Proc. SCAM'01 - IEEE 1st International Workshop on Source Code Analysis and Manipulation, Florence, November 2001, pp. 168-178

Software Engineering by Source Transformation - Experience with TXL

James R. Cordy Thomas R. Dean+ Andrew J. Malton‡ Kevin A. Schneider†

Department of Computing & Information Science, Queen's University
Kingston, Ontario, Canada K7L 3N6
{cordy,dean,malton,kas}@cs.queensu.ca

Abstract

Many tasks in software engineering can be characterized as source to source transformations. Design recovery, software restructuring, forward engineering, language translation, platform migration and code reuse can all be understood as transformations from one source text to another. TXL, the Tree Transformation Language, is a programming language specifically designed to support rule-based source to source transformation. Originally conceived as a tool for exploring programming language dialects, TXL has evolved into a general purpose software transformation system that has proven well suited to a wide range of software maintenance and re-engineering tasks, including the design recovery, analysis and automated reprogramming of billions of lines of commercial Cobol, PL/I and RPG code for the Year 2000. In this short paper we introduce the basic features of modern TXL and its use in a range of software engineering applications, with an emphasis on how each task can be achieved by source transformation.

1. Background

Many tasks in software engineering and maintenance can be characterized as source to source transformations. Reverse engineering or design recovery [1] can be cast as a source transformation from the text of the legacy source code files to the text of a set of design facts. Software re-engineering and restructuring [2] can be cast as a source transformation from the poorly structured original source code text to the better structured new source code. Forward engineering or metaprogramming [3], can be cast as a transformation from the source text of design documents and templates to the instantiated source code files. Platform translation and migration tasks are easily understood as transformations from the original source code files to new source code files in the new language or paradigm. And code reuse tasks can be implemented as a source transformation from existing, tested source code to generic reusable source code modules.

While many other methods can be applied to various parts of these problems, at some point each of them must involve dealing with actual source text of some kind at each end of the process. In this short paper we describe our experiences with attempting to tighten the relationship between the source text artifacts at each end of the processes by experimenting with actually implementing these and other software engineering tasks using pure source text transformations in the TXL source transformation language [4,5]. The experience we report is a summary of the results of many different projects over the past ten years, culminating with the success of the approach in addressing the difficulties associated with the famous "millennium bug" for over three billion lines of source code.

2. Overview of TXL

TXL is a programming language and rapid prototyping system specifically designed to support structural source transformation. Source text structures to be transformed are described using an unrestricted ambiguous context free grammar in extended Backus-Nauer (BNF) form, from which a structure parser is automatically derived. Source transformations are described by example, using a set of context sensitive structural transformation rules from which an application strategy is automatically inferred.

In order to give the flavor of the by-example style of TXL, Figure 1 shows a simple transformation rule for the base step of a transformation to vectorize sequences of independent scalar assignments in a Pascal-like programming language.

2.1. The TXL Processor

The TXL processor is a compiler and run time system for the TXL programming language that directly interprets TXL programs consisting of a grammatical specification of the structure of the input text and a set of structural transformation rules such as the one in Figure 1 to implement a source to source transformation. The result is a rapid prototype of the source transformer described by the rules that can be used immediately on real input (Figure 2).

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).
+ Author's current address: Department of Electrical & Computer Engineering, Queen's Univ., Kingston, Ontario, Canada K7L 3N6.
‡ Author's current address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
† Author's current address: Dept. of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada S7N 5A9.

```
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
```

Figure 1. Simple Example TXL Transformation Rule.

*The **replace** clause gives the pattern for which the rule searches by example in actual source text, binding names to parts (such as the [expression]s) which may vary in each instance. The **by** clause gives the transformed result in similar style, using the bound names to transfer parts from the matched pattern. The **where** clauses specify additional semantic constraints on when the rule can be applied.*

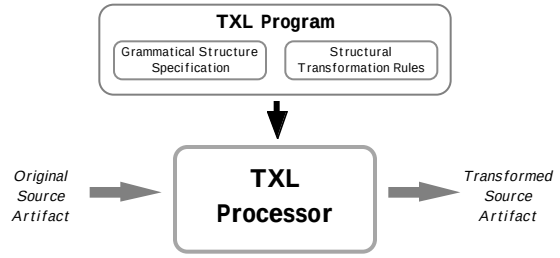


Figure 2. The TXL Processor.

The TXL Processor automatically implements source transformations written in the TXL language.

2.2. Grammatical Notation - Specifying Source Structure

TXL uses a BNF-like grammatical notation to specify source structure (Figure 3). In order to keep the notation lightweight and in a by-example style, terminal symbols of the input, for example operators, semicolons, keywords and the like, appear simply as themselves. References to nonterminal types defined elsewhere in the grammar appear in square brackets [ ]. The usual set of BNF extensions for sequences, written as [**repeat** X] for nonterminal [X], optional items, written as [**opt** X], and lists, written as [**list** X], are available.

```
define program                  % goal symbol
    [expression]                % of the grammar
end define

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

define term
    [primary]
  | [term] * [primary]
  | [term] / [primary]
end define

define primary
    [number]
  | ( [expression] )
end define
```

Figure 3. Simple Example of a TXL Grammar.

