中華民國八十六年全國計算機會議

程式轉換與資料強固維護回顧與展望

Program Transformation and Data Intensive Program Maintenance : A Brief Review and Future Research Agenda

楊宏戟 Hongji Yang*　　朱正忠 William C. Chu　　盧志偉 Chih-Wei Lu

*英國狄蒙佛大學電腦科學研究所
*Computer Science Department
De Montfort University
England
*hjy@dmu.ac.uk

逢甲大學資訊工程研究所
Information Engineering Department
Feng Chia University
Taichung, Taiwan, R.O.C.
Chu@fcu.edu.tw

摘要

本論文首先介紹應用於軟體發展上的程式轉換技術，同時回顧其相關於軟體維護上的技術。尤其，我們將針對資料密集程式的維護上的工作成效、經驗及學習進行總結，而最近更有融入再利用技術的方法問世。最後以未來對於這領域研究的趨勢及建議作結。

Abstract

This paper first introduces program transformation techniques used in software development and then reviews the techniques used for software maintenance. In particular, our work on data intensive program maintenance summarized, together with experiences and lessons learnt. More recently, a method for incorporating reuse has been added, and experience with this is described. Conclusion and recommendations for future research in the field are made.

1 Introduction

1.1 Software Maintenance and Reverse Engineering

In the early days of computing (1950s and early 1960s), software maintenance comprised a very small part of the software lifecycle. In the late 1960s and the 1970s, as more and more software was produced, people began to realize that old software does not simply die, and at that point software maintenance started to be recognized as a major activity. By the late 1970s, industry was suffering major problems with the applications backlog, and software maintenance was now taking more effort than initial development in some sectors. In the 1980s, it was becoming evident that old architectures were severely constraining new design [9]. All of these were placing demands that the changes to the software were performed. Changes include, for instance, fixing errors, adding enhancements and making optimizations. Besides the problems whose solutions required the changes in the first place, the implementation of the changes themselves create additional problems. One of the five Lehman's laws of the evolution of a software system directly addresses the modification of software. It states that "a program that is used in a real world environment must change or become less and less useful in that environment" [34]. So mechanisms must be developed for evaluating, controlling, and making changes.

Software Maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [1]. So maintenance activities can be divided into these categories correspondingly [42]: *corrective maintenance*, to remove faults, which do not conform to the specification, in the software; *adaptive maintenance*, to adapt the change to the external environments; *perfective maintenance*, to be undertaken as a consequence of a change in user requirements of the software; and preventive maintenance, to be undertaken on a system in order to anticipate future problems and make subsequent maintenance easier [8].

The large cost associated with software maintenance is the result of the fact that software has proved difficult to maintain. Early systems tended to be unstructured and ad hoc. This makes it hard to understand their underlying logic. System documentation is often incomplete, or out of date. With current methods it is often difficult to retest or verify a system after a change has been made. Successful software will inevitably evolve, but the process of evolution will lead to degraded structure and increasing complexity [11,25,34].

Software maintenance has its own life cycle and its own features. Over the years, several software task models have been proposed, while the model by Bennett [9] is used here:

1. *request control*: the information about the request is collected; the change is analyzed using impact analysis to assess_cost/benefit; and a priority is assigned to each request.
2. *change control:* the next request is taken from the top of the priority list; the problem is reproduced (if there is one); the code (and design and the specifications if available) are anal- the changes are

designed and documented and tests produced; the code modifications are written; and quality assurance is implemented.

3. *release control*: the new release is determined; the release is built; confidence testing is undertaken; the release is distributed; and acceptance testing by the customer takes place.

Currently, these three steps are almost always undertaken in terms of source code. Design information and even adequate documentation often do not exist. Thus software maintenance is thought of predominantly as a source code activity. Understanding the functions and behaviors of a system from the code is hence a vital part of the maintenance programmer's task [39]. Approaches to program comprehension will be described in later chapters.

Reverse Engineering involves the identification or "recovery" of program requirements and/or design specifications that can aid in understanding and modifying the program. The main objective is to discover the underlying features of a software system including requirements, specification, design and implementation. In other words, it is to recover and record high level information about the system including:

- the system structure in terms of its components and their interrelationships expressed by interface,
- functionality in terms of what operations are performed on what components,
- the dynamic behavior of the system in understanding how input is transformed to output,
- rationale - design involves deciding between a number of alternatives at each design step,
- construction - modules, documentation, test suites etc.

There are several purposes for undertaking reverse engineering listed in [9]. They can be separated into the quality issues (e.g., to simplify complex software, to improve the quality of software which contains errors, to remove side effects from software, etc.), management issues (e.g., to enforce a programming standard, to enable better software maintenance management techniques, etc.) and technical issues (e.g., to allow major changes in a software to be implemented, to discover and record the design of the system, and to discover and represent the underlying business model implicit in the software, etc.). In most cases, reverse engineering is the first step of software maintenance. The analysis of the object software is crucially important to accomplish the request control stage of software maintenance.

1.2 Program Refinement and Transformation

By the term *refinement*, we mean a technique to produce correct implementations from specifications [35,33]. From this, we know that specification and implementation are two essential elements in the refinement process.

