Proc. ICSM 2001 - IEEE International Conference on Software Maintenance, Florence, November 2001, pp. 622-631

Using Design Recovery Techniques to Transform Legacy Systems

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

Legasys Corporation, Kingston, Ontario, Canada
{dean,cordy,kas,malton}@cs.queensu.ca

Abstract

The year 2000 problem posed a difficult problem for many IT shops world wide. The most difficult part of the problem was not the actual changes to ensure compliance, but finding and classifying the data fields that represent dates. This is a problem well suited to design recovery. This paper presents an overview of LS/2000, a system that used design recovery to analyze source code for year 2000 risks and guide a source transformation that was able to automatically remediate over 99% of the year 2000 risks in over three billion lines of production IT source.

1. Introduction

Design Recovery is an automated approach for recovering a design model from source code artifacts [1,2]. Legasys Corporation was formed in 1995 to apply design recovery techniques to very large legacy information technology systems.

The Y2K problem required information technology departments to ensure that their software would function correctly in the year 2000 and beyond. This was particularly an issue when years were being represented as 2–digit numbers (…,98,99,00,01,…) with the century being implied to be 1900. An estimated 400 billion dollars was spent worldwide to remediate and test hundreds of billion lines of program source code for the Y2K problem. Over 80% of these lines were written in COBOL.

The size and scope of the Y2K problem made it ideal for automated design recovery techniques. Accurately identifying and classifying the dates in a system turned out to be a complex and subtle problem, requiring careful design-level analysis of the source code. However, once the dates were found, determining which dates were being used incorrectly and making the appropriate changes was relatively straight forward.

The result of our application of design recovery techniques to the Y2K problem was LS/2000, a highly automated process that operated with a minimum of human intervention on COBOL, PL/I and RPG source code. LS/2000 was licensed to IBM Canada for exclusive use in Canada and to several other Y2K vendors world wide. In all, more than 3.3 billion lines of source code was remediated or independently verified and validated with the LS/2000 process.

This paper presents presents an overview the LS/2000 solution for the Y2K problem with emphasis on its use of automated design recovery and analysis techniques. The Y2K problem is a well understood case that is typical of a class of maintenance tasks. The approach discussed in this paper, with minor modifications, was applied to several hundred million lines of code to assist some of these other maintenance tasks in 1999 and 2000 as well.

2. System Architecture

Figure 1 shows the system architecture of the LS/2000 system using a variant of the software architecture notation of Dean and Cordy [3]. (In this paper we use the Dean/Cordy notation in preference to UML since it was the one actually used in the LS/2000 project.) In this notation, boxes represent artifacts, ovals represent processes, arrows represent data flow, dashed arrows represent either input or output, and thatched boxes represent tables.

The LS/2000 process is divided into five phases: Import; Design Recovery; Date Analysis; HotSport Markup and Transformation; and, Version Integration.

For each program, original source code is converted by the *Import* phase to produce two internal forms of source. Both of these forms have all of the copy books (i.e., include files) inlined in the code. The first internal form, the *UID factor*, is a free form version of the code suitable for automated processing. In this form lexical details have been removed, syntax has been normalized and all data fields in the system have been given globally unique names (UID = Unique IDentifier). The second internal form, the *lexical factor*, retains the original formatting of the source code, but each line has been annotated with the UIDs of the fields referred to on that line. It is used as a part of the user interface in the Date Analysis phase. Since the *Import* phase removes some information, namely the copy file

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).
** Author's current address: Department of Computing & Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6.*
+ 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.

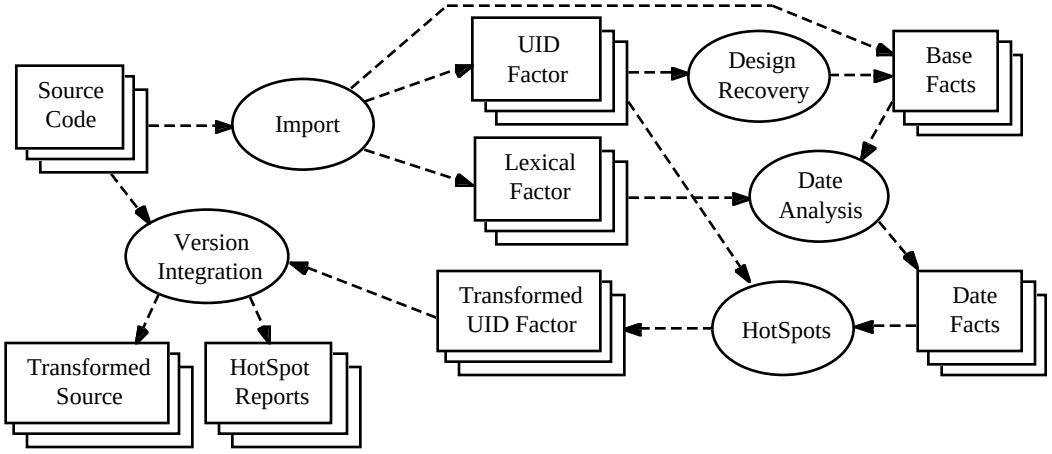

Figure 1. System Architecture

boundaries, it also generates some of the base facts of the design database (i.e. the source file facts).

The *Design Recovery* phase generates a base design recovery model as a Prolog-style database. Each fact in this model is directly related to a source code artifact. For example, the *Picture* fact indicates that a given data field is declared with a given COBOL or PL/I "picture". ("Pictures" describe COBOL's elementary data types).