*Terminal symbols such as +, -, *, / and the parentheses in the definitions above represent themselves. References to nonterminal types are denoted by square brackets, as in* [expression] *above. TXL comments begin with % and continue to the end of the line.*

Lexical specification is by regular expression patterns in special **tokens** and **compounds** sections, and keywords can be distinguished using a **keys** statement. A large set of predefined nonterminal types for common tokens, including identifiers, string literals, numeric literals, etc. are built in to TXL.

2.3. Rule Notation - Specifying a Transformation

TXL transformation rules are specified using a by-example pattern notation that binds matched items by name in the pattern and copies them by name in the replacement (Figure 4). Pattern variables are explicitly typed using the square bracket

```
rule resolveAddition
   replace [expression]             % target
                                    % type
      N1 [number] + N2 [number]     % pattern to
                                    % search for
   by
      N1 [+ N2]                     % replacement
                                    % to make
end rule
```

Figure 4. Simple Example of a TXL Transformation Rule.

*The **replace** clause specifies the target nonterminal type to be transformed and the by-example pattern to be matched. The **by** clause specifies the by-example replacement to be made for the matched pattern. Because rules are constrained to preserve structure, both the pattern and the replacement must be parseable as the target type. The square bracket notation is used in the pattern to specify type and in the replacement to specify subrule invocation.*

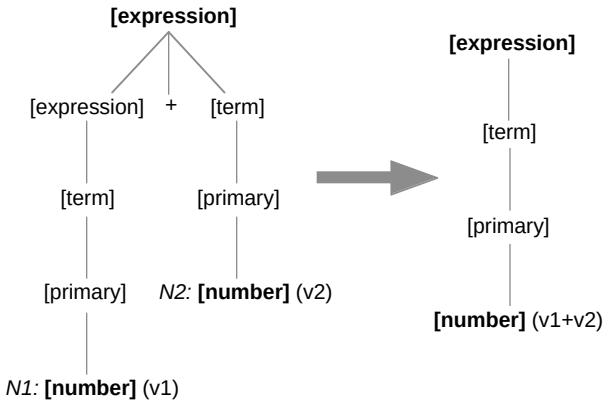


Figure 5. Semantics of the Simple Example Rule in Figure 4.

The example source text given for the pattern and replacement of a rule are parsed into structure tree patterns which are efficiently matched to subtrees of the parse tree of the input to the transformation. Because rules are constrained to preserve nonterminal type, the result is always well formed.

```
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 6. Using Rule Parameters to Reorganize Source.

This example demonstrates the use of rule parameters to implement rules that depend on items bound in previous patterns. In this case the second rule implements inlining of constant values by searching for [primary] *references that match the identifier of a named constant, and replaces each such reference with the constant's value. The first rule insures that this is done for every declared named constant.*

nonterminal type notation. For example X[T] binds a pattern variable named X of nonterminal type [T].

Pattern variables may be used in the replacement to copy the item bound in the pattern into the result of the rule. Variables copied into the replacement may optionally be further transformed by *subrules* using the notation X[R], where R is the name of a transformation rule. Subrule invocation has the semantics of function application - in traditional functional notation, the TXL subrule invocation X[R] would be written *R(X)*, the composed subrule invocation X[R1][R2] would be written *R2(R1(X))*, and so on.

Transformation rules are constrained to preserve nonterminal type in order to guarantee a well-formed result. Each transformation rule searches its *scope* (the item it is applied to) for instances of its pattern and replaces each one with an instance of its replacement, substituting pattern variables in the result. Patterns and replacements are source text examples from which the intended structure trees are inferred by the parser (Figure 5). Rules automatically re-apply to their own result until no further matches are found.

Subrules can be parameterized to use items bound from previous pattern matches in their patterns and replacements (Figure 6). In this way complex transformations involving large scale reorganization of source structures can be specified.

Pattern matches can be refined using **deconstruct** clauses, which constrain bound pattern variables to match more detailed patterns (Figure 7), and using **where** clauses, which allow for semantic constraints on items bound to pattern variables (Fig. 8).

