ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

Reverse Engineering and Reusing COBOL Programs: A Program Transformation Approach

Hongji Yang and Yong Sun

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Reverse Engineering and Reusing COBOL Programs: A Program Transformation Approach

Hongji Yang
Department of Computer Science
De Montfort University
England

Yong Sun
Department of Computer Science
The Queen's University of Belfast
Northern Ireland

Abstract

Since the 1980s, software maintenance started to attract attention. Some progress has been made in using formal methods on software maintenance, especially on reverse engineering. This paper attempts to summarise some major advances in this area over the last one and a half decades. Firstly, we introduce program transformation techniques for software development and review the techniques used for software maintenance. We then describe a method for reverse engineering and reusing COBOL programs using program transformations. Finally, we suggest a future investigation direction for this work.

Keywords: reverse engineering, reuse, COBOL, program transformation

1 Introduction

Program transformation techniques were invented to be used in software development. Recently, a number of projects (cf Section 3) have attempted to apply these techniques to software maintenance. However, there are many problems associated with the application of program transformations to software maintenance. For example, because of the features of COBOL programs, to maintain COBOL programs by using transformations needs certain adjustments. Furthermore, as the demand of reuse increases daily, it is very attractive to apply program transformations into this situation.

This paper gives a review on program transformation techniques for both software development and software maintenance. Then, we use a software maintenance project, the REFORM, as our example to summarise our experiences and lessons. Based on these, we propose a method for reverse engineering and reusing COBOL programs by using program transformations. Finally, we suggest a future investigation direction based on both our experiences and the proposed method.

In software development, the term *refinement* is used to mean a technique which produces a correct implementation from an original specification step by step [22]. Each step is one refinement, which brings us closer to implementation than the previous step. The refinement process can be regarded as a design process. At each design stage, we make a design decision, which may make a new version of the previous specification. However, we need to make sure that the new specification is acceptable with regards to the previous one, i.e. the new one “satisfies” the old one. From the new specification, we can make a further refinement. Therefore, we need to ensure that the series of design decisions interact with each other correctly.

A refinement can be an enhancement of the *functional constraints* imposed upon from a new specification compared with the previous one. At each time, we need to check the new constraint to be consistent with other old constraints. The final implementation needs to be checked that it does indeed satisfy all constraints.

A refinement can be carried out either informally or formally. Let us restrict our attention on a formal approach. In Figure 1, we present a general formal program development method, in which programs were gradually evolved from specifications via a series of refinement steps. One of the potential applications, which is most useful for formal specifications, is a gradual refinement in a formal development of programs from a high-level specification to a low-level “program” or to a (low-level) “executable specification” [12, 19]. Some

$$SP = SP_0 \xrightarrow{\text{refinement}} SP_1 \xrightarrow{\text{refinement}} SP_2 \xrightarrow{\text{refinement}} \dots \xrightarrow{\text{refinement}} P$$

Figure 1:

refinement steps are more or less routine. These routine refinement steps can be described as *transformational rules*. By applying these transformational rules, we can change a program or its specification to a different program or a new specification with the same semantics compared with the original program or specification. The process of the above changing is also called *program transformation*.

Any refinement obtained by instantiating a transformation rule should preserve the semantics. Instead of proving correctness each time for each instantiation, the rule can be proved to be preserving correctness at any circumstance. After that, we can apply this type of semantics-preserved rules as we wish without a further proof. Sometimes, such a rule can only be applied in a correctness-preserved way under certain conditions. These conditions can be met either by the program fragments matching the schematic variables or by the context where the rule is applied. In either case the proof obligation is reduced to checking whether these conditions are satisfied.

Transformational programming is a method of program construction by successive applications of transformation rules. For this method, a design process starts with a formal specification and ends with an executable program. Existing work on the program transformation aims at the development from a specification to an implementation which can be supported formally and mechanically. To improve this situation better, we need to develop appropriate formalisms and notations, to build computer-based support systems, to compile libraries of useful transformation rules, and to develop strategies for conducting the transformation process automatically or semi-automatically. The long term objectives are dramatical improvements in the software construction, software reliability, and software maintainability.

2 Transformation Systems for Software Development

In software development, a feasible method is to derive the final program from a specification. Let us use SP to denote a specification of (user) requirements and P to denote the ultimate program which satisfies the specification SP . A practitioner's way to proceed is to construct P by whatever means are available. He/she may make informal references to SP while he/she is in the design progress. Then, he/she will validate in some ways or the other that P (which he/she produced) does indeed satisfy SP . To convince the user that he/she has delivered the final product, he/she will test P which checks that in certain selected cases the input/output relation of the product satisfies the constraints imposed by SP . This practitioner's method has an obvious disadvantage, i.e., except for some trivial programs, the correctness of P is never guaranteed by the method. Even if the correct outputs are produced in all preset test cases by the product, we will never be sure of non-existence of bugs in the product. An alternative to this method, which only employs testing, is to deploy formal proofs into the method, i.e. we need a supplemental formal proof that program P is correct with respect to specification SP .

