework based on generalized compiler technology. APTS[26, 25] is a very general transformation system based on parse tree rewriting that is primarily aimed at program derivation. Like TXL, APTS uses nonlinear tree pattern matching, but using a bottom-up search in place of TXL's top-down search. APTS is particularly well suited to expressing constraint-based transformations.

```
rule transformLessThanYYMMDD
   replace $ [repeat statement]
      IF {DATE-INEQUALITY-YYMMDD
           LeftOperand [name] < RightOperand [name]
         }DATE-INEQUALITY-YYMMDD
      ThenStatements [repeat statement]
      OptElse [opt else_clause]
      END-IF
      MoreStatements [repeat statement]
   construct RolledLeftOperand [name]
      LeftOperand [appendName "-ROLLED"]
   construct RolledRightOperand [name]
      RightOperand [appendName "-ROLLED"]
   by
      {TRANSFORM-INSERTED-CODE
        ADD LeftOperand ROLLDIFF-YYMMDD GIVING RolledLeftOperand
        ADD LeftOperand ROLLDIFF-YYMMDD GIVING RolledRightOperand
      }TRANSFORM-INSERTED-CODE
      IF {TRANSFORMED-DATE-INEQUALITY-YYMMDD
           RolledLeftOperand < RolledRightOperand
         }TRANSFORMED-DATE-INEQUALITY-YYMMDD
      ThenStatements
      OptElse
      END-IF
      MoreStatements
end rule
```

**Figure 35.** An Example LS/2000 Transform Rule

In the commercial domain, Semantic Designs' DMS Software Reengineering Toolkit (SRT) [2] provides a framework for implementing software analysis and transformation tasks in a wide range of languages based on user-configurable, generalized compiler technology, and GrammaTech's CodeSurfer [1] and related tools can be used to provide code analysis and transformation for C and Ada.

## 6. Conclusion

This paper has provided a quick introduction to the basic ideas of TXL and its use in a few practical applications in software analysis and transformation. Obviously in such a short paper there has been no room to explore any details of semantics or implementation or to do a really thorough review of applications and technique. Details of the language itself can be found in the TXL Reference Manual [8], and its mathematical semantics is defined denotationally in terms of functional tree rewriting [23]. Many other papers on TXL and its applications are listed on the TXL website (*http://www.txl.ca*).

## Acknowledgments

## References

[1] P. Anderson and M. Zarins, "The CodeSurfer Software Understanding Platform", Proc. IEEE 13th International Workshop on Program Comprehension, 147–148 (2005).

[2] I.D. Baxter, "Design Maintenance Systems", *Communications of the ACM* **35**(4), 73–89 (1992).

[3] K. Beck, *Extreme Programming Explained: Embrace Change*, 2nd Edition, Addison-Wesley (2004).

[4] J.A. Bergstra, J. Heering and P. Klint, *Algebraic Specification*, ACM Press (1989).

[5] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser and J. Visser, "The ASF+SDF Meta-environment: A Component-Based Language Development Environment", Proc. Compiler Construction 2001, 365–370 (2001).

[6] T. Cassidy, J.R. Cordy, T. Dean and J. Dingel, "Source Transformation for Concurrency Analysis", Proc. LDTA 2005, ACM 5th International Workshop on Language Descriptions, Tools and Applications, 26–43 (2005).

[7] J.R. Cordy, "TXL - A Language for Programming Language Tools and Applications", Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications, *Electronic Notes in Theoretical Computer Science* **110**, 3–31 (2004).

[8] J.R. Cordy, I.H. Carmichael and R. Halliday, *The TXL Programming Language*, Queen's University at Kingston (1988, rev. 2005).

[9] J.R. Cordy and E.M. Promislow, "Specification and Automatic Prototype Implementation of Polymorphic Objects in Turing Using the TXL Dialect Processor", Proc. 1990 IEEE International Conference on Computer Languages, 145–154 (1990).

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

[11] J.R. Cordy and M. Shukla, "Practical Metaprogramming", Proc. CASCON'92, IBM Center for Advanced Studies 1992 Conference, 215–224 (1992).