```
rule createSimultaneousAssignments
   replace [repeat statement]
      IfStatement [if_statement] ;
      RestOfStatements [repeat statement]
   deconstruct * [if_condition] IfStatement
      IfCond [if_condition]
   deconstruct IfCond
      false
   by
      RestOfStatements
end rule
```

Figure 7. Constraining Variables to More Specific Patterns.

*The **deconstruct** clause allows for stepwise pattern refinement. The first **deconstruct** in the example above constrains the item bound to* IfStatement *to contain an* [if_condition], *which is bound to the new pattern variable* IfCond. *The second deconstruct constrains* IfCond *to be exactly the identifier* false. *If either **deconstruct** fails, the entire pattern match fails and the scope is searched for another match.*

```
rule sortNumbers
   replace [repeat number]
      N1 [number] N2 [number] Rest [repeat number]
   where
      N1 [> N2]
   by
      N2 N1 Rest
end rule
```

Figure 8. Semantic Constraints on Bound Variables.

*The **where** clause constrains items bound to pattern variables using a (possibly complex) set of subrules. In the rule above, the number bound to* N1 *is constrained to be greater than the number bound to* N2, *otherwise the pattern match fails. (This rule is the entire specification for bubble sorting a sequence of numbers in TXL.)*

3. Software Engineering by Source Transformation

The remainder of this paper gives examples of how TXL has been used in various software engineering projects in research and industry over the past ten years.

3.1. Interface Translation

ESPRIT Project REX [6] was an ambitious and wide ranging project to explore the specification and implementation of reusable, extensible distributed systems, and was the first serious use of TXL in software engineering tasks. One of the key ideas in REX was the use of language independent interface specifications in the ISL notation [7]. Closely related to IDL [8], ISL allowed the specification of complex data structures to be passed between nodes in a widely distributed heterogeneous network. Each node in the network could be implemented using different hardware, operating system and programming language.

In order to insure that data could be reliably passed between nodes implemented using different programming languages, it was important that the interfaces described in ISL be accurately and consistently represented in each language. Initially such representations were carried out by hand translation, but very quickly it became obvious that it was much too difficult and error prone to incrementally adapt such translations in response to changes in the ISL specifications of the interfaces.

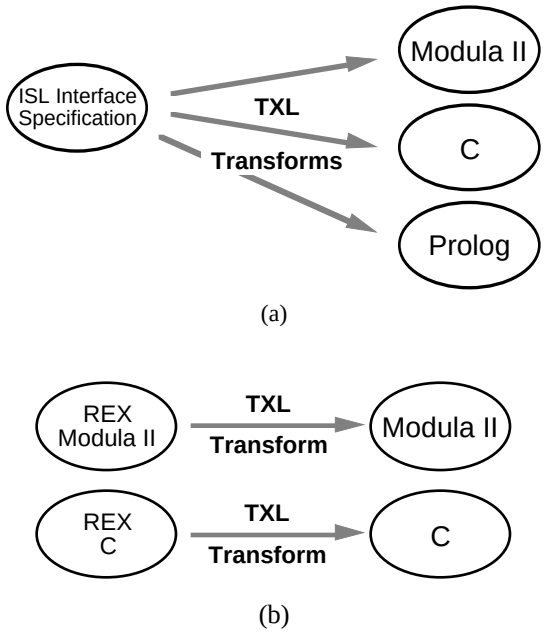


Figure 9. Applications of TXL in ESPRIT Project REX.

TXL transformation rule sets were used to instantiate ISL data structure descriptions into data type declarations in each of the REX target languages (a), and to implement REX extended dialects of each target language (b).

```
% Project REX - TXL ruleset for transforming from
% REX Extended Modula II -> unextended Modula II
% Georg Etzkorn, GMD Karlsruhe, 25.02.91
% Part 4 - Transform SELECT statements

rule transformSelectStatement ModuleId [id] PortListId [id]
   replace [repeat statement]
      SELECT
         Alternatives  [repeat or_alternative]
         OptElse       [opt else_StatementSequence]
      END;
      RestOfStatements [repeat statement]

   construct InsertPortStatements [repeat statement]
      _ [mapAlternativeToIf PortListId ModuleId each Alternatives]

   construct PortMessageCases [repeat or_case]
      _ [mapAlternativeToCase ModuleId each Alternatives]