The facts generated by Design Recovery are used by *Date Analysis* to derive facts about which fields are dates and what date format those fields use. This is an imperfect process that requires some human intervention. The date facts are used to guide a markup and transform of the internal form of the source code by the *Hot Spots* phase. Finally, the *Version Integration* phase merges the formatting and comments of the original source code from the lexical factor to produce the final transformed code and reports.

Each of the phases of LS/2000 is discussed in more detail in the following sections. Although there are slight differences in the phases for each language (COBOL, PL/I, RPG), in this document we will focus on the COBOL version of the process.

2.1 Import

The Import phase takes the original source code and produces two internal forms and some of the base facts. Figure 2 shows the structure of the Import phase. It, in turn, is composed of several sub-phases. The first of these, *Front End*, takes each of the source code program files and inlines all of the copy files referred to by the program. At the same time, it removes all of the comments, the REMARKS section (for older dialects of COBOL), sequence numbers and maintenance initials.

Since this removes the copy file boundaries, any data fields that are included as a consequence of a COPY statement are "mangled" to encode the name of the copy file, and in the case of COPY REPLACING statement, the original unreplaced name of the data field.

Most of the internals of the LS/2000 system were implemented in TXL [4,5,6], a functional transformation language that works on parse trees given by a grammar. If the data hierarchy is to be made explicit in the parse tree, then the languages processed by LS/2000 are not context free. For example, in COBOL and PL/I, the subfields of a record structure are given by the level number, where larger numbers represent deeper nesting within the data structure.

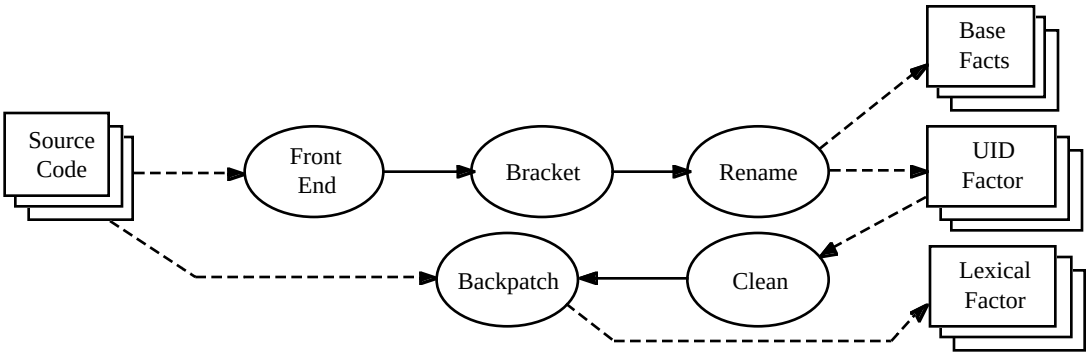

Figure 2. Structure of the Import Phase

The next phase of Import, *Bracket*, is responsible for the context sensitive parse necessary to resolve this problem. Subfields of records are "bracketed" (hence the phase name) by the Bracket phase using square brackets [ ], characters that are not part of the COBOL language definition, to explicitly represent record structure boundaries. This results in a version of the language where record nesting structure can be easily recognized using a context-free parser.

The *Rename* phase of Import traverses the data hierarchy of each program, and gives each data field in the system a unique name. This unique name encodes the source file name, the program name within the file and the record and the position of the field within the record hierarchy. All references to data fields in the program are modified to use the unique name given in the data hierarchy. The form of a unique id (UID) is:

data_part – program_part `filename`

where *data_part* and *program_part* are sequences of identifiers. The file name is enclosed in back quotes ` ` to hide any special characters permitted by the file system that would not normally be part of an identifier. To permit the original form of the name to be easily recovered at the end of the transformation process, the UID is encoded in a source code factor [7] of the form:

[*uid # original_code*]

For example a simple record in the program A in the file "A.CBL" with the following declaration:

```
01 DATE-REC.
   05 YY                                PIC 99.
   05 MM                                PIC 99.
   05 DD                                PIC 99.
```

would be transformed by Bracket and Rename into a record of the following form:

```
01 [ DATE-REC - A `A.CBL` # DATE-REC] . [
   05 [YY DATE-REC - A `A.CBL` # YY]    PIC 99.
   05 [MM DATE-REC - A `A.CBL` # MM]    PIC 99.
   05 [DD DATE-REC - A `A.CBL` # DD]    PIC 99.]
```

and a reference in the code such as:

```
YY OF DATE-REC
```

would be converted by Rename to:

```
[YY DATE-REC - A `A.CBL` # YY OF DATE-REC]
```

The Rename phase also unmangles the names for fields imported from copy files. Each of these fields is given a *CID* (Copy IDentifier) which is similar in structure to the UID. The CID uniquely identifies the field within the copy file, and since it includes the copy file name, within the system. The relationship between CID and the UID is asserted in the design database using the *CopyID* fact.

The *Clean* phase of Import is a general purpose program that removes the data hierarchy bracketing and the UID factors from the code. When used in the Import phase, it transforms the UID factor into a markup of the field. This markup has the form:

{"*UID*" *original code* }"*UID*"

The declaration of YY in the previous example would now appear as:

```
{"YY DATE-REC - A `A.CBL`" YY }"YY
        DATE-REC - A `A.CBL`"
```

Both declarations and references to data fields are transformed in this way.

