Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension, Toronto, May 2001, pp. 127-134

Processing Software Source Text in Automated Design Recovery and Transformation

Andrew Malton
James R. Cordy
Darren Cousineau
Kevin A. Schneider
Thomas. R. Dean
Jason Reynolds
Legasys Corporation, Kingston, Ontario, Canada
{malton,kas,cordy,dean}@cs.queensu.ca

Abstract

Software source text is the raw material of program understanding and transformation systems. In order to share the results of source analyses, both between phases of a design recovery process, and between tools and systems in different processes, a source text interchange format is needed. This paper describes a simple technique, 'source factoring', by which a common structural decomposition of source text can address the many issues of preprocessing, macro processing, lexical analysis, design recovery, and automated transformation. Above all, source factorization allows the results of design analysis to be attached to source, and the results of source transformation to be reinstalled cleanly into the code base. This view of source text underlies the architecture of a successful software maintenance system which has processed billions of lines of legacy code in all major programming languages.

1. Design recovery and transformation

During the last few years increasing attention has been paid to the question of suitable architectures for development of software reengineering systems. As observed by Ebert *et. al* [16], with the increasing complexity of such systems, and the need to exchange results between unrelated software reengineering environments, there arises the need for common schemata, and consequently common interchange formats.

A number of such schemata and interchange formats have been described and developed in the literature. By and large they model recovered data by means of various kinds of graph, generally representing entity-relationship databases. So, for example, in the Rigi system [12], the Rigi Standard Format [13] enables the sharing of semantic graphs derived from source, very suitable to visualization of architectural or other design views. Most recently the GXL project [14] is unifying several model for interchange or source facts at every level: again the basic paradigm is the representation of source code data as graphical or E-R data bases.

At Legasys we developed a software architecture [6,8] for design recovery and transformation of large legacy code bases. During this development we discovered the value of a hybrid approach (as well justified in [15]) in which recovered source data are partly represented as graphs or database facts, and partly

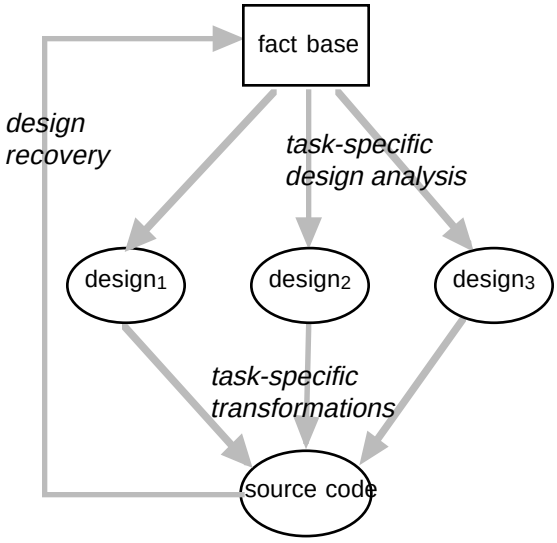

Figure 1

as marked-up source code. This enabled us to distinguish task-specific analysis and transformation processes (see Figure 1) from general source processing activities within the architecture.

For us, *large* means *having up to around ten million source lines*. The processing which we have carried out on these code bases has included the following tasks:

- Source analysis: identifying source files by language, checking completeness and duplication, both of copy-text (include files) and external symbols, handling lexical problems.
- Design analysis: discovering abstract data types, uncovering common data structures and common file structures, duplicated

source, and common processing paradigms.

• Formal transformation: dialect migration, millennium bug renovation, field-size adjustment, technology migration, and language migration (source-to-source transformation).

• Code-based reporting: technology usage reports, dead code identification, backward and forward program slicing, abstract interface identification, "business rule" identification, and the like.

Figure 1 shows at a high level how the terms "design recovery", "base facts", "design analysis", and "transformation" fit together in our architecture. In Figure 1, the *source code* is understood as the real code base, "as given" from the maintainer or code owner. Code bases are imported into our environment by the most straightforward file-and-directory transfer protocol from a code owner's site. In other words, the code is copied warts and all, and an important aspect of dealing with such a code base is robust response to the warts.