   by
      AllocPortList (PortListId);
      InsertPortStatements
      CASE WaitOnPortList (PortListId) OF
         PortMessageCases
      END;
      ReleasePortList (PortListId);
      RestOfStatements
end rule
```

Figure 10. Transformation Rule to Implement the REX Modula II SELECT Statement.

Transformation from REX Modula II to pure Modula II is not just a matter of syntax, as this rule demonstrates. Each REX Modula II SELECT statement is transformed to a complex set of logic involving a sequence of IF statements derived from each alternative followed by a CASE statement whose cases are derived in a different way from the same set of alternatives. The by-example style of TXL makes the overall shape of both the original and the translated result easy to see. Figure 11 shows an example of this transformation.

Instead, a source transformation from ISL to each target language was designed and implemented in TXL (Figure 9(a)). Once completed, these transformations allowed much more rapid experimentation since only the ISL specification need be changed when updating interface data structures.

3.2. Language Extension

Project REX also involved research in appropriate language features for supporting effective distributed computing. New language primitives were designed to support REX message passing for each target language. REX extended languages included REX Modula II, REX C, REX Prolog, and so on. In each case, the semantics of the new features were specified using templates in the original unextended language, augmented with calls to a standard REX communication library that was common across all languages.

TXL was used to provide usable implementations of each of the REX extended languages by directly implementing their semantics as source transformations to the original languages (Figure 9 (b)). The relationship between the source of the REX extended language and the original language was often both semantically and structurally complex, and is not simply a question of syntax. Figure 10 shows one of the transformation rules in the TXL implementation of REX Modula II, and Figure 11 shows an example of the transformation it implements.

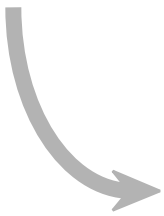
3.3. Design Recovery

The Software Design Technology (SDT) project was a joint project involving IBM Canada and Queen's University. The global goal of the project was to study how we can formalize and better maintain the relationship between design documents and actual implementation code throughout the life cycle of a software system. Early on it was realized that if such an approach is to be introduced into practice, it must be applicable to 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 [9].

While analysis of source code for the purpose of design recovery does not at first glance seem like a good application for source transformation, it quickly became clear that the pattern matching capabilities of TXL made it well suited to this task. Using the three stage source annotation approach shown in Figure 12, a completely automated design recovery system was implemented in TXL and used to recover the design of a large piece of software for the purpose of design analysis.

The approach involves several TXL transformations, each of which searches for a set of source patterns for a particular design relationship and annotates the source with design facts for the relationship in Prolog notation (Figure 13). These "embedded" design facts are then extracted and merged to create a Prolog design database for the entire program.

```
SELECT
  (allocated = None) OR (allocated = Left) && ACCEPT(left,req):
      CASE req.order OF
          pickup: allocated := Left;  | putdown: allocated := None;
      END;
      REPLY(left,ack);
  | (allocated = None) OR (allocated = Right) && ACCEPT(right,req):
      CASE req.order OF
          pickup: allocated := Right; | putdown: allocated := None;
      END;
      REPLY(left,ack);
END;
```



```
AllocPortList (XdrM2_PortList1);
IF (allocated = None) OR (allocated = Left) THEN
    InsertPort (XdrM2_PortList1, XdrM2_Fork.left, NoHandle);
END;
IF (allocated = None) OR (allocated = Right) THEN
    InsertPort (XdrM2_PortList1, XdrM2_Fork.right, NoHandle);
END;
CASE WaitOnPortList (XdrM2_PortList1) OF
    XdrM2_Fork.right:
      XdrM2_Fork.Accept_right (XdrM2_Fork.right, NoHandle, req);
      CASE req.order OF
          pickup: allocated := Right; | putdown: allocated := None;
      END;
      XdrM2_Fork.Reply_left (XdrM2_Fork.left, NoHandle, ack);
  | XdrM2_Fork.left:
      XdrM2_Fork.Accept_left (XdrM2_Fork.left, NoHandle, req);
      CASE req.order OF
          pickup: allocated := Left;  | putdown: allocated := None;
      END;
      XdrM2_Fork.Reply_left (XdrM2_Fork.left, NoHandle, ack);
END;
ReleasePortList (XdrM2_PortList1);
```

Figure 11. Trivial Example of the Transformation Implemented by the TXL Rule in Figure 10.

This example demonstrates the complexity of the relationship between the original source and the result source in this transformation. For larger SELECT statements, the difference is even more striking.

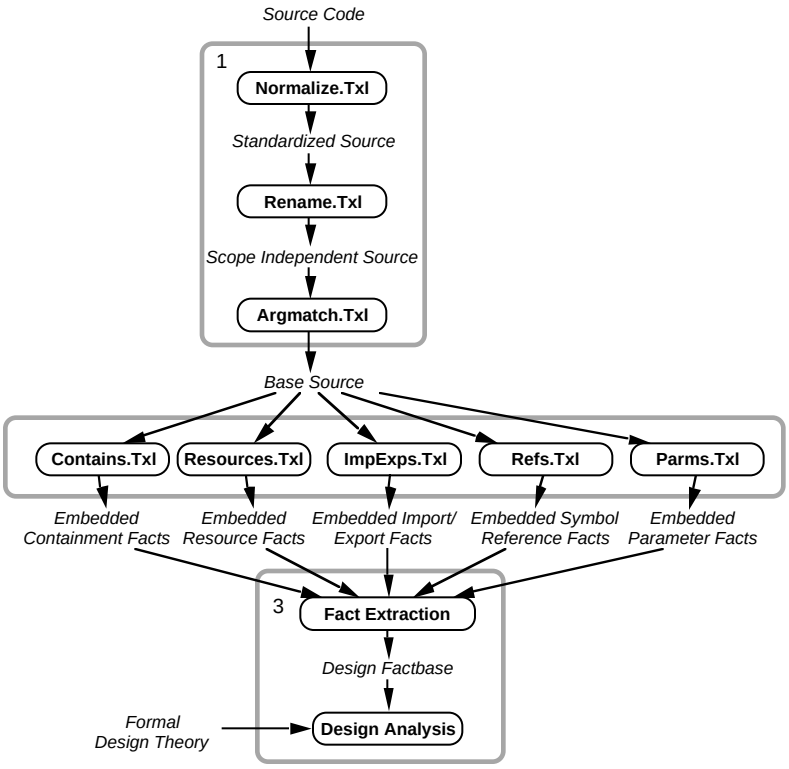


Figure 12. Design Recovery by Source Transformation.

The process involves three stages: a normalization phase which uses TXL transforms to resolve global unique naming of identifiers (1), a fact generation phase which uses a set of TXL transforms to recognize and annotate source with design relationships (2), and an extraction phase which gathers the annotations into a design database (3). This same basic strategy was used in the LS/2000 system to recover the designs of over three billion lines of source applications written in Cobol, PL/I and RPG.