Refinement can be carried out informally or formally. Figure 1 presented a general picture of formal program development in which programs were evolved from specifications in a gradual fashion via a series of refinement steps. Probably the most useful potential application of formal specifications is to the formal development of programs by gradual refinement from a high-level specification to a low-level "program" or "executable specification" [26,30,40]. Actually, some refinement steps are more or less routine. Such refinement steps can typically be described schematically as transformational rules. The process of changing a program (specification) to a different program (specification) with the same semantics as the original program (specification) is called *program transformation*.

Any refinement obtained by instantiating a transformation rule will be correct. Rather than proving correctness separately for each instantiation, the rule itself can be proved correct and then applied as desired without further proof. This led to a method of program construction - *transformational programming*, i.e., to construct program by successive application of transformation rules. Usually this process starts with a formal specification and ends with an executable program.

Much recent work has been focused on the program transformation as one kind of programming paradigm in which the development from specification to implementation is a formal, mechanically supported process. Research on program transformation aims at developing appropriate formalisms and notations, building computer-based systems for handling the bookkeeping involved in applying transformation rules, compiling libraries of useful transformation rules, developing strategies for conducting the transformation process automatically or semi-automatically. The long-range objective of this paradigm is dramatically to improve the construction, reliability, and maintainability of software.

2 Program Transformation Systems

2.1 Program Transformation for Software Development

Let us start with the software system development. The most widely used method is to derive the final program from a specification. We use SP to represent a specification of requirement which the software system is expected to fulfill; and P to represent the ultimate object program which satisfies the specification in SP.

The usual way to proceed is to construct P by whatever means are available, making informal reference to SP in the progress, and then verify in some way that P does indeed satisfy SP. The only practical verification method available at present is to test P. checking that in certain selected cases that the input/output relation it computes satisfies the constraints imposed by SP. This has the obvious disadvantage that (except for trivial programs) correctness of P is never guaranteed by this process, even if the correct output is produced in all test cases. An alternative to testing is a formal proof that the program P is correct with respect to specification SP.

Most work in this area has focused on methods for developing programs from specification in such a way that the resulting program is guaranteed to be correct by construction. The main idea is to develop P from SP via a series of small refinement steps, inspired by the programming discipline of stepwise refinement [33]. Each refinement step captures a single design decision, for instance a choice between several algorithms which implement the same function or between several ways of efficiently representing a given data type. This yields the following diagram (Figure 1) (SP_0 represents the initial specification; those steps in between SP_0 and P are represented by SP_1, SP_2 and etc.).

$$SP_0 \rightarrow SP_1 \rightarrow SP_2 \rightarrow \ldots \rightarrow P$$

Figure 1: Stages of Program Development

If each individual refinement step ($SP_0 \rightarrow SP_1$, $SP_1 \rightarrow SP_2$ and so on) can be proved correct. the P itself is guaranteed to be correct. Each of these proofs is orders of magnitude easier than a proof that P itself is correct since each refinement step is small.

To compare the various transformation systems for software development, and to judge whether a transformation system is good eventually depends on the extent to which it can fulfill the goal - transforming a specification to a running program. However, it is not the only purpose of this review, and a more important aspect is to learn what can be used in undertaking software maintenance.

There exist a number of transformation systems which have been described:

- **Optimizing Compilers** — program transformation techniques have been used for many years in optimizing compilers, because inefficient programs can be transformed into efficient programs (e.g., loop induction, strength reduction, expression reordering, symbolic evaluation, constant propagation, loop jamming).

- **Burstall and Darlington's Work** — the work on program transformation by Burstall and Darlington was done in the mid-1970's [16,38]. Their system was based on schema-driven method for transforming applicative recursive program into imperative ones with improving efficiency as the ultimate goal. The system worked largely automatically, according to a set of built-in rules with only a small amount of user control.

- **Balzer's Work** — built an implementation system for program transformation [4,5,6]. This system was designed mechanically to transform formal program specifications into efficient implementations under interactive user control. He expressed the problem by a formal specification language GIST, which was operational (i.e., having an executable semantics).

- **ZAP** — Feather's ZAP system [24] is based on the Burstall/Darlington system with a special emphasis on software development by supporting large-program transformation. The input/target language of the system is NPL (an applicative language for first-order recursion equations). The system provides the user with a means for expressing guidance. An overall transformation strategy is hand-expanded by the user into a set of transformation tactics such as combining, tupling generalization.

- **DEDALUS System** — The DEDALUS system (DEDuctive Algorithm Ur-Synthesiser) by Manna and Waldinger was implemented in QLISP [36]. Its goal was to derive LISP programs automatically and deductively from high-level input-output specifications in a LISP-like representation of mathematical-logical notation. The system incorporates an automatic theorem prover and includes a number of strategies designed to direct it away from rule applications unlikely to lead success.

- **CIP-S** — CIP-S is the approach of the Project CIP (computer-aided, intuition-guided programming) [7], which is to develop along the idea of transformational programming within an integrated environment, including methodology, language, and system for the construction of "correct" software. A prototype system was built. The system is interactive and the development process is guided by the programmer who has to choose appropriate transformation rules. The system is language-independent and is based on the algebraic view of language definition; any algebraically defined language is suited for manipulation, provided respective facilities for translating between external and internal representations are available.