Backpatch is also a general purpose phase used in several places in the process. It merges the formatting of the original source code with the cleaned free format internal code. When used in the Import phase, it generates a version of the original code with all copy files included in which each line is annotated with the UID for any fields that occur on that line. This annotation takes the form of a null byte with the high bit set (0x80 in hexadecimal) at the end of the source code line, followed by all of the UIDs for the identifiers on that line. Additional markup on the line is used to indicate copy file boundaries. This form is used later in the analysis and reporting stages of the process.

2.2 Fact Extraction

All other base facts (other than CopyID facts) of the design database are generated by the *Design Recovery* phase. Base facts are facts that are (or can be) directly generated from the source software artifacts. These facts represent the type and storage allocation of data fields, and how the fields interact with each other by data movement, comparison and so on.

Figure 3 shows the structure of the Design Recovery phase of LS/2000. The phase reads from the uniquely named internal format and generates Prolog-style base facts. The *Data Facts* part of the phase is responsible for generating facts from the DATA division of the program. These facts represent the data hierarchy (parent–child, level number), the data type (PICTURE, USAGE, and JUST clauses), initial values, array dimensions (OCCURS clause) and other declared relationships between the data fields (REDEFINES clauses, and file records).

The *Size Facts* subtask compresses the information from several of these base facts into a *FieldSize* fact and an *Offset* fact. The FieldSize fact gives the total number of bytes occupied by the field, the number of digits before and after the decimal point (for numeric data) and a code representing the base type of the data in the field. This fact provides a concise representation of facts representing the PICTURE, USAGE, JUST, and OCCURS clauses of COBOL. It also provides a somewhat language independent view of the storage of a field, permitting a more general Date Analysis phase for all three languages. The Offset fact gives the position of a data field within its record. Although both the FieldSize and Offset facts are generated from other facts and not from the source, they convey no more information than the original source artifacts and thus we consider them base facts.

The only facts directly extracted from the PROCEDURE division of the program (i.e. the executable statements of the program) are *Move* facts and *Compare* facts. These facts abstract assignment (the MOVE statement) and comparison between fields respectively. While there are other relationships

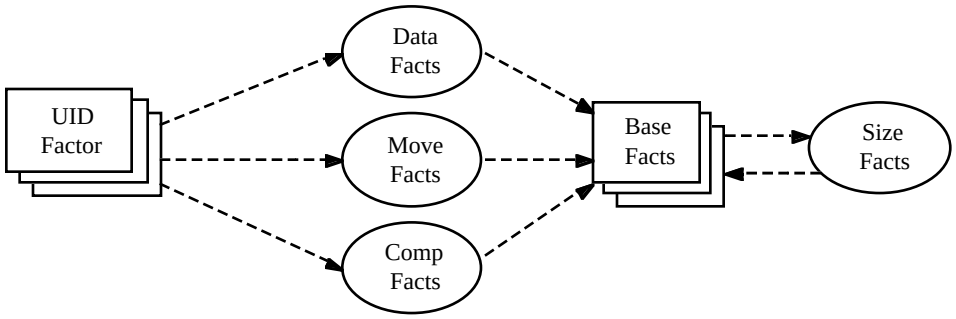

Figure 3. Structure of Design Recovery

between fields in the PROCEDURE division, these facts proved sufficient to discover which fields are dates and the formats of the dates. Movement of literal values to data fields and comparisons of data fields to literal values were also captured by the Move and Compare facts.

Some statements other than MOVE statements were modeled as MOVE statements in the recovered design. In COBOL, certain special forms of the READ and WRITE statements imply implicit moves. These implicit moves were modeled explicitly in the design fact base. Similarly, the ACCEPT … FROM DAY/DATE statement, which generates the current date as a Julian day (YYDDD) or an ISO date (YYMMDD) and assigns it to the target data field, was modeled as a MOVE from a suitably named predefined data field recognizable by the Date Analysis phase.

For reasons that will be explained in the next section, there was no need in LS/2000 to recover facts for procedure calls or facts that relate the arguments of a call to the parameters of the called program.

2.3 Date Analysis

The *Date Analysis* phase is responsible for generating a set of derived facts that identify which data fields represent dates and what format of date is stored in them. This is a difficult and challenging problem that is not entirely automatable. Figure 4 shows the structure of Date Analysis.

The inference of dates is seeded using *Naming Convention*. This part of Date Analysis uses the declared name and size of the data field in conjunction with a set of naming convention tables to identify data fields that have a high probability of representing dates. If Naming Convention can also determine the format of the date from the name, for example if a data field is named ACCT-YYNNN and is declared with a picture of 9(5), that is, five numeric digits, then it also associates the date format with the data field. The format is referred to as the *date type* of the data field. In the above example, the date format is probably a Julian date, that is, a two digit year followed by a three digit day number within the year. If Naming Convention believes that a data field is a date, but cannot determine the format, then the special date type UNKNOWN is used. Fields that are known to not be dates were given the date type IGNORE.

The rules used in the naming convention table allowed a variety of pattern matching primitives. Fields could be matched based on substrings at the beginning, middle or end of the field name. Since it is common for COBOL (and PL/I) fields to be named as a sequence of words separated by hyphens (underscores in PL/I), pattern matching operators to match words within field names were also provided.