In Figure 1, the "design recovery" process is understood as extracting "base recovered design" from the code [3]. We represent the recovered base design as text files containing ground *facts*, as if intended for input to Prolog (although we do not use Prolog or logic programming techniques at present). The data model, which these base facts instantiate, is common to all design recoveries, regardless of source language and intended analysis and transformation tasks.

Figure 1 shows that analysis and transformation tasks are understood as specialized to the task or problem at hand: this may invoke the automated inference of further design information, and also the some hand-tuning by someone playing the role of an "analyst". We have not developed many general analysis or transformation tasks, because the advantage in doing so (i.e. reuse of tools) is gained instead by reusing of the base recovered design, and also by the unified view of source text which is the subject of this report. In summary, task-specific analysis and transformation tasks are specified and developed with the assumption of

• a standard body of available information down to the level of code facts, and

• a standard view of the source code

2. Processing software source text

Source code must be recognized as an extremely rich and varied medium in its own right. It is extremely misleading view it solely as expressions generated by a grammar, or as graphs, or as text files; and it is similarly misleading to reduce the semantics of a live program to the semantic function determined by either a compiler or a formal semantic theory. Source code contains all these things, but more.

When we consider the source of a large system or application suite, we recognize a collection of **independent and interrelated texts**.

Source code is **text**. Source has two purposes, both essential: to represent an artifact which can be realized mechanically (*i.e.* by compilation), and to record communication between human beings. It is text by virtue of its recording and communication role.

Source texts are **independent**. Source texts are maintained as themselves, rather than being dependent on some prior data. They are *input* to the mechanical realization of the software.

(In practice, we frequently encounter 'generated code'. We prefer to obtain and process the (independent) input to the generator instead, but it is not always possible..The usual situation is code which was built by a generator and then has been subject to ongoing maintenance as is.)

Source texts are **interrelated** because of all the links between them that arise as a result of abstraction, including lexical links (*e.g.* "include" directives) and semantic links (subroutine calls, global data, class instantiations, *etc.*, *etc.*)

It is also essential to recognize at least three domains of discourse in source code, and for each of them, several possible subjects. The subjects (relative to the application as a whole) include at least

• design

• implementation

• history

• and the domains of discourse must include at least

• the source domain, regarding the physical organization of the source code itself

• the architecture domain, regarding the logical structure of the application(s), and

• the functionality domain, regarding the mechanical details of functionality.

With these observations, source code is seen to be 'about' many different matters simultaneously, and in many ways to be closer to being natural language than formal language.

Large code bases present peculiar difficulties. Issues which might be ignored or fixed by hand when processing ten or a hundred thousand lines can be overwhelming when the scale is two orders of magnitude greater. Our techniques were developed for application to such large legacy code bases, containing software in many languages at once, including especially COBOL, PL/I, and RPG, and in addition embedded uses of database and transaction processing software. Some of the difficulties in practice have included

• Undocumented or obsolete compiler features

• Varying comment and coding conventions

• Bizarre (but documented!) syntax rules

• Varying lexical and syntax conventions within a single source file.

• Use of macro preprocessor features

• Unparsable text due to the presence of syntax errors

• The requirement to produce results in the presence of errors

• The requirement to produce natural-looking transformed results

• The requirement to deliver results containing excerpts from

the code base.

We have found that by adopting a unified view of source text, we are able to address these difficulties more easily. A