To summarize, there is widespread demand for safe, verified, and reliable software. This demand arises from economic considerations, ethical reasons, safety requirements, and strategic demands. Transformational programming can clearly make a valuable contribution toward this goal. It already covers several phases of the classic software engineering lifecycle and shows promise of covering the remaining ones. But, after near twenty

year's research, existing transformation systems are still experimental and the problems they are capable of coping with are still more or less toy problems. To make practical use of transformation systems is no doubt the key problem to be solved in transformational programming.

2.2 Program Transformation for Software Maintenance

Several recent transformation systems (formal and informal) have been described:

- **Reverse Engineering in REDO** — REDO (Restructuring, Maintenance, Validation, and Documentation of Software Systems) is an ESPRIT II project, which ran from 1989-93 and is concerned with "rejuvenating" existing applications into more maintainable forms by improving documentation, by restructuring code, and by validating the code against the original intentions. As one part of the REDO project, reverse engineering (reverse-engineering COBOL programs into Z specifications) was carried out at Oxford University [14,15,31,32]. The strategy here is to perform abstraction first, and then perform transformation on the high level language. The method looks promising, but has not been investigated in depth on industrial-scale code.

- **Sneed's Work** — Sneed and Jandrasics use automated tools to support the retranslation of software code in COBOL back into an application specification by the process of reverse engineering [41]. Two steps are needed, to recover a program design from the source code and to recover a program specification from the program design. A set of transformation rules for mapping COBOL source code back into the design schema is obtained by inverting those rules used to generate COBOL programs from the design. The programs are modularized and restructured as a by-product of the reverse transformation process. Sneed's work has been thoroughly described in a number of papers, and is one of the very few commercially successful reverse engineering methods.

- **A CASE Tool for Reverse Engineering** — Bachman introduced a CASE tool, DOCMAN, for reverse engineering COBOL programs [3]. The Re-Engineering Cycle chart provides an architectural view of this CASE tool, which features both forward and reverse engineering. Particularly, reverse engineering begins at the bottom left with the definition of existing applications and raises the applications to successively higher levels of abstraction. At the top, the design objects created by the reverse engineering steps are enhanced and validated to become the revised design objects used in the forward engineering process. At the bottom, a new applications system becomes an existing applications system at the moment that it goes into production. Basically, this is an informal approach.

- **TMM** — A method was proposed in [2] for recovering abstractions and design decisions that were made during implementation. This method is called Transformation-based Maintenance Model (TMM). The purpose of this system is to reimplement a system in order to adapt it to a new environment through reuse. The abstractions and design decisions of software must be recovered first before the software is reimplemented. The recovery work in TMM paradigm is done by maintenance by abstraction (MBA).

- **A Concept Recognition-Based Program Transformation System** — This is an approach that applies a transformation paradigm to automate software maintenance activities [22]. The characteristic of this approach is its use of concept recognition, the understanding and abstraction of high-level programming and domain entities in programs as the basis for transformations. Four understanding levels are defined: the text level, the syntactic level the semantic level, and the concept level. The program transformation system depends on its program understanding capabilities up to the concept level. The key component is a concept library which contains the knowledge about programming and application domain concepts, and the knowledge about how these concepts are to be transformed. Concept recognition is done by pattern matching.

- **REFORM** — REFORM (Reverse Engineering using FORmal Methods) is a joint project between University of Durham, CSM Ltd. and IBM (UK) to develop a tool called the Maintainer's Assistant. The main objective of the tool is to develop a formal specification from old code. It will also reduce the costs of maintenance by the application of new technology and increase quality so producing improved customer satisfaction. The old code in this project is the IBM CICS. The aims of the Maintainer's Assistant are to provide a tool to assist the human maintainer, handling assembler and Z in an easy to use way [10,18,13].

To summarize, most of these approaches have been advocated for reverse engineering, but few have been evaluated in practice on large-scale code. Results are only available in depth for Sneed's method, and for ReForm.

From the above systems, we know that a great effort is still needed to put the paradigm of reverse engineering into practical use. It is a hard job to reverse an existing program back to its design or specification. For instance, one of the problems is that the availability and accuracy of the design information are both assumed. Typically, such information is obsolete or lacking in systems which

have gone through years of maintenance. For such systems, source code is the only reliable source of information. Another problem is that there is not a method for coping crossing levels of abstraction covering all abstraction levels in these systems.

The state of the art in reverse engineering may be summarized as follows. Most existing commercial tools are basically restructurers, and these operate at the same level of abstraction. Even module recovery tools, such as those in Sneed's work, operate at the syntactical level, e.g., grouping variables and operations on them. Where genuine crossing of levels of abstraction occurs, this is done manually, e.g., in Sneed's system for COBOL, or in redocumentation systems such as DOCMAN [3].

The first step for conducting software maintenance is to understand the software to be maintained and the abstraction of the program design and specification from the existing source code is one of the methods which helps us to understand software systems. Reverse engineering is particularly important when the source code is the only source with which to work. Reverse engineering data intensive programs based on a transformation approach for the purpose of software maintenance and aims to find out the formal relation between program code and its design and eventually specification surely crucial for software maintenance.