The naming convention tables specified both positive and negative clues to look for. For example, the sequence UPDATE in field name is not an indication that the field is a date, even though the substring DATE appears in it. Rules were ranked

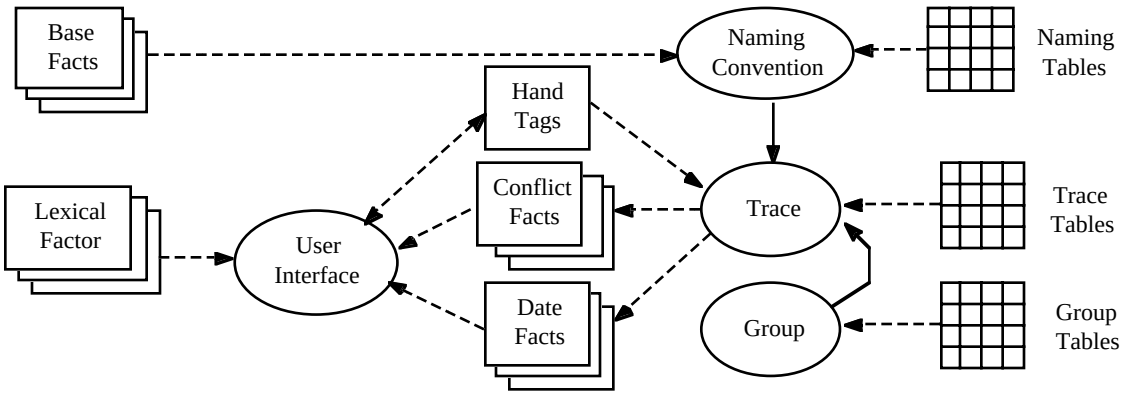

Figure 4. Structure of Date Analysis

using set of *priorities*. The purpose of this was twofold. It allowed for general rules that gave the UNKNOWN date type, and more specific rules with a higher priority providing specific formats. It also allowed us to rank different naming conventions according to their strength. The priority system of rules, while powerful, proved difficult to use, requiring a fair amount of tuning. A sanity filter also prevented any name longer than 20 characters from being identified as a date.

The initial evidence collected by Naming Convention along with the recovered design factbase is given to the *trace engine*. The trace engine has two components. The first, called *Trace*, is responsible for following references between data fields. Traced references include the Move, Compare and Redefines facts. The source code is not used, and the facts do not contain any data flow dependencies. Thus a pure reference model is used, not a data flow model. The algorithm begins with each of the data fields that are currently marked as Dates, and considers all data fields that are referenced by these fields to determine if they might be dates, continuing until all reference chains have been examined.

For Move and Compare facts, if a referenced data field has the same number of digits, it is considered a trivial reference, and the fields are considered to have the same date type. If the two fields have different sizes, then the *Date Trace Tables* are used to determine the types of the related fields.

Date trace tables contain two kinds of entries. The first kind is a *type inference* entry. In this case, one of the data fields has already been assigned a date type, and the other has not. The table then provides the date type for the unassigned field (if possible). The other kind of entry is a *sanity check* entry. These apply when both data fields have already been given date types. In this case the table provides a sanity check to determine if the relation is legitimate. Sanity check entries must be much more liberal than type inferences. For example, consider the statement:

```
MOVE A-DATE TO B-YY.
```

Where A-DATE is a field of length 6 and B-YY is a right justified field of length 2. If we already know that B-YY has a date type of YY, then A-DATE may be an MMDDYY (two digit month-day-year), or a DDMMYY (two digit day-month-year). The trace table cannot contain a type inference rule that infers the date type of A-DATE, because it could be either one of the possibilities. However, the tables must contain sanity check entries for both possibilities, since both are meaningful assignments between date types.

The date inferences resulting from Trace as well as the design factbase are then passed to the *Group Analysis* part of the tracing algorithm. This algorithm uses another set of tables, the *Date Group Tables*, to attempt to infer the date types of group fields (i.e., records) from the types of their member fields. For example, given the following COBOL group definition for POST-DATE:

```
01 MAIN-REC.
   …
      10 POST-DATE.
         15 POST-DATE-YY          PIC 99.
         15 POST-DATE-MM          PIC 99.
         15 POST-DATE-DD          PIC 99.
   …
```

where we already know that the date types of the subfields POST-DATE-YY, POST-DATE-MM and POST-DATE-DD are two digit year, month and day respectively, we can infer that POST-DATE has the date type YYMMDD (two digit year-month-day).

The date analysis engine of LS/2000 traced date types for the entire application at once, in order to permit date information to be inferred between programs. In particular, every instance of a data field declared in a copy file included in multiple programs is combined into a CopyID cluster that is treated as a single entity by the date trace algorithm. This is based on the assumption that if a data field is a date in one program, then it is a date in all programs. This was a *very* strong inference method. We performed several experiments adding facts linking the arguments in a CALL statement to the parameters of the called programs. None of these extra facts were found to add any information to the trace of date types - the CopyID clusters were sufficient to transfer date inference information from one program to another.

The date type inference algorithm in LS/2000 was good, but by no means perfect. Various program constructs could trip it up, preventing data fields from being assigned consistent dates. The imperfections showed up in one of two ways.

The first was the case where data fields that were found to be dates could not be given a format (date type). These were given the date type UNKNOWN. Unknown dates were not traced, so these were the result of the Naming Convention part of Date Analysis. In this case the system did not do well because it could not trace the dates.

The second was the case where a program contained a move or comparison in which the date types of the two fields were incompatible. This resulted in the generation of *Conflict* facts, which also limited the trace. Conflicts could arise for several reasons. The first reason was that a potential date type inference rule may not have been considered strong enough to include in the trace or group tables. The second is that authors of the application being analyzed may have used a temporary buffer for both dates and other business types (such as account codes). The last possibility is that there may be a bug in the application code.