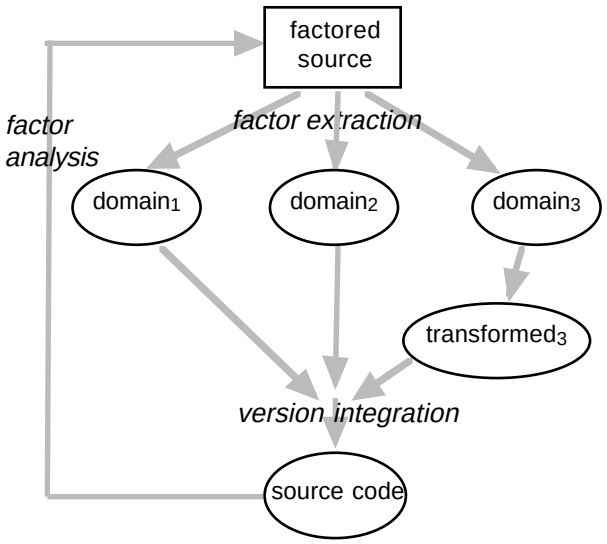

Figure 2

characteristic of all the above problems is that they arise when a formal process (which may be as simple as searching for a string, or as complex as language transformation) applies to a "view" of the text, but its results must somehow be integrated with the text as a whole. This integration problem is characteristic of transformation systems (how do we incorporate modified code back into the original code base?) and informs the whole development.

3. Source factors

Since software source text is similar in richness and complexity to other kinds of source text [2], our approach is a generalization and application of source text markup techniques in the domain of software source code. We use the term 'factored source' to refer to the view we take of the source texts and the manner in which we process them. Figure 2 illustrates how source factors relate to the design recovery and transformation processes: the similarity to Figure 1 is not a coincidence.

In Figure 2, the domains of discourse of a software source text are identified by lexical analysis.

3.1 Kinds of source factors

A **source factor** is a distinguished subsequence of a text, where by 'subsequence' we mean a subset of the character positions in the text, in order. Factors are typically characterized by what is suppressed to produce them, and so they are "views" of the source, suitable for processing in one way or another. A **factored source file** is a file of software source text typically compounded of at least the following factors:

- The 'code factor' contains the text of the program itself, minus comments, continuation markers, directives, *etc.*
- The 'directive factor' contains compiler directives.
- The 'copy prefactor' contains copy directives.
- The 'copy postfactor' contains text copied from include files.

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
SAMPLE.COB
…
000050 DATA DIVISION.
000060 WORKING-STORAGE SECTION
       EJECT
000070 01  REC1.
000080     05 FLD1      PIC ZZZZ9.                                    99/01/03
000090 01  MY-REC2 COPY REC2.
000095* THIS FIELD ADDED 00/01/02.
000100 77  FLD-B        PIC Z9.                                       JAN2AJM
…
REC2.CPY
      * REC2 RECORD STRUCTURE
      01  REC.
          10 FLD1      PIC ZZ9.
…
```

Figure 3 – Raw Source