```
rule processProcedureRefs
   replace $ [declaration]
      procedure P [id] ParmList [opt parameter_list]
         Scope [repeat statement]
      'end P
   by
      procedure P ParmList
         Scope [embedProcCalls P]
               [embedFuncCalls P]
               [embedVarParmRefs P]
               [embedPutRefs P]
               [embedGetRefs P]
      'end P
end rule

rule embedVarParmRefs ContextId [id]
   replace $ [argument]
      ReferencedId [id] Selectors [repeat selector] : var FormalId [id]
   by
      ReferencedId Selectors : var FormalId [id]
      $ vararguse (ContextId, ReferencedId, FormalId) $
end rule
```

Figure 13. A Simple Design Recovery Rule.

This simple rule demonstrates the strategy of source annotation to represent design relationships. For each procedure declaration in the source, the first rule invokes a number of different subrules to recognize design relationships in the procedure's inner scope. One such relationship is recognized by the second rule, which annotates each argument of each procedure call in the scope with a "vararguse" design fact. Most design relationship rules are more complex than this one.

```
\   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 14. A μ* template for generating a C entry point array.

The code between the lines beginning with a backslash \ is a template for the C code to be generated. The code following the second backslash is μ notation for a complex query on the design database that guides the generation.*

While this first TXL-based design recovery system was only a small scale research prototype, this same approach has been used in large scale industrial applications such as the LS/2000 Year 2000 analysis and conversion system [10], which used TXL-based design recovery to process more than three billion lines of Cobol, PL/I and RPG source code.

3.4. Metaprogramming

The other half of maintaining the relationship between design documents and actual implementation concerns the generation of original code from design documents, often called automatic programming or metaprogramming [3]. In metaprogramming, the generation of code 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.

μ* (pronounced "mew star") is a family of metaprogramming languages sharing a common notation and implementation. In μ*, templates are written as example source in the target programming language, which may be one of many, including C, Prolog, etc., 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 14).

Implementation of μ* uses a two-stage source transformation system implemented in TXL. 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 (Figure 15). This 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.

3.5. Software Restructuring

Code restructuring [2] is perhaps the most natural application of source program transformation. The Modularity Toolkit [11] is a sequence of TXL transformations designed for remodularizing legacy source code. Even in a well designed system, as maintenance proceeds over the years, original module boundaries are blurred as requirements shift and the system is tuned for performance. This blurring of the original modular design is a major contributor to the degradation in maintainability of mature systems.

In extreme cases, blurring becomes so extreme that existing module boundaries in the code actually get in the way of further maintenance. The modularity toolkit addresses this situation by attempting to remodularize to better reflect the design relationships actually present in the maintained code. Using

---

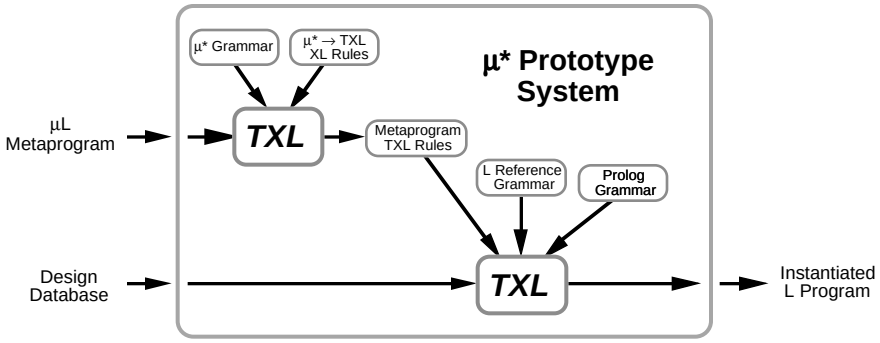


Figure 15. The μ* prototype implementation.