3 Summary of REFORM and Data Intensive Programs

3.1 Failures of REFORM

The prototype of the Maintainer's Assistant in the REFORM Project started with tackling computation intensive program. Before the work on data intensive program maintenance began, the following points were noticed:

- Almost all program transformations in the transformation library based on Ward's work were mainly for dealing with functional abstraction (or control abstraction) — most transformations operated on control structures of a program while few transformations on data structures. In another words, the system was only suitable to operate on computation-intensive programs, not data-intensive programs. The program transformer can only deal with the construction of well-structured code.

- To obtain a specification expressed in Z is a long-term goal for the REFORM project. Most of the program transformations can only be used for restructuring programs at the code level, i.e., both programs before and after the transformation being applied are in the same abstraction level.

- Most of the program transformations that currently are implemented can only be used for restructuring programs at relatively low levels of abstraction.

- No representations of types, complex data structures and data design yet exist in WSL.

- A new application area of the tool was identified as acquiring data design from data-intensive programs written in e.g. COBOL. After seeing the demonstration of the prototype of the Maintainer's Assistant, many industrialists were disappointed with the tool for being unable to deal with COBOL programs though they confirmed the potential capability of the Maintainer's Assistant.

These facts urged a new research direction to be set up i.e., employing program transformation technique emphasizing data abstraction and to end up with data designs.

3.2 Problems with of Data Intensive Programs

Data-intensive programs and computation-intensive programs are comparative notions. There is no clear distinction between these two sorts of programs. *Data-intensive programs* mean programs which are written in data-intensive programming languages that provide complex data structuring mechanisms and high-level composite operations to manipulate them. *Computational-intensive programs* mean programs which are written in computational-intensive languages that provide ways to express computations using relatively simple operations on elementary objects [28]. COBOL is a typical data-intensive programming language.

The COBOL language used in this research not only is unrestricted to any dialect of COBOL but also covers features written in ANSI COBOL Standard 1985. More importantly, this research will be not only of benefit to COBOL programs but also to other data intensive programs written in other languages. Programs written in COBOL have characteristics which are different to those of typical computation-intensive programs. and these are important constraints in reverse engineering such systems, e.g.:

- Important data is represented in the form of records and operations on data are therefore heavily record based.
- COBOL programs are often designed using Entity-Relationship Attribute Diagrams, rather than process based design methods.
- COBOL allows the programmer to specify that two different records (with different structures) may share the same memory location. This is known as

the aliasing problem and is found in many COBOL programs.
- COBOL programs usually have external calls to the operating system and database management system.
- COBOL programs may use many foreign keys to represent complex data structures which in other languages would use pointers.

4 *Recent Research Results*

One of the characteristics of data intensive third generation languages is that high level data designs often translate at the implementation level to constructs in both the code and data. For example, a reference in the data design between two data structures is typically implemented in COBOL by a foreign key, i.e., an integer index from one to the other. The relation between the two data structures can only be discovered by examination of the data and the code, not the data alone. Existing reverse engineering techniques have difficulty handling this. It seemed to us that formal transformation offered potential to solve this problem.

It is considered that the approach using program transformations is also a suitable method for acquiring data designs, because performing data abstraction operations also needs the properties of program transformations, such as the preservation of semantics and suitability for tools, etc.

WSL currently has declarations which introduce the name of an identifier without its type. Therefore, variables are not typed, but all values in WSL have a type which belongs to a distinct set of values. Since COBOL treats all significant data as records, defining "records" in WSL for modeling COBOL records is a clear requirement. The external calls to the underlining operating system and the embedded database can be modeled as external procedure calls and external functions. WSL already has mechanisms for dealing with external calls. The foreign key problem can be dealt with by program transformations. These transformations analyze the code with foreign keys and relations between modules using foreign keys could be found.

ERA (Entity-Relationship Attribute) Diagrams are based on entity models [17,20,21,37]. Entity models provide a system view of the data structures and data relationships within the system. Entity-Relationship Attribute Diagrams are suitable forms for representing data designs for data-intensive programs and therefore WSL needed to be extended to include Entity-Relationship Attribute Diagrams.

4.1 A Method for Design Recovery and Reuse

A new method for design recovery and identifying reusable components through program understanding was proposed.

1. Translating a COBOL program into WSL.
2. Using initial tidy-up transformations to "clean up" the target program in WSL in order to reduce the redundant statements introduced during the translation.
3. Looking for functionally self-contained modules. A code module, a function or a procedure in the original software system, are potentially self-contained modules. A reusable component may well be obtained from one of the above modules. A module which is not a function or a procedure may also transformed into an abstract data type, and hence also a candidate of a reusable component.
4. Taking one module obtained from the above process to work, one at a time. Program transformations are applied to the module to reverse the module into its high-level representation in ERA diagrams. These ERA diagrams are the main results of the reverse engineering process.
5. The obtained ERA diagrams are viewed as reusable components. The ERA diagrams, together with the original code, are used by a Semantic Interface Analysis tool to generate semantic predicates and interface predicates for a reusable module in terms of its preconditions, post-conditions and obligations. These predicates are used to serve as the rules of describing implicit semantics, characteristics, and interface requirements of each software component explicitly.
6. Storing a reusable module in the Reuse Library, and maintaining a formal link between the reusable module and its high abstraction level representation.