```
{code…
{seqn000050}seqn DATA DIVISION.
{seqn000060}seqn WORKING-STORAGE SECTION{mdot\.}mdot
{seqn      }seqn {drct EJECT}drct
{seqn000070}seqn 01  REC1.
{seqn000080}seqn     05 FLD1      PIC ZZZZ9.                    {init99/01/03}init
{seqn000090}seqn {copy01  MY-REC2 COPY REC2.\
{seqn      }seqn{cmnt* REC2 RECORD STRUCTURE}cmnt
{seqn      }seqn 01  {replREC\MY-REC2}repl.
{seqn     }seqn     10 FLD1      PIC ZZ9.
…
}copy
{seqn000095}seqn{cmnt* THIS FIELD ADDED 00/01/02.}cmnt
{seqn000100}seqn 77  FLD-B        PIC Z9.                       {initJAN2AJM }init
…
}code
```

Figure 4 – Factored Source

• The ‘comment factor’ contains the commentary text in the source file.

• The ‘macro prefactor’ contains text rewritten during macro preprocessing.

• The ‘macro postfactor’ contains the result of rewriting the macro prefactor during macro preprocessing.

We allow factors to be nested, even improperly nested. This corresponds to the fact that the same piece of text may be ambiguously ‘about’ more than one domain of discourse. A factor which it not itself further factored, that is, which properly containing no factors within it, may be called a ‘prime factor’. We see the relationship of the whole text to its prime and compound factors as analogous to the relationship between prime and compound members of an algebraic domain – hence the term ‘factor’.

3.2 Prefactors and postfactors.

Formal transformation processes apply to particular factors of a source. Below the *merge* and *integrate* steps are described which ‘multiply’ the transformed factor back into the whole. But it is frequently desirable to record both the input and the output of such transformations within the same factored source file. This typically arises when textual transformation directives have to be processed in order for the code factor to be valid according to the language’s grammar. If the results of transformation or other processes are then to be reflected back in the original source (that is, back through source inlining or macro expansion) that original source is still needed.

To deal with this situation some factors are pairs, distinguishing the “prefactor” and “postfactor”. The “pre” is “prior to transformation” and the “post” is “after transformation.” The original is available by suppressing the postfactor; the transformed is available by suppressing the prefactor.

Figure 3 contains a snippet of COBOL raw source, which illustrates several of the issues involved in processing ‘live’ legacy code. In card-image COBOL, the code factor generally appears in columns 6 through 72, and material outside these columns includes sequence numbers and maintenance history information. There are listing directives, source inclusion directives, comments, and ‘code’ all jumbled together.

3.3 Input processing

During input, the system accepts raw source, recognizes it, and prepares it for the later phases. Source files which pass this phase are in a normal form and ready for input to design recovery tools. The input phase must perform at least the following tasks:

• *Lexical normalization* is reducing unnecessary lexical variations. This includes especially the handling of continuation lines, for which there are about eight *different* rules between the three principal languages we process! (In PL/I, a statement or even a literal can be broken across lines without special indication; in COBOL a statement can be broken across lines but a broken literal must be marked with a combination of hyphens and extra quotation marks; in RPG, plus signs and minus signs in various columns, and somtimes other letters, indicate continuation of literals across cards. The complexity of continuation rules is generally in inverse propoertion to the dependence of the lexical design on fixed-format card images.) Continuation lines are used to construct a whole line, but the construction process must be undoable when integrating transformed results.

• *Source inlining* is inserting the contents of copy (include) files. This is generally done so that all downstream tools (design recovery, analysis, transformation) can operate on a single text. The inlining directive is treated as a ‘rewriting’ directive, replacing the directive with the contents of the file.

Again, it must be 'undoable' when integrating results (because transformations must be propagated back into the copy file.)

(The subject of a source inclusion directive is named in different ways in the jargon of different programmers. Generally speaking the Unix jargon refers to "include files" and marks this fact by storing some of the common ones in `/usr/include`. However, the mainframe world calls them "copy files" because of the directive `COPY` which causes inlining in COBOL.)

• *Macro preprocessing* is analogous to source inlining, because it analogously replaces directive text (macro invocations) by the source code it macro-replaces to. In our

```
…
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  REC1.
     05 FLD1      PIC ZZZZ9.


 01  MY-REC2.
     10 FLD1      PIC ZZ9.
…

 77  FLD-B        PIC Z9.
…
```

Figure 5 – Code with Copy Postfactor

system, the mother of all macro preprocessors (PL/I) is handled by interpreting the macro definitions directly and inserting the replacement text. But again, source transformations which affect the results of macro preprocessing then have to be somehow integrated with the original macro invocations. In C, for example, the macro call `putc(c,f)` is factored like this

```
…
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  REC1.
     05 FLD1      PIC ZZZZ9.
 {copy01  MY-REC2 COPY REC2.\

 01  {replREC\MY-REC2}repl.
     10 FLD1      PIC ZZ9.
…
}copy
 77  FLD-B        PIC Z9.
…
```

Figure 6 – Code/Copy Factor with Markup

(spacing supplied by `gcc -E`):

```
{macroputc(c,f)\_IO_putc ( c ,  f )}macro;
```

Figure 4 shows the example source program after factoring. It illustrates several things. The various factors are explicitly indicated. The copy factor is shown to have a prefactor and postfactor form, and the peculiar meaning in COBOL for this kind of source inclusion is illustrated by the macro replacement