LS/2000 allowed human intervention to resolve both unknown date types and date type conflicts. A web-based interface was provided in which a human *date analyst* was presented with a list of data fields whose date types were unknown or in conflict. This interface included hyperlinks to the data fields related to each of the questionable fields, including their declarations and uses in the original code.

The *Lexical Factor* of the source contained the source code in its original formatting. Each line was annotated with the UIDs of the fields on the line. These annotations were used by

the user interface to present each of the fields in context to the analyst. The annotations were invisible to the analyst, who saw the code only in its original form.

When the analyst had made a decision on the resolution of a field, he/she assigned a date type to the field. This assignment was represented internally as a *Hand Tag* fact. These hand tags were read by the trace engine and used to override any inference made by Naming Convention, or inferred by the trace or group algorithm.

In some cases, even the human date analyst could not determine the date type of a field because there was too little information available in the code. If the field was not involved in any comparisons, or used as in the key to a file, merge or sort, then the field could not possibly be a critical date and was ignored. If it was, then it would be left as an unknown field. All such fields were reported to the client as requiring further application knowledge to fix.

2.4 Hot Spot Markup and Transformation

The *Hot Spots* phase of LS/2000 was responsible for using the date information discovered in the Date Analysis phase and applying it to the application code. In our experience, the vast majority of cases where date fields were used in a Y2K sensitive manner could be transformed automatically. The small number of remaining cases, which require application knowledge to properly handle, were marked as Y2K sensitive and provided in a report along with those changes automatically made by the system.

The strategy taken by our approach was an aggressive markup of the Y2K risks, followed by a conservative transformation. It was felt that having some false positives in the reports (that would not be changed by the transform) were better than having false negatives (missed Y2K risks).

Figure 5 shows the structure of the Hot Spots phase of LS/2000. *Extract Facts* is a program used to extract a subset of the facts from the recovered design fact base that match a given criterion. It is invoked twice in Hot Spots. The first time it is invoked, it is used to obtain all of the Date facts for the application program being marked up.

The Date facts along with the UID factor of the program is processed by *Hot Spots Markup*. This program identifies those places in the code where a field that has been identified as a date is used in a Y2K sensitive manner. It also identifies the context of the use (the statement containing the use), the declaration of the date field, and the group in the data hierarchy that contains the field. This identification takes the form of a markup similar to the UID markup used by Clean in the Import phase.

The Y2K sensitive situations that were identified by Hot Spots Markup were comparisons between dates, comparisons between dates and literals, use of dates as keys for indexed files in sorts and merges, use of dates embedded in file keys, use of dates in arithmetic, and non-trivial initial values for date fields.

Hot spots identified points of actual risk, which in our experience was less than 0.4% of the source lines in an application, and typically contained in less than 40% of the programs of the application. Each hot spot fell into one of four categories:

- Those that had already been converted by previous maintenance
- Those that involved safe or benign side-effects
- Those that were automatically converted by LS/2000
- Those that required application knowledge to resolve (statistically less than 1% of the hot spots, or less than 0.004% of the lines in the application)

The second time *Extract Facts* is run, it extracts only those Date facts with a format that includes a year in the date type. For example, fields that contain only the month would not be included. These facts along with the marked up source generated by Hot Spots Markup were passed to Hot Spots Transform.

Hot Spots Transform performs a conservative transformation of the code to resolve Y2K risks. For example, comparisons between dates only makes sense if the years are the leading digits in the transform. Using an internal list of the date types for which transforms were supported, Hot Spots Transform examined each of the hot spots identified by Hot Spots Markup, and if it had a transform for that particular case, it was applied. Several cases were handled. These included comparisons of dates, increment and decrement of dates, use of dates as looping constraints and sorts based on dates. The use of dates in keys in files was not in general automatically remediated, but several

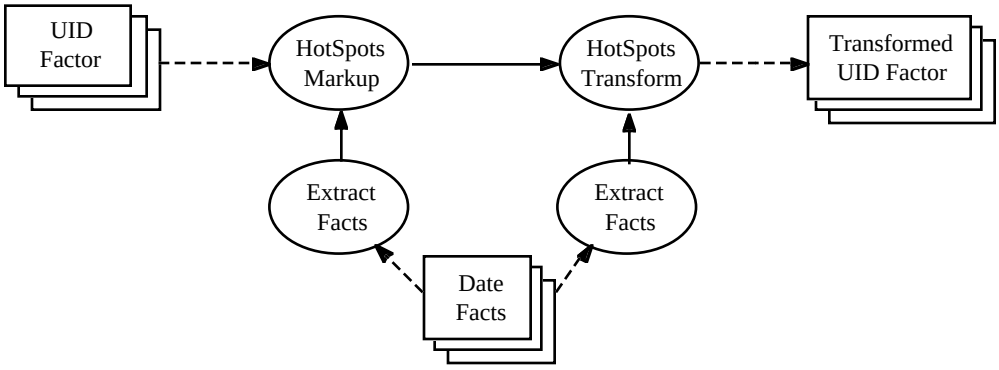

Figure 5. Structure of Hot Spots

other transforms for I/O of dates were supported, including removal of zero suppression from output year fields (the COBOL picture character "Z" causes a space to be printed instead of a 0).

All LS/2000 transforms were based on a static windowing solution to the Y2K problem. In this solution, a given date, the "rollover" date, is chosen as a pivot. All dates with a value greater than the pivot date are considered to be in the previous century and all dates less than the pivot date are considered to be in the next century. For example, if the chosen pivot were 30, then 50 would be considered to represent 1950 and 25 would be considered to represent 2025. There has been some objection to this approach. The main objection is that the remediation must be done again when the pivot year is approached in the next century.