Templates for the language L are expressed as mL metaprograms. A generic TXL transformation translates μ metaprograms (for any target language) to a TXL ruleset. The TXL ruleset is then combined with the TXL grammar for target language L to implement a second TXL transformation that transforms the design database into instantiated language L program code.*

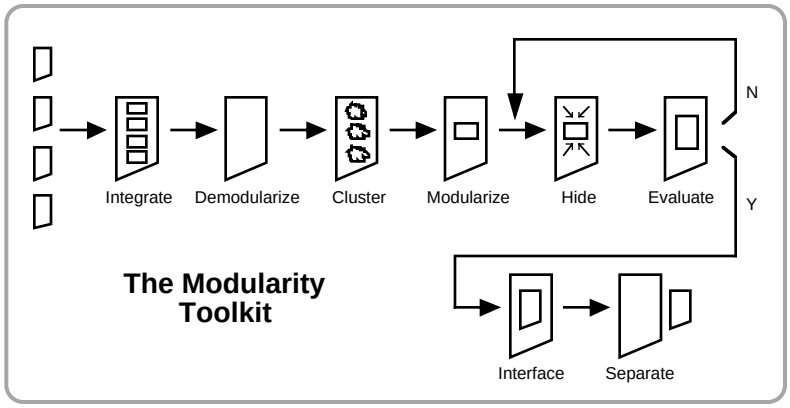


Figure 16. The Modularity Toolkit.

Remodularization consists of integrating the source code of all existing modules and removing module boundaries, textually clustering together related source items, introducing new module boundaries for the clusters and evaluating the result. When a new module is accepted by the user, additional source transformations introduce the new interfaces and separate the result into new modules.

clustering algorithms to gather together the most tightly related data and code, the system propose new modularizations that can be considered by an expert programmer and accepted or modified interactively to yield a better modularized result (Figure 16).

Each stage of this process is implemented by an independent TXL source transformation under control of a unified user interface that allows the transformations to be applied sequentially or in any chosen order that makes sense to the expert. Each stage is itself a complex multi-stage TXL source transformation (Figure 17). This system was used to improve the modularity of several heavily maintained Turing language [12] programs, including the implementation of TXL itself.

3.6. Maintenance Hot Spots

Maintenance hot spots [13] are a generalization of 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, expansion of credit card numbers from 13 to 16 digits, or changes to European exchange computation laws, 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.

```
function createModule ModuleId [id] VarsToHide [repeat id]
   replace [program]
      Comments [opt comment_lines]
      Imports  [repeat import_list]
      Body     [repeat statement]
   by
      Comments
      Imports
      Body  [createEmptyModule         ModuleId]
            [hideVarsInModule          ModuleId VarsToHide]
            [createAccessRoutines      ModuleId each VarsToHide]
            [moveRoutinesIntoModule    ModuleId VarsToHide]
            [qualifyExportedReferences ModuleId VarsToHide]
            [createImportExports       ModuleId VarsToHide]
            [relocateModuleInProgram   ModuleId VarsToHide]
end function
```

Figure 17. Main TXL function of the Modularize stage of the Modularity Toolkit.

Each stage is implemented by an independent TXL source transformation which itself may involve multiple transforms. In the Modularize stage, new module boundaries are introduced using source transformations to create a new empty module, to hide the clustered variables in it, to create access routines for the variables, to move routines in the cluster inside the module, to qualify all external references to the new module's variables and routines, to create the module's import/export interface and to relocate the new module appropriately in the program source.

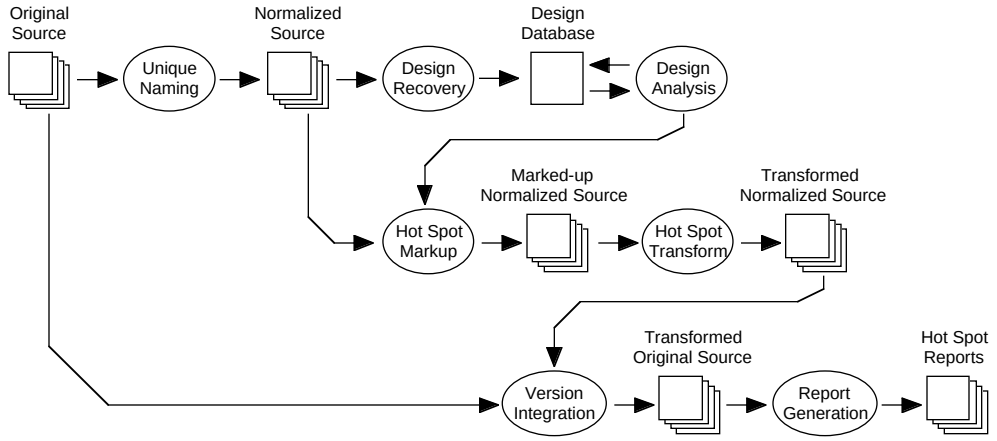


Figure 18. The LS/2000 Process Architecture.