```
99/01/03
* REC2 RECORD STRUCTURE
* THIS FIELD ADDED 00/01/02.
JAN2AJM
```

Figure 7 – Documentation Factors

factor. (This implicit replacement meaning for the COPY statement is an typical legacy-code wrinkle. It is no longer documented in the languages manuals but still implemented by the compilers for compatibility [1]. The given raw source also lacks a dot required by the COBOL syntax: this happens surprisingly often and is tolerated by many compilers. The factored code shows how the dot can be supplied during lexical normalization.)

3.4 Design recovery processing

During design recovery, that is, when extracting 'facts' from the code base which are to be reusable common input to analysis and transformation, the key is to be able to present to the tools only that part of the source which is required and relevant. Three kinds of design recovery task in our architecture illustrate this:

• *Code facts*, which is the largest and historically most important part of the recovered design, are extracted from the code proper. This kind of design recovery is quite like the semantic analysis phase of a compiler, and has similar input, namely, the code viewed as a term in a (context-free) grammar. Figure 5 shows the code/copy postfactor, which is the usual input.

• *Source structure facts*, which contributed especially to the success of our design analysis for millennium bug renovation,

```
…
DATA DIVISION.
WORKING-STORAGE SECTION.
01 REC1.
   05 FLD1 PIC 9(5).
01 MY-REC2.
   10 FLD1 PIC 9(3).
…
77 FLD-B PIC 9(2).
…
```

Figure 8 – Transformed Code Factor

are recovered from the code as well, but annotated or extended to indicate the source structure, especially the boundaries of copy files. Figure 6 shows the copy/copy factor with markup showing the copy file boundaries, which we use to extract the important "copy identifier" facts linking variables instantiated from the same copy file.

• *Maintenance history facts* have been a useful sideline: we have shown [10] that a large code base can be viewed in terms of the history of its development as recorded in the source. This kind of design recovery applies only to the comments! Figure 7 shows the documentation factor, input to this maintenance fact extraction.

3.5 Transformation processing

Our **transformation** tools are almost all written in TXL [7]. TXL is a special purpose programming language for expressing high-level source transformations. The (context-free) grammar of the input is an essential part of a TXL program; but well-designed grammar exploit TXL's features in such a way as to allow operations to abstract from irrelevant grammatical details. This is done, in part, by writing grammars which reflect the expected semantics of the input. Our transformation phases are applied to the code factor, parsed according to such a semantic grammar, and usually with side input from design recovery and analysis processes. (Side inputs are also provided as facts in a context-free interchange language. This combination of *facts* and *factored source* together represent our interchange format properly speaking.) Two general classes of transformation, *replacement* and *markup*, show the need for flexible handling of source text:

• *Replacement* is as in TXL: a whole compilation unit is input and processed as a 'program' in the official grammar of the source language, and nonterminal structures in the parse tree are replaced according to the requirements of the transformation. Figure 8 shows the result of a "remove zero suppression" transformation, which changed some of the leading-zero-suppressed-on-output variables to leading-zero-displayed-on-output. The formal description of this (omitted) is a simple TXL program which operates within the COBOL reference grammar: it need no be concerned with formatting details, comments, card layout, *etc.*, *etc.*, because of the factoring. Note that even the spacing no longer needs to follow the 'original'.

• *Markup* is factor refinement, generally of the code factor, to indicate regions of the code ("hot spots") of interest for a maintenance task: declarations, decision points, key subroutine calls, etc. Markup tools are specified in HSML, and a fuller discussion of appears elsewhere [5].

3.6 Output processing

On output, that is, when results of the foregoing are to be delivered to a user, the principal problem is to display them or print them *in the context of the original code*. Without exception, code-owners in our experience required reports and transformed results (even when the task was as simple as pretty-printing) to minimally different from the original. Two general kinds of output have been especially important for us:

• *Transformed original source* is the characteristic output of design recovery and transformation. The ideal result is formatted in the same way as the original ("warts and all") and constitutes a minimal line-by-line difference from the given source file. Even one character of needless difference is unacceptable in a large code base, where one measure of quality may be the rate at which source files are left 'unchanged' (and hence don't need to be examined during quality control). We provided this kind of output using "version integration and merging", described in more detail below. The transformation illustrated in Figure 8 is shown again, recompounded into the original source, by Figure 11.

• *"Hot-spot reports"* [5, 6] are the characteristic output of design analysis. A 'maintenance hot spot' may be defined as "a region of code requiring attention during maintenance". The success of a large maintenance task, done by hand or partly or wholly automated, depends in large part on finding its hot spots: if there are few, and they're identified accurately, maintenance can be efficient. A Hot-Spot report for a source file is, essentially, excerpted from the original source to show the hot spots of a required maintenance task. The excerpts must be relative to the true original warty source, but annotated with reasons and remarks generated during design analysis.