In implementing the above method, MA has to be extended (Figure 2), which mainly includes development of transformations for dealing with data intensive programs and development of the Semantic Interface Analyzer.

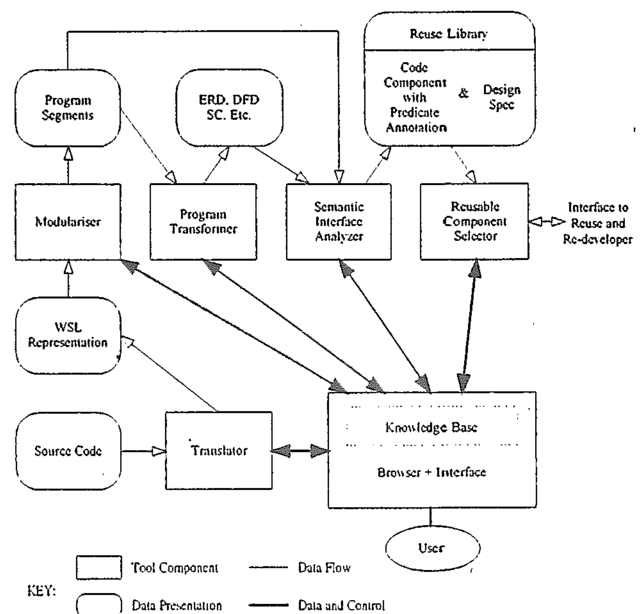

Figure 2: Extended MA

4.2 Development of Transformations

Transformations developed for extracting Entity-Relationship models in our tool are divided into seven categories [18]:

1. **Basic Data Structures and Entity Relationship Components**: Transformations in this category deal with simplifying data objects in basic data types according to the properties of data type, i.e. (1) from record to entity (2) acquiring a relationship from a record with sub-records.

2. **Transformations for Manipulating Data Items**: Transformation in this category deal with manipulating data items for the preparation of applying further transformations, for example, to move a record to a position closer to another record so permitting them to be joined to form an entity.

3. **Files**: Though COBOL file operations can be translated into WSL as external procedures and external functions (i.e., we effectively ignore them, and knowledge of variable usage is lost across calls), more suitable forms of data representations are required to replace these external procedures and functions in order to examine file operations at a high abstraction level. In our tool, a queue data type is proposed to model COBOL sequential files and operations on these files, in order that files (external storage objects) can be transformed into queues (internal mathematical objects). We have not yet addressed random access files, but would model them with arrays.

4. **Aliases**: We first determine which records are aliased and determine a mapping between the aliased records. We then define a function to describe a WRITE to an aliased record as mapping the COBOL data structure to a low level memory model and a function to describe a READ from an aliased record as a mapping in the reverse direction.

5. **Foreign Keys**: A relationship can exist between two entities that both have the same attribute (known as a foreign key) and the relationship can be spotted by transformations in the imperative code. This relationship can be abstracted from two entities which have been derived already from source code (e.g. record definitions) and two relations which between two pairs of entity attributes (e.g. assignment statements).

6. **Abstract Data Types** : Transformations in this category deal with recognizing an abstract data type from constituent data declarations and operations on them. An abstract data type consists of "objects" and "operations". Objects are usually implemented as variables and operations are implemented as procedures and functions. In reverse engineering, an abstract data type may be formed by looking for a closure of a group of variables and a group of procedures (or functions).

7. **Functional Relationships**: Transformations in this class address how ERA models are extracted from code, in particular, from assignment statements, branching structures and loop statements.

4.3 Case Studies

A number of case studies on COBOL code (including real programs of several thousand lines) were used in experiments [23].

- Some of the "data" statements which are not able to contribute to the eventual ERA diagrams are omitted. For example, COBOL statements, such as MOVE 0 TO A-VARIABLE, MOVE SPACES TO A-VARIABLE, INSPECT statements, COMPUTE statements, etc. were omitted. This represents information not needed in abstraction.
- Almost all the assignment statements that were originally translated from the MOVE statements in COBOL were abstracted to *relate* statement. So were those assignment statements originally from ADD and SUBTRACT statements. A *comment* statement was usually added along with each abstraction in order to record information which will be used to decided the degree of the relationship between the two entities which would be obtain from the two records linked by the *relate* statement later on.
- The REDEFINES statements in the original COBOL program were translated into *redefine* statements in WSL. According to observation, all original records to be redefined were of the same data types as that of the redefining records. The conclusion how applying those functions for dealing with aliased records was that aliased records would not affect each other in the abstraction process. Therefore, a record and its redefining record were treated as independent records.
- Entities were abstracted from records and this is the starting point of moving from the code level to the conceptual level. This was done when restructuring work at the code level had been finished.
- Relationships mainly derived from the *relate* statements and information recorded by the *comment* statements were used to decide the degrees for the relationships by transformations with a help of human expertise.
- Duplicate entity relationships which might be obtained from different places of the program, and these were checked and removed. All the *comment* statements were removed when the abstraction process was finished.
- Data designs obtained from programs made these programs much more comprehensible.