Software system source files are first uniquely renamed and normalized to factor out irrelevant formatting and syntactic details. The normalized source files are then analyzed using design recovery to produce a detailed design database. Under guidance of a human analyst, design analysis identifies those design entities which are of interest in the context of the particular maintenance task. Hot spot markup then uses the results of the design analysis to identify sections of code that may need to be changed ("hot spots"), which hot spot transform automatically reprograms using a set of transformation patterns. Finally, version integration restores original source formatting, naming and syntactic details to the reprogrammed source and report generation summarizes the changes. All phases of LS/2000, with the exception of version integration and report generation, were implemented using TXL source transformations.

LS/2000 [10] used the concept of maintenance hot spots to assist in the Year 2000 conversion of over three billion lines of Cobol, PL/I and RPG source code. Using a variant of the design recovery process described in Section 3.3 followed by design analysis and hot spotting processes 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 set of transformation patterns. 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 a few thousand lines of source was reduced from a few hours to less than five minutes, and accuracy of conversion before testing was increased from about 75% to over 99%.

At the core of LS/2000 were the hot spot markup and hot spot transform phases of the process (Figure 18). 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. Potentially Year 2000 sensitive operations in the source were marked as hot by a set of TXL rules using source patterns guarded by matches of the source facts. Figure 19 shows one such markup rule.

Hot spot transform then took as input the 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

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

Figure 19. Example LS/2000 Hot Spot Markup Rule.