```
…
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  REC1.
     05 FLD1    PIC {diffZZZZ9\9(5)}diff.


 01  MY-REC2.
     10 FLD1    PIC {diffZZ9\9(3)}diff.
…

 77  FLD-B      PIC {diffZ9\9(2)}diff.
…
```

Figure 9 – Factored Differences

4. Factor operations

There are four operations on factored text which are used systematically to present the appropriate factors to task-specific tools, and to recombine them properly into the required results.

```
{code…
{seqn000050}seqn DATA DIVISION.
{seqn000060}seqn WORKING-STORAGE SECTION{mdot\.}mdot
{seqn      }seqn {drct EJECT}drct
{seqn000070}seqn 01  REC1.
{seqn000080}seqn     05 FLD1      PIC {diffZZZZ9\9(5)}diff.     {init99/01/03}init
{seqn000090}seqn {copy01  MY-REC2 COPY REC2.\
{seqn      }seqn{cmnt* REC2 RECORD STRUCTURE}cmnt
{seqn      }seqn 01  {replREC\MY-REC2}repl.
{seqn      }seqn     10 FLD1      PIC {diffZZ9\9(3)}diff.
…
}copy
{seqn000095}seqn{cmnt* THIS FIELD ADDED 00/01/02.}cmnt
{seqn000100}seqn 77  FLD-B        PIC {diffZ9\9(2)}diff.       {initJAN2AJM  }init
…
}code
```

Figure 10 – Merged Differences

The three basic ones are ‘factor projection’, ‘factor difference’, and ‘factor merge’. The latter two are combined into a fourth, ‘factor integration’, the characteristic postprocessing step after a transformation. In conjunction with language-independent markup specifications these allows the same formal design analysis and reporting to be applied to code bases in any language, subject only to the language-specific support for initial factorization.

4.1 Factor projection

Factor projection, also known as ‘factoring out’, produces a factored text which is a projection of a given text: it suppresses some factors and leaves the rest. It comes in two flavours, with markup (as in Figure 6) and without markup (as in Figure 5).

4.2 Factor integration

After transformation, we have two ‘versions’ of the same factor: the old one from the original text (as in Figure 5) and the new one resulting from the transformation (as in Figure 8). In order to produce a transformed version of the original source, it’s necessary to ‘install’ the transformed factor back into the original factored source. This is done in two stages: factor difference and factor merge, as described below.

Factor integration process is usually called by us ‘version integration’ because it is the integration of a new version of the code into the old matrix of comments, directives, layout, and so forth. It is also affectionately known as ‘backpatch’, having been originally conceived as a method of ‘patching’ the raw source under control of the transformed code.

4.3 Factor difference

The first stage of factor integration (see above) is to factor the differences between the new and the old versions. A standard difference algorithm [9,11] can provide the basis of this stage. Figure 9 shows the factored differences resulting from the example transformation.

This stage works well, in our experience, when the changes are relatively small. For millennium bug renovations, typical changes were the local insertion of a few lines of code. When the changes are large, or involve code movement, difference algorithms don’t work very well, and then transform tools themselves have to produce the factored differences explicitly.

4.4 Factor merge

The second stage of factor integration is the merging of two factored texts into one. Factor merge requires one factor identically in common between the two texts, in order to synchronize the merging of the factors which are *not* in common. Figure 10 shows the result of merging Figure 9’s difference factors into the original factored text (see Figure 4). Note that in this case the replacement prefactor is, indeed, identical to the code/copy factor which was input to the transformation; and the replacement postfactor is merged with all the factors (comments, sequence numbers, directives, *etc.*) which were suppressed to produce that input in the first place.

4.5 Factor markup

The factors in a text are represented in practice by a various markup conventions. The markup is also character data, and so when markup is part of the factor-output (as in Figure 6) it can be parsed along with the source code using ‘base grammar overrides’ [7]. The notation used in this paper is similar to that

SAMPLE.COB