The method developed in this research can deal with most code in the case studies. In particular, COBOL records and files are able to be represented in WSL and this is crucial to the implementation of the prototype as well as the successful application of the method. Therefore, it is comparatively easier to extract ERA models from a relatively independent (self-contained) segment of COBOL code with record (file) definitions, but it is more difficult for a COBOL segment with many calls to other segments (i.e. with many PERFORM statements) because the structural complexity is increasing. The problem may be helped by building more powerful "restructuring" transformations, which is not a main thrust of the paper.

4.4 Reuse of COBOL Code and Design

Semantic interface analysis is a formal approach where semantic attributes of software components were described by formal notations. Software reuse includes areas of concern such as representation, retrieval, and adaptation and integration [12]. Our work, at this stage, is focussing on representation and retrieval, i.e., firstly to identify reusable components and to store these reusable components in a reuse library, which contain formal semantic interface specifications consisting of precondition, postcondition, and obligation predicates represented as specialized comments. An existing reuse library system will provide the initial retrieval mechanism for the selection of candidate reuse components. A candidate reuse component is then inserted into the application system. The application system consist of both newly developed components, and previously adapted reuse components all of which contain formal semantic predicates.

There are two basic technical approaches to reuse: parts-based and formal language-based [27], the parts-based approach assumes a human programmer integrating software parts into an application by hand. In the formal language-based approach, domain knowledge is encoded into an application generator or a programming language. Our study on COBOL code reuse focus on the parts-based approach. In parts-based approach, components are required to be found and understood, and then incorporated into the designed system. Reusable parts are identified through reverse engineering via program transformation. Program understanding is done inside the program transformation process. Data structures are the main points to reverse COBOL programs because they are written in a data intensive program language (COBOL).

4.5 An Example

The example program used in this illustration was taken from a COBOL text book [29] and its COBOL source code is shown in Figure 3.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  THIS PROGRAM SEQUENTIALLY ACCESSES TO TWO SEQUENTIAL    *
*  FILES, ONE IN INPUT MODE AND ONE IN OUTPUT MODE.         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COPY-CUSTOMER-LIST.

 ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
          SELECT CUSTOMER-LIST ASSIGN TO XYZ
              ORGANIZATION SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.
          SELECT CUSTOMER-LIST-BACK ASSIGN TO WXY
              ORGANIZATION SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.

 DATA DIVISION.
     FILE SECTION.
     FD CUSTOMER-LIST.
     01 CUSTOMER-RECORD.
        02 NAME                     PIC X(20).
        02 ADDRESS                  PIC X(50).
        02 PHONENUM                 PIC X(20).

     FD CUSTOMER-LIST-BACKUP.
     01 BACKUP-RECORD.
        02 B-NAME                   PIC X(20).
        02 B-ADDRESS                PIC X(50).
        02 B-PHONENUM               PIC X(20).

     WORKING-STORAGE SECTION.
     01 EOF PIC X.

 PROCEDURE DIVISION.
 MAIN.
     OPEN INPUT CUSTOMER-LIST
          OUTPUT CUSTOMER-LIST-BACKUP
     PERFORM, WITH TEST AFTER, UNTIL EOF = "T"
          READ CUSTOMER-LIST NEXT;
          AT END
              MOVE * TO EOF
          NOT AT END
              MOVE "F" TO EOF
              MOVE CUSTOMER-RECORD TO BACKUP-RECORD
              WRITE BACKUP-RECORD;
     END-PERFORM
* THE STOP RUN STATEMENT CLOSES THE FILES
     STOP RUN.
```

Figure 3: A File-backup Program in COBOL

This program is translated into its equivalent form in WSL (Figure 4). The program module was a procedure in the original program and it would be called by a (COBOL) PERFORM statement. This program copies the contents in one file to another file. Figure 4 shows the format of the program when loaded in to the prototype transformation tool. The program in Figure 5 is then dealt with by the Program Transformer, which applies transformations to it. The final result of the "file-backup" program can be shown by an ERA diagram (Figure 6).

When the original program was transformed into an ERA diagram, the user can easily decide that the program segment can be a good candidate of a reusable component. Therefore, the component in WSL (Figure 3 and 4) will also be analyzed by the Semantic Interface Analyzer (SIA) in order to generate a form annotated with formal pre-conditions, post-conditions and obligations. For details of annotation rules of SIA, please refer to [19].

Based on the predefined software templates for COBOL in the Knowledge Base, predicate analysis and propagation, we can infer that the module file-backup contains the predicates as shown in Figure 7.

```
comment: "program-id: file-backup";
file source-file-name with
    record source-record end;
```

```
end;
file target-file-name with
    record target-record end;
end;
eof:=0;
!p open_file (i var source-file-name);
!p open_file (o var target-file-name);
while ( eof ≠ 1 )
  do
  if non_empty? (!f eof? (source-file-name))
    then eof := 1
    else eof := 0;
      !p read_file (source-record var source-file-name);
      target-record -= source-record;
      !p write_file (target-record var target-file-name); fi;
  od;
```

Figure 4: The File-backup Program in WSL

```
entity source-record end;
entity backup-record end
relationship entity target-record has one back-up relation
        with one entity source-record;
```

Figure 5: An Entity Relationship Diagram for the File-backup Program in WSL

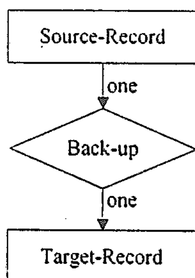

Figure 6: Entity Relationship Diagram for "File-Backup" Program