Our solution avoided this problem using a "sliding window' solution, in which the pivot date and the century dates (19 and 20) are defined in a global copy file that is included by all remediated programs. When the pivot date approaches, the values in the copy file is changed and the application is recompiled. Since the century dates are also included in the copy file, when the next century change approaches, the 19 and 20 in the copy file may be changed to 20 and 21, moving the next pivot to the middle of the 22nd century.

LS/2000 supported several different pivots in the same remediated application. One reason why more than one pivot might be required is to treat business dates separately from birth dates. The LS/2000 system provided an interface to allow the analyst to identify which pivots should be applied to each date, and to specify the initial pivot values to be placed in the pivot copy file.

LS/2000 transforms were designed to be applied locally at the actual code location of the hot spot. This minimized the amount of code changed, and localized the change to the point of actual risk. The remediation transform designed for LS/2000 used the built in functions provided by the newest version of COBOL (COBOL for MVS and VM), which allowed a solution in which code was remediated directly inside the offending comparisons. For clients that had not migrated their applications to the latest version of COBOL (or those that preferred a different solution), three other solutions were provided. Each of these solutions used additional statements inserted just before the sensitive statement to resolve the problem, adding several temporary buffers to hold the converted versions of the dates.

The first of these solutions used inline COBOL arithmetic statements to convert the dates. For example, consider the following code snippet:

```
01 FISC-DTE-JUL                PIC S9(5) COMP-3.
77 WS-FISC-DTE-JUL              PIC S9(5) COMP-3.
IF FISC-DTE-JUL > WS-FISC-DTE-JUL
    PERFORM FISCAL-DATE-PROCESS.
```

This transform would result in the following code:

```
01 FISC-DTE-JUL                PIC S9(5) COMP-3.
77 WS-FISC-DTE-JUL              PIC S9(5) COMP-3.
77 Y2K-FISC-DTE-JUL             PIC S9(5).
77 Y2K-WS-FISC-DTE-JUL          PIC S9(5).
ADD ROLLDIFF-1-YYNNN FISC-DTE-JUL
  GIVING Y2K-FISC-DTE-JUL
ADD ROLLDIFF-1-YYNNN WS-FISC-DTE-JUL
  GIVING Y2K-WS-FISC-DTE-JUL
IF Y2K-FISC-DTE-JUL > Y2K-WS-FISC-DTE-JUL
    PERFORM FISCAL-DATE-PROCESS.
```

Since both dates are in the same window (use the same pivot date), a full conversion to 4 digit years is not necessary. Instead a constant is added to the date (ROLLDIFF-1-YYNNN) that will normalized the date within the window. For example, if the first window is 30, then the ROLLDIFF-1-YYNNN will be 70000. If WS-FISC-DTE is 01313 (November 9, 2001), 71 years after the pivot date, then adding 70000 to the date gives 71313, or the location of the date within the window given by the pivot. A date of 990101 (Jan 1, 1999) in WS-FISC-DTE-JUL gives a result of 690101 in Y2K-WS-FISC-DTE because of the left truncation inherent in COBOL arithmetic. Since the date in Y2K-FISC-DTE-JUL is less than the date ins Y2K-WS-FISC-DTE-JUL (1999 is before 2001), the paragraph FISCAL-DATE-PROCESS is performed.

When the dates compared are in different windows (i.e. different pivots are used), then the temporary buffers created are large enough for a the 4 digit year version of the date and the inserted code performs a full conversion. The full conversion is also performed by the other two transforms provided by LS/2000.

Instead of inserting the arithmetic code in line, the other two solutions provided called a routine for each of the conversions. This routine was given parameters providing the date to be converted and the window to use for the conversion. The first of these solutions use a PERFORM statement to call a paragraph in the same program, which was inserted by the transform. The other solution used the CALL statement to call an external program that provided the conversion.

2.5. Version Integration

The *Version Integration* phase of LS/2000 was responsible for merging the changes made by Hot Spots with the code formatting from the original code. It was also responsible for producing a set of reports that detailed the changes made and any risks that were detected but not remediated. Figure 6 shows the structure of the Version Integration phase. As with the Import phase, Clean is used to remove data hierarchy bracketing and the UID factors. The factors used to mark up hot spots and the transformations remain in the code.

The Backpatch program, used to produce the formatted UID in the Import phase, is used again to merge the formatting in to the transformed source code. To do this, it applies a difference algorithm [8,9]. The formatting for inserted code follows standard COBOL formatting conventions, with initial indentation based on the format of the surrounding original code.

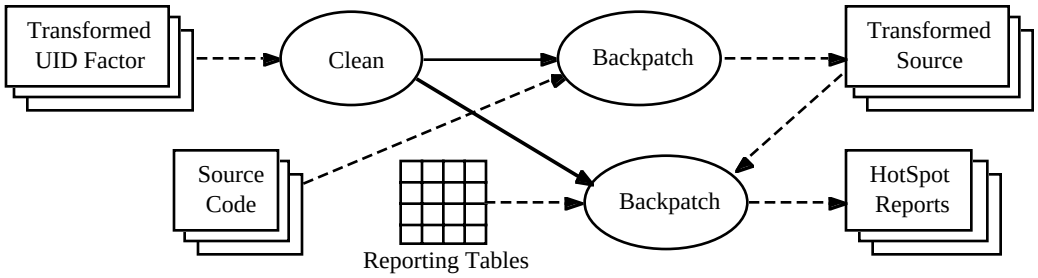

Figure 6. Structure of Version Integration

A set of user preferences guided the integration process. The original code may be included as comments (which eases the reversal of any transform that may have been applied in error). Another option asks LS/2000 to generate new sequence numbers for inserted lines. Comments may be added to the top of the program giving the date of the conversion and documenting the fact that LS/2000 was used to make the change. If the client was upgrading all of the code to operate with compilers later than COBOL I compilers, the obsolete paragraphs in the identification division (AUTHOR, REMARKS, etc.) could automatically be commented out during remediation.