```
…
000050 DATA DIVISION.
000060 WORKING-STORAGE SECTION
       EJECT
000070 01  REC1.
000080     05 FLD1      PIC 9(5).                                    99/01/03
000090 01  MY-REC2 COPY REC2.
000095* THIS FIELD ADDED 00/01/02.
000100 77  FLD-B        PIC 9(2).                                      JAN2AJM
…
```

REC2.CPY

```
      * REC2 RECORD STRUCTURE
      01  REC.
          10 FLD1      PIC 9(3).
…
```

Figure 11 – Reconstructed Source

used in practice, except that lexical conventions are assumed instead of **boldface**. In fact a variety of conventions are used in order to distinguish factor markup from code. So far, the convention curly-brace, factor-name, one space, has served well, but that is because our target languages were originally EBCDIC-encoded, and hence lack the curly-brace. These syntax details can vary while preserving the concepts. The use of syntax extensions to indicate factorization has the advantage of allowing the processing of factored text using ordinary tools (like grep and sed) and preserving the readability.

5. Conclusion

We have shown how a unified view of source text in which subsequences of text, called *factors* can be the subject of automated design recovery and transformation. By separating the source code into lexically standard substreams, indicated by markup, it is possible to design and build source processing tools for design recovery which have simply stated requirements.

References

1. IBM, *IBM VS COBOL II Migration Guide for MVS and CMS* , Order no. GC26-3151-00, 1993
2. ISO SGML 86. *Information processing – text and office systems – Standard Generalized Markup Language (SGML)*, ISO 8879-1986, International Organization for Standardization (1986).
3. J. R. Cordy, C. D. Halpern, E. Promislow. "TXL: a rapid prototyping system for programming language dialects", *Computer Languages* 16,1 (Jan. 1991), pp 97-107.
4. J.R. Cordy, K.A. Schneider. Architectural design recovery using source transformations. In CASE'95: Workshop on Software Architecture, Toronto, Canada, July 1995.
5. J. R. Cordy *et al.* "HSML: design directed source code hot spots", Ninth IEEE International Workshop on Program Comprehension, Toronto, May 2001
6. J. R. Cordy. *The DRI Legasys Group LS/2000 Technical Guide to the Year 2000*, TR. ED5-97, Legasys Corp. (Kingston) and IBM Canada (Toronto), 1997.
7. J. R. Cordy *et al. The TXL Programming Language / Version 10*, TXL Software Research Inc., `www.txl.ca/txldocs.html`
8. T. R. Dean *et al.* "A scalable design recovery and migration system", in preparation.
9. J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report Computing Science TR #41, Bell Laboratories, Murray Hill, N.J., 1975.
10. A. Malton *et al.* "Exploring the maintenance history of source code bases", in preparation.
11. E. Myers. An O(ND) difference algorithm and its variations. Algorithmica, 1(2):251--266, 1986
12. H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, Dec. 1993. pages 33–43, Los Alamitos, 1997. IEEE Computer Society Press.
13. K. Wong. RIGI User's Manual, Version 5.4.4. `http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml?Download`, June 1998.
14. Richard C. Holt, Andreas Winter, Andy Schurr. GXL: Toward A Standard Exchange Format, May 2000, WCRE 2000: Working Conference on Reverse Engineering, Brisbane, Australia, Nov 6, 2000.
15. R. Kazman, S. J. Carriere, Playing Detective: Reconstructing Software Architecture from Available Evidence, Journal of Automated Software Engineering, 6:2, April, 1999, 107-138.
16. J. Ebert, B. Kullbach, A. Winter. GraX – An Interchange Format for Reengineering Tools. In Sixth Working Conference on Reverse Engineering. IEEE Computer Society, Los Alamitos, 89–98. 1999.