```
pre-condition
!Sequential-file( File1, record-type1 )
!Sequential-file( File2, record-type2 )
!EQU( record-type1, record-type2 )

post-condition
!EQU( File1, File2 )

file-backup( File1, File2 )
```

Figure 7: Semantic Interface Predicates Generated

5 What still Needs Doing

This paper has reviewed program transformation techniques for both software development and software maintenance, focussing in particular on our work in data intensive program maintenance. From our experiences in acquiring data designs from data intensive programs for software maintenance using program transformation, it is possible to identify weaknesses as well as strengths.

There is an "informal" stage in our approach, i.e. the translation from COBOL to WSL is currently done manually. For the time being, building a translator is considered as a disadvantage because it needs to consider all possible syntax combinations of the COBOL language while hand translation can tailor any program as long as the program is semantically translated into an equivalent form in WSL. Such a translator needs to be built.

More experiments should also be carried out with data intensive programs in other programming languages. This includes not only building translators for translating programs in other languages but also studying the features of programs in those languages. At present, the prototype is mainly COBOL-specific and it can only deal with the sort of data structures that exist in COBOL. However, the method developed in this paper is general enough to cope with data-intensive programs in other languages. Experiments carried out so far have only been focussed on capturing reusable data intensive program components. Future work needs to be conducted in adapting and integrating these reusable components into a new system to being developed or being re-engineered.

Another area for future research is to find out whether the approach developed in this paper can be used to acquire specifications (e.g., specification written in Z) from programs, which was the original aim of the REFORM project and remains an open question.

Combining program transformations with object orientation will also be an very important issue. Initial thoughts about introducing an object-oriented approach into our system include extending the existing kernel and re-building a WSL kernel language in order to develop an object-oriented WSL, developing transformations under the object-oriented approach for reverse engineering data intensive programs, developing a new method or extending the existing method for data intensive program reuse, etc.

Measures for both code and designs/specifications need to be defined. It is perhaps important that a metric can reflect the process of crossing levels of abstraction.

This research has so far indicated that the approach of program transformation can be used to acquire data designs from data intensive programs. However, the real application of this approach will not be seen until an industrial-strength tool has heed built. Therefore, more research should be conducted to improve the prototype developed in this paper into a practical tool.

6 Concluding Remarks

We have reviewed briefly the approach using program transformations as a tool for software development and software maintenance. We have also giving an example of investigation and feasibility study of the problems concerning data intensive program maintenance and reuse using a program transformation approach.

Program transformations are a powerful tool in reverse engineering existing data intensive programs and providing a facility for reuse of these programs. Our approach of dealing with data-intensive programs is to derive a program data design from a data intensive program through program transformations, to represent designs in Entity-Relationship Attribute Diagrams and to annotate reusable components with pre-conditions, post-

conditions and obligations through Semantic Interface Analysis techniques.

The development of the method and the implementation of the prototype show that this approach has covered a scope from theory to practice. Other systems which can derive data designs represented in ERA Diagrams have not been described. The abstracted ERA Diagrams are able to represent the designs of the original programs. The correctness of the obtained ERA diagrams is at present checked manually based on human knowledge and expertise.

Finally, research work on dealing with data intensive programs through program transformations remains a fruitful area, because it offers much potential for solving programs of major importance in industry. Further research into program transformation techniques may useful be extended to incorporate other techniques, such as Object Oriented technology

References

[1] ANSI, Standard 729, IEEE Standard Glossary of Software Engineering Terminology, 1983.

[2] Arango, G., Baxter, I., Freeman, P. and Pidgeon, C., Software Maintenance by Transformation", IEEE Software, May, 1986.

[3] Bachman, R., A CASE for Reverse Engineering, Cahners Publishing Company, July, 1988, reprinted from DATAMATION.

[4] Balzer, R., "Transformational Implementation: An Example", IEEE Transactions on Software Engineering, Vol. SE-7, No. 1, pp. 3-14 (January 1981)

[5] Balzer, R., "A 15 Year Prospective on Automatic Programming", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp. 1257-1267 (November 1985).

[6] Balzer, R, Goldman, N. and Wile, D, "On the Transformational Implementation Approach to Programming", The 2nd International Conference on Software Engineering, San Francisco, California, 1976.