The final transformed source was merged again against the transformed code containing the hot spot markup to produce a set of hot spot reports. These reports took the form of elided source code, with the hot spots elements highlighted. Figure 7 shows a sample of one such report. In the report, declarations of fields involved in hot spots and any unremediated hot spots are shown with a single arrow highlighting the line on the right hand side of the report. Any changed lines are highlighted with a double arrow. The hot spot report also identifies the source file and the line within the source file. If requested, contextual information can be provided, both for the definitions of the fields and for the statements containing the hot spots. These options are all governed by the Reporting Tables used by Backpatch when producing the reports.

Our clients found the reports invaluable for several reasons:

- The reports provide a guide for the application programmer when certifying each of the programs in the application
- They focus the effort of the application programmers on the points of actual exposure
- They provide a checklist of potential failures to be tested.

3. Implementation

The original implementation of the LS/2000 system was on Apple Power Macintosh computers using the MPW command line environment. However, when the system had evolved to the state that it could be licensed, a user interface was needed. The tool set was ported to run on IBM RS/6000's running AIX.

Rather than build a custom graphics interface, we designed a web driven interface. This interface, generated by CGI programs invoked by the web server, gave the analyst the ability to run the phases of the process and to interact with the Date Analysis engine. This proved to be an enormous win. We did not have to deal with the details of graphical interface programming, and our licensees were free to place whatever platform they wished on the analyst's desk, as long as a recent version of Netscape Navigator was available. Any direct interaction with the interface was implemented using JavaScript on the web pages.

The application systems to be analyzed and remediated were loaded onto a file server, which also ran the web server.

```
Program: XYEGPROG
Line  Program Source Line                                                    HS Src File
----  --------------------                                                   -- --------
26    002600      16  FISCAL-DATE-JULIAN      PIC S9(5) COMP-3.              <- XXCOPYDJ

52    005300            24  WS-FISCAL-DATE-JULIAN   PIC S9(5) COMP-3.      <- XYEGPROG

63              COPY LS2KROLL.                                               <= XYEGPROG
64              77  Y2K-FISCAL-DATE-JULIAN     PIC 9(5).                    <= XYEGPROG
65              77  Y2K-WS-FISCAL-DATE-JULIAN  PIC 9(5).                    <= XYEGPROG

232                ADD ROLLDIFF-1-YYNNN FISCAL-DATE-JULIAN GIVING            <= XYEGPROG
233                    Y2K-FISCAL-DATE-JULIAN                                <= XYEGPROG
234                ADD ROLLDIFF-1-YYNNN WS-FISCAL-DATE-JULIAN GIVING         <= XYEGPROG
235                    Y2K-WS-FISCAL-DATE-JULIAN                             <= XYEGPROG
236         *******IF FISCAL-DATE-JULIAN IS NOT GREATER THAN                  <= XYEGPROG
237         ****************WS-FISCAL-DATE-JULIAN                            <= XYEGPROG
238                IF Y2K-FISCAL-DATE-JULIAN IS NOT GREATER THAN             <= XYEGPROG
239                         Y2K-WS-FISCAL-DATE-JULIAN                        <= XYEGPROG
240                   PERFORM FISCAL-DATE-LESS.                                 XYEGPROG
```

Figure 7. Sample Hot Spot Report

Additional RS/6000's were tied together into a cluster using IBM's LoadLeveler software. LoadLeveler provides batch services for clusters of UNIX based hardware. When a new application was loaded into its own directory, located in a particular location on the file server, the system would automatically detect it and make it available on the analyst interface.

The web interface and clustering software provided a scalable system. Smaller licensees could run all of the software on a single machine, while larger licensees, such as IBM Global Services in Canada, built large factories with a significant number of machines in the cluster.

The performance of the system was reasonable. The Import and Design Recovery phases on an average sized application (about 1,000 files with a total of about 500,000 source lines) took between 8 and 12 hours. Each iteration of Date Analysis typically took under 1 hour, most of which was spent reading in the fact base and writing the results. The actual date type reference tracing took only about 10 minutes. The Hot Spot and Version Integration phases took about the same time as the Import and Design Recovery phases.

Some experiments were made in using conventional database technology to store the design recovered from the source code.

In the later stages of the Y2K timeframe, the tool set was used as a validation tool to check that previously converted systems (manually converted or converted by other Y2K vendors) were compliant. The applications were analyzed as if they were to be transformed, but the Hot Spots Transform pass was not run. The Hot Spot reports then contained all Y2K sensitive locations in the code.

4. Conclusions and Lessons Learned

In this paper we have presented an overview of the use of design recovery techniques to implement a not insignificant design analysis and transformation task, Year 2000 remediation. More than 3.3 billion lines of code were remediated (or validated) using the LS/2000 tool set using fewer than 40 human analysts world wide. Our work is by no means unique. Other techniques have applied design recovery techniques to legacy systems [10,11,12], but few have been proven on this scale of application.