Each hot spot markup rule searches for a particular pattern, in this case a condition using the operator "<", which involves something classified by design analysis as interesting, in this case operands known to represent dates of type "YYMMDD". Interesting instances of the pattern are marked up using hot spot brackets, in this case"{DATE-INEQUALITY-YYMMDD" and "}DATE-INEQUALITY-YYMMDD". "DateFacts" above refers to the Prolog source facts for the result of design analysis. The deconstruct *statements in the rule use source pattern matching to query the source facts for "Date" facts about the operands. This rule has been simplified for presentation in this paper. In practice hot spot markup rules are much more general than this one.*

```
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 20. Example LS/2000 Hot Spot Transform Rule.

Each hot spot transform rule implements the reprogramming template for a particular kind of hot spot. In this case the rule searches for IF statements whose condition has been marked as DATE-INEQUALITY-YYMMDD. When one is found, it is replaced by a copy of the statement in which the inequality operands have been replaced by new operands whose value is computed by ADD statements inserted to implement the Year 2000 windowing computation. A separate transformation rule later inserts declarations for the new operands. This rule has been simplified for presentation in this paper. In practice, most hot spot transform rules are much more general than this one, covering many cases in one rule.

---

reprogramming templates based on a "windowing" solution to Year 2000. Because the hot spot markup phase had explicated the kind of potential risk in the markup label of each hot spot, these templates could be applied very efficiently. Figure 20 shows an example of a TXL hot spot transform rule for reprogramming one kind of Year 2000 hot spot.

The ideas behind LS/2000 have been generalized and codified in the LS/AMT automated maintenance system discussed in [13]. By customizing the hot spot markup and transform phases to a range of other large scale software maintenance problems, LS/AMT has already been used to process more and one and a half billion lines of additional code in addressing problems such as IT mergers, database migrations, web migrations and other large scale maintenance tasks.

4. Summary

TXL is a general and flexible source transformation language and rapid prototyping system which has been used in a wide range of software engineering and maintenance applications. This short paper has given a quick introduction to the flavor of TXL and our experience in its application to a range of software maintenance activities. We observed that most automatable software engineering applications can be modeled as source to source transformations, and showed how TXL can be used to implement some of these models in practice.

TXL has been used in hundreds of projects in industry and academia all over the world, including several other software engineering applications such as implementation of the HSML design-directed source code mining system [13] and the reverse engineering facilities of the commercial Graphical Designer CASE tool [14]. With the move towards XML [15] based software representations such as GXL [16], we expect that source to source transformation will be playing an even greater role in software engineering in the future.

5. Related Work

The important role of transformation in software engineering has been pointed out by several other researchers. Perhaps the most eloquent spokespersons for transformation in software engineering are Ira Baxter and Christopher Pidgeon of Semantic Designs, who have proposed a design-directed transformational model for the entire software life cycle [19]. Other source transformation tools have been used to implement software engineering tasks of various kinds. Of particular note are Gentle [20], FermaT [21] and NewYacc [22], any of which could be used to implement most of the techniques described in this paper. TXL is distinguished from these systems in its use of by-example patterns and replacements, which simplify the specification of practical transformation tasks by shielding the user from direct manipulation of internal abstract structures.

6. Acknowledgments

TXL has benefited from the contributions of a range of people over many years. The original Turing programming language extension tool from which TXL has evolved was designed by Charles Halpern and James R. Cordy at the University of Toronto in 1985, and the first practical implementations were developed by Ian Carmichael and Eric Promislow at Queen's University between 1986 and 1990. The design and implementation of the modern TXL language and transformation system was undertaken by James R. Cordy at GMD Karlsruhe and Queen's University between 1990 and 1995. Andrew Malton developed the formal semantics of the modern TXL language at Queen's University in 1993 [17].

Independent early explorations of the application of TXL to software engineering tasks were undertaken by Georg Etzkorn, Nicholas Graham, Kevin Schneider and Donald Jardine of Queen's University and GMD Karlsruhe. The TXL approach to software engineering tasks was heavily inspired by the theory-model paradigm for software design described by Arthur Ryman of the Centre for Advanced Studies of IBM Canada [18].

A range of experiments with TXL have been carried out by graduate students of Queen's University including Medha Shukla Sarkar, Ramesh Srinivasan, Rateb Abu-Hamdeh, Chunsheng Xie, Edna Abraham, Russell Halliday, Darren Cousineau, Andy Maloney, Minchul Cha and Richard Zanibbi. Finally, several hundred TXL users from institutions all over the world have contributed to the evolution of TXL from a compiler technology academic toy to an industrial strength software transformation system over the past ten years.

Development of TXL has been funded at various stages by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the Information Technology Research Centre (ITRC, a province of Ontario Centre of Excellence), by ESPRIT project REX and GMD (the German National Research Centre for Information Technology) Karlsruhe, and by the University of Toronto and Queen's University.

References.

[1] T.J. Biggerstaff, "Design recovery for maintenance and reuse", *IEEE Computer* 22,7 (July 1989), pp. 36-49.

[2] R.S. Arnold, "Software Restructuring", *Proceedings of the IEEE* 77,4 (April 1989), pp. 607-617.

[3] J.R. Cordy and M. Shukla, "Practical Metaprogramming", *Proc. CASCON '92, IBM Centre for Advanced Studies 1992 Conference*, Toronto, Canada (November 1992), pp. 215-224.

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

[5] J.R. Cordy, I.H. Carmichael and R. Halliday, *The TXL Programming Language - Version 10*, Legasys Corp. and Queen's University, Kingston, Canada, January 2000.

[6] J. Magee, J. Kramer, M. Sloman and N. Dulay, "An Overview of the REX Software Architecture", *Proc. 2nd IEEE CS Workshop on Future Trends of Distributed Computing Systems*, Cairo, Egypt (October 1990), pp. 396-402.

[7] F. Bieler (ed.), "The REX Interface Specification Language", Technical Report REX-WP2-GMD-36.1.1, GMD Karlsruhe, Karlsruhe, Germany (June 1990).

[8] D.A. Lamb, "IDL: Sharing Intermediate Representations", *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), pp. 297-318.

[9] D.A. Lamb and K.A. Schneider, "Formalization of information hiding design methods", *Proc. CASCON '92, IBM Centre for Advanced Studies Conference*, Toronto, Canada (November 1992), pp. 201-214.

[10] J.R. Cordy, "The LS/2000 Technical Guide to the Year 2000", Technical Report ED5-97, Legasys Corp., Kingston, and IBM Corp., Toronto (April 1997).

[11] R. Srinivasan, "Automatic Software Design Recovery and Re-Modularization Using Source Transformation", M.Sc. thesis, Department of Computing and Information Science, Queen's University, Kingston, Canada (April 1993).

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

[13] J.R. Cordy, K.A. Schneider, T.R. Dean and A.J. Malton, "HSML: Design Directed Source Code Hot Spots", *Proc. IWPC 2001 - 9th Int. Workshop on Program Comprehension*, Toronto, Canada (May 2001).

[14] Advanced Software Technologies Inc., *GDPro 5.0* (Sept. 2000).

[15] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0", W3C Recommendation REC-xml19980210 (February 1998).

[16] R.C. Holt, A. Winter and A. Schürr, "GXL: Toward A Standard Exchange Format", *Proc. WCRE 2000 Working Conference on Reverse Engineering*, Brisbane, Australia (November 2000).

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

[18] A.G. Ryman, "Constructing Software Design Theories and Models", in *Studies of Software Design*, Springer Verlag *Lecture Notes in Computer Science* 1078 (1996), pp.103-114.

[19] I. Baxter and C. Pidgeon, "Software Change Through Design Maintenance", *Proc. ICSM'97, IEEE 1997 International Conference on Software Maintenance*, Bari, Italy (October 1997), pp. 250-259.

[20] F.W. Schröer, *The GENTLE Compiler Construction System*, R. Oldenbourg Verlag, Munich and Vienna, 1997.

[21] M.P. Ward, "Assembler to C Migration using the FermaT Transformation System", *Proc. ICSM'99, IEEE 1999 International Conference on Software Maintenance*, Oxford (Sept. 1999), pp. 67-76.

[22] James J. Purtilo and John R. Callahan, "Parse-Tree Annotations", *Communications of the ACM* 32,12 (December 1989), pp. 1467-1477.