[7] Bauer, F L, Moller, B B, Partsch, H and Pepper, P, "Formal Program Construction by Transformation - Computer-Aided, Intuition-Guided Programming", IEEE Transactions on Software Engineering, Vol. 15, No. 2, pp. 165-180 (February 1989)

[8] Bennett, K. H., "The Software Maintenance of Large Software Systems: Management Method and Tools", Technical Report, Durham University, 1989

[9] Bennett, K. H, "An Overview of Maintenance and Reverse Engineering", in The REDO Compendium, John Wiley & Sons, Inc., Chichester, 1993

[10] Bennett, K. H., Bull, T and Yang, H., "A Transformation System for Maintenance - Turning Theory into Practice", IEEE Conference on Software Maintenance-1992, Orlando, Florida, November, 1992.

[11] Bennett, K. H, Denier, J and Estublier, J., "Environments for Software Maintenance", Technical Report, Durham University, 1989

[12] Biggerstaff, T J and Ritcher, C., "Reliability Framework, Assessment and Direction", IEEE SOftware, Vol. 14, No. 4, pp. 252-257 (1987).

[13] Yang, H. and Bennett, K. H., "Extension of A Transformation System for Maintenance - Dealing With Data-Intensive Programs", IEEE International Conference on Software Maintenance (ICSM '94), Victoria, Canada, September, 1994.

[14] Breuer, P., "Inverse Engineering: The First Step Backwards", Technical Report (ESPRIT Project: 2487-TN-PRG-1031), Programming Research Group, Oxford University, 1990.

[15] Breuer, P., "Tackling Reverse Engineering", Technical Report (ESPRIT Project: 2487-TNPRG-1037), Programming Research Group, Oxford University, 1990.

[16] Burstall, R. M. and Darlington, J. A., "A Transformation System for Developing Recursive Programs", Journal of the ACM, Vol. 24, pp 44-67 (1977).

[17] Chen, P. P., "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transaction on Database Systems, Vol. 1, No. 1 (March 1976).

[18] Yang, H. and Bennett, K. H., "Aquairing Entity-Relationship Attribute Diagrams from Code and Data through Program Transformation", IEEE International Conference on Software Maintenance (ICSM '95), Nice, France, October, 1995.

[19] Chu, W. C. and Yang, H., "Component Reuse Through Reverse Engineering and Semantic Interface Analysis", Accepted by The 19th IEEE Annual Computer Software Application Conference (CompSac '95), Dallas, Texas, August, 1995.

[20] Cutts, G, Structured Systems Analysis and Design Methodology, Paradigm Publishing Company, London, 1987.

[21] Date, C. J., An Introduction to Database Systems, Vol I, Addison-Wesley Publishing Company, Manchester, 1986.

[22] Engberts, A., Kozaczynski, W. and Ning, J., "Concept Recognition-Based Program Transformation", IEEE Conference on Software Maintenance-1991, Sorrento, Italy, 1991.

[23] Yang, H, "Acquiring Data Designs from Existing Data Intensive Programs", Ph.D. Thesis, Durham University, 1994.

[24] Feather, M S., "A System for Assisting Program Transformation", ACM Transactions on Programming Language Systems, January, 1982.

[25] Federal Information Processing Standards, "Guidelines on Software Maintenance", U.S. Department Commerce/National Bureau of Standards, Standard FIPS PUB 106, June, 1984.

[26] Fickas, S. F., "Automating the Transformational Development of Software", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11 (November 1985).

[27] Frakes, W. B. and Pole, P. T., "An Empirical Study of Representation Method for Reusable Software Components", IEEE Transactions on Software Engineering, Vol. SE-20, No. 8, pp. 617-630 (August 1994).

[28] Ghezzi, C., "Modern Non-Conventional Programming Language Concepts", in Software Engineer's Reference Book, Butterworth Heinemann, 1991, pp. 44/1-44/16.

[29] Inglis J., COBOL 85 for Programmers, John Wiley & Sons, Inc., Chichester, 1989.

[30] Kant, E., "Efficient Synthesis of Efficient Programs", in Artificial Intelligence and Software Engineering, 1986, pp. 157-188.

[31] Lano, K. and Breuer, P. T., "From Programs to Z Specifications", Technical Report, Oxford University, 1990.

[32] Lano, K. and Haughton, H., "Applying Formal Methods to Maintenance", Technical Report (ESPRIT Project: 2487-TN-PRG-1042), Programming Research Group, Oxford University, 1990.

[33] Wirth, N., "Program Development by Stepwise Refinement", CACM, Vol. 14, No. 4 (1971).

[34] Lehman, M. M., "Programs, Life Cycles, and Laws of Software Evolution", Proc IEEE, Vol. 68, No. 9 (1980).

[35] Whysall, P., "Refinement", in Software Engineer's Reference Book, Butterworth Heinemann, 1991.

[36] Manna, Z. and Waldinger, -"A Deductive Approach to Program Synthesis" ACM Transactions on Programming Language Systems, February, 1980.

[37] Martin, J. and McClure, C., Structured Techniques for Computing, Prentice-Hall International,
Inc., Englewood Cliffs, New Jersey, 1985.

[38] Partsch, H. and Steinbruggen, R., "Program Transformation Systems", Computing Surveys, Vol 15, No. 3 pp. 198-236 (September 1983).

[39] Robson, D. J., Bennett, K. H., Cornelius, B. J. and Munro, M., "Approaches to Program Comprehension", Journal of Systems Software, 1991.

[40] Sannella, D. and Tarlecki, A., "Toward Formal Development of Programs from Algebraic Specification: Implementation Revisited", ACTA Informatica, 1988.

[41] Sneed, H. M. and Jandrasics, G., "Inverse Transformation of Software from Code to Specification", IEEE Conference on Software Maintenance-1988, Phoenix, Arizona, 1988.

[42] Swanson, E. B., "The Dimension of Maintenance", Second International Conference on Software Engineering, Los Alamitos, California, 1976.