We have focused on the Y2K application of our design recovery techniques since that is the problem that we had the most experience with. The same techniques, with minor modifications were used for other similar maintenance tasks. Some examples of these tasks were the identification of fields of a particular business type (e.g. credit card numbers, employee numbers), dead (unreachable) code analysis, and error analysis (identifying condition statements that lead to error codes). While several hundreds of millions of lines of code were processed solving these maintenance tasks, most individual tasks were not large enough to warrant a transformation stage. Hot sport markup and the generated reports were sufficient for the client to perform manual remediation.

Several lessons were learned from this system. The first is that the manuals for programming languages and compilers do not tell the whole story. Compilers accept variations of languages that are not described in the manuals, and programmers will take advantage of anything the compiler will give them. Early in the LS/2000 life cycle, many applications would break the system as they were processed. Any design recovery of operational legacy systems must be able to adapt to the true semantics provided by the compiler, not just the subset described by the manuals.

A similar lesson is that production code contains errors. Programmers often develop code under pressure and take short cuts. COBOL compilers will ignore an erroneous statement and continue with the next recognized statement. As a result, programs that compile and produce testable results are often left with errors. One case we encountered was the removal of a field from a record. Not all statements that reference that field were removed from the program. Two move statements that assigned values to the field were left in. Since the compiler ignored the statements, and they had no effect on the execution of the program, they were never removed by the programmers. Design Analysis techniques must be prepared to adapt to errors that remain in the application code.

One important lesson we learned, both with LS/2000, and with several other design recovery projects since then, is that design recovery is more effective and efficient when it is task directed. As explained in the section on Date Analysis, in LS/2000 we found that procedure call linkage provided no additional information about the use of dates over and above the CopyID facts. This allowed us to skip generation of parameterization relations in our recovered design factbases, and significantly speeded up our processing. Later projects aimed at other maintenance tasks did require procedure call analysis, but did not benefit as much from the CopyID relations. The set and schema of the design facts required should be based at least in part on the maintenance task at hand.

There are a variety of maintenance tasks that this type of system can solve. The ones we believe to be most promising are based on the migration of technology in legacy systems. Two examples are upgrades of databases (e.g. IMS to DB2), and web enabling of legacy systems.

The hot spot technique is an important contribution of our work. It allows us to tie discoveries made by analyzing the model generated by Design Recovery back to the source code. We have generalized this technique [13], extending its application to other design types and business rules.

Acknowledgements

The authors would like to acknowledge the contributions of the research and development team at Legasys Corp., including Donald Jardine, Russel Halliday, Andy Maloney, Darren Cousineau, Jason Reynolds, Chris Walmsley, Anna Brescher and Brent Nordin.

References

[1] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer*, Vol. 22, No. 7, July 1989, pp. 36–49.

[2] J.R. Cordy, K.A. Schneider. "Architectural Design Recovery Using Source Transformations", *CASE '95 Workshop on Software Architecture*, Toronto, Canada, July 1995.

[3] T.R. Dean, J.R. Cordy, "A Syntactic Theory of Software Architecture", *IEEE Transactions on Software Engineering* Vol. 21, No, 4, April 1995, pp 302–313.

[4] J.R. Cordy, I.H. Carmichael, R.Halliday, *The TXL Programming Language / Version 10*, TXL Software Research Inc., Kingston, Canada, 2000 (http://www.txl.ca).

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

[6] J.R. Cordy, T.R. Dean, A.J. Malton, K.A. Schneider, "Software Engineering by Source Transformation - Experience with TXL", *SCAM'01 – IEEE First International Workshop on Source Code Analysis and Manipulation*, Florence, November 2001, 10 pp.

[7] A. Malton, K. Schneider, J.R. Cordy, T. Dean, D. Cousineau, J. Reynolds, "Processing Software Source Text in Automated Design Recovery and Transformation", *IWPC-2001 – IEEE Ninth International Workshop on Program Comprehension*, Toronto, May 2001, pp. 127-134.

[8] J. W. Hunt and M. D. McIlroy. "An algorithm for differential file comparison." Computing Science TR #41, Bell Laboratories, Murray Hill, N.J., 1975.

[9] E. Myers. "An O(ND) difference algorithm and its variations." *Algorithmica*, Vol. 1, No. 2, 1986, pp. 251–266.

[10] K. Kontogiannis, M. Bernstein, E. Merlo, and R.D. Mori, "The Development of a Partial Design Recovery Environment for Legacy Systems", *Proceedings of CASCON '93*, Toronto, Canada, 1993, pp. 206–216.

[11] K. Sartipi, K.Kontogiannis, F. Mavaddat, "Architecture Design Recovery using Data Mining Techniques", *4th European Conference on Software Maintenance and Reengineering (CMSR 2000)*, March 2000, pp. 129–139.

[12] K. Kontogiannis, S. Tilley, R. Demori, H. Muller, "User-Assisted Design Recovery for Legacy Software Systems", *Workshop on Software Engineering and Artificial Intelligence* at *IEEE International Conference on Software Engineering, (ICSE 16)*, Sorrento, Italy, May 1994.

[13] J.R. Cordy, K. Schneider, T. Dean, A. Malton, "HSML: Design Directed Source Code Hot Spots", *IWPC-2001 – IEEE Ninth International Workshop on Program Comprehension*, Toronto, May 2001, pp. 145–154.