[12] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System", *Journal of Information and Software Technology* **44**(13), 827–837 (2002).

[13] J.R. Cordy, K.A. Schneider, T.R. Dean and A.J. Malton, "HSML: Design Directed Source Code Hot Spots", Proc. IEEE 9th International Workshop on Program Comprehension, 145–154 (2001).

[14] T.R. Dean, J.R. Cordy, K.A. Schneider and A.J. Malton, "Experience Using Design Recovery Techniques to Transform Legacy Systems", Proc. 2001 IEEE International Conference on Software Maintenance, 622-631 (2001).

[15] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, "Agile Parsing in TXL", *Journal of Automated Software Engineering* **10**(4), 311–336 (2003).

[16] G. Gelernter, "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems* **7**(1), 80-112 (1985).

[17] P. Godefroid, "On the Costs and Benefits of using Partial-Order Methods for the Verification of Concurrent Systems", Proc. DIMACS Workshop on Partial-Order Methods in Verification (1996).

[18] X. Guo, J.R. Cordy and T.R.Dean, "Unique Renaming of Java Using Source Transformation", Proc. SCAM 2003, IEEE 3rd International Workshop on Source Code Analysis and Manipulation, 151–160 (2003).

[19] C. Halpern, "TXL: A Rapid Prototyping Tool for Programming Language Design", M.Sc. thesis, Department of Computer Science, University of Toronto (1986).

[20] R.C. Holt and J.R. Cordy, "The Turing Programming Language", *Communications of the ACM* **31**(12), 1410–1423 (1988).

[21] N. Kiyavitskaya, N. Zeni, J.R. Cordy, L. Mich and J. Mylopoulos, "Applying Software Analysis Technology to Lightweight Semantic Markup of Document Text", Proc. ICAPR 2005, 3rd International Conference on Advances in Pattern Recognition, *Lecture Notes in Computter Science* **3686**, 590–600 (2005).

[22] D.A. Lamb and K.A. Schneider. "Formalization of Information Hiding Design Methods", Proc. CASCON '92, IBM Center for Advanced Studies 1992 Conference, 201–214 (1992).

[23] A.J. Malton, "The Denotational Semantics of a Functional Tree Manipulation Language", *Computer Languages* **19**(3), 157–168 (1993).

[24] T.J. Parr and R. W. Quong, "ANTLR: A Predicated LL(k) Parser Generator," *Software, Practice and Experience* **25**(7), 789–810 (1995).

[25] R. Paige, "Viewing a Program Transformation System at Work", Proc. 6th International Conference on Programming Language Implementation and Logic Programming, *Lecture Notes in Computer Science* **844**, 5–24 (1991).

[26] R. Paige, "APTS External Specification Manual", Unpublished manuscript, available at *http://www.cs.nyu.edu/ jessie* (1993).

[27] R. Ramesh and I. V. Ramakrishnan, "Nonlinear Pattern Matching in Trees", *Journal of the ACM* **39**(2): 295–316 (1992).

[28] O.Tal, S. Knight, and T. Dean, "Syntax-based Vulnerability Testing of Frame-based Network Protocols", Proc. 2nd Annual Conference on Privacy, Security and Trust (2004).

[29] M. Tomita, "An Efficient Augmented Context-free Parsing Algorithm", *Computational Linguistics* **13**(1–2), 31–46 (1987).

[30] E. Visser, "Stratego: A Language for Program Transformation based on Rewriting Strategies", Proc. Rewriting Techniques and Applications (RTA'01), *Lecture Notes in Computer Science* **2051**, 357–361 (2001).

[31] E. Visser, "Program Transformation in Stratego/XT: Rules, Strategies, Tools and Systems in Stratego XT/0.9", Proc. Domain Specific Program Generation 2003, *Lecture Notes in Computer Science* **3016**, 216–238 (2004).

[32] R. Zanibbi, D. Blostein and J.R. Cordy, "Recognizing Mathematical Expressions Using Tree Transformation", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(11), 1455–1467 (2002).