Various transformation systems for software development [11, 18, 17] were introduced to overcome the problem faced by the above mentioned practitioner's method. In general, these systems have following features:

1. Purpose

Generally speaking, transformation systems are built to experiment with the mechanically assisted development of a broad range of programs.

A first goal of program transformation is *program synthesis*: the generation of an equivalent, executable, and efficient program from a (formal) description of the problem. Program synthesis may start from specifications in either a (restricted) natural language, or a formal language. The final product should be correct with respect to the specification.

The second goal is a *general support for program modification*. This includes: optimisation of control structures, efficient implementation of data structures, and the adaptation of data structures and given programs to particular styles of programming (e.g., applicative, procedural, machine oriented).

A third goal is that of *program adaptation* to particular environments. For example, a program written in one language (say Fortran 77) may need to be adapted to a related language (say C or C++) with different primitives.

2. Functions

- (a) *Transformation data base*: The system consists of a facility for keeping the (predefined) collection of transformations for use by the end user.
- (b) *User Guidance*: Nearly all transformation systems are *interactive*. Even the “fully automatic” ones require an initial user input and rely (interactively) on the user to resolve unexpected events. The system’s reaction to input may include automatic checks on the “reasonableness” of given commands, as well as incremental interactive parsing using correction mechanisms.
- (c) *History recording*: Most systems also have some facility for *documenting the development process*, i.e. one of the promising aspects of the transformational approach. These facilities include internal preservation of the source program, of final output, and of all intermediate versions. The documentation itself ranges from a simple sequential log of the terminal session (bookkeeping) to rather sophisticated database mechanisms.
- (d) *Assessment of Programs*: Assessment of programs can be supported in qualitatively different ways: the system may incorporate some execution facility, such as an interpreter or a compiler to some target level, or it may utilise aids for “testing”, such as symbolic evaluators. Occasionally the system will also have tools for program analysis, either for aiding in the selection of transformation rules or simply for “measuring” the effect of some transformation.

3. Working Mode

- (a) A “*manual*” system makes the user responsible for selecting and applying every single transformation step. It is the simplest implementation and the system must provide some means for building up compact and powerful transformation rules. The system checks the applications of their usage.
- (b) A *fully automatic* system enables the selection and appropriate rules to be determined completely by the system using built-in heuristics, machine evaluation of different possibilities, or other strategic consideration.
- (c) A *semi-automatic* system works both autonomously for predefined subtasks and manually for unresolvable problems.

4. Type of transformation

Basically, there are two different methods for keeping transformations in the system: the *catalogue approach* and the *generative set approach*.

A *catalogue of rules* is a linearly or hierarchically structured collection of transformation rules relevant for a particular aspect of development process. Catalogues may contain, for example, rules about *programming knowledge*, optimisations based on *language features*, or rules reflecting *data domain knowledge*. A user can select certain transformation rules from the catalogue and apply the selected transformation.

By a *generative set* we mean a small set of powerful elementary transformations to be used as a basis for constructing new rules. A user can decide what transformation rules are to be constructed from the generative set.

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.

Let us look at some existing transformation systems below:

1. *Optimising Compilers* - Program transformation techniques have been used for many years in optimising compilers, because inefficient programs can be transformed into efficient programs (e.g., loop induction, strength reduction, expression reordering, symbolic evaluation, constant propagation and loop jamming).
2. *Burstall and Darlington's Work* - The work on program transformation by Burstall and Darlington was done in the mid-1970's [8, 18]. 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.
3. *Balzer's Work* - Balzer built an implementation system for program transformation [4, 3]. 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).
4. *ZAP* - Feather's ZAP system [11] 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 recursive 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* and *generalisation*.
5. *DEDALUS System* - The DEDALUS system (DEDuctive Algorithm Ur-Synthesiser) by Manna and Waldinger was implemented in QLISP [11]. 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.
6. *The DRACO System* - The DRACO System [18] is a general mechanism for software construction based on the paradigm of "reusable software". "Reusable" here means that the analysis and design of some library programs can be reused, but not their code. DRACO is an interactive system that enables a user to refine a problem, stated in a high level problem domain specific language, into an efficient LISP program.
7. *CIP-S* - CIP-S is the approach of the Project CIP (computer-aided, intuition-guided programming) [5], 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 summarise, 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 nearly 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 a practical use of transformation systems is no doubt the key problem to be solved in transformational programming.

3 Program Transformation for Software Maintenance

Several recent transformation systems (formal and informal) for software maintenance are briefly introduced as follows:

1. *Reverse Engineering in REDO* - REDO (Restructuring, Maintenance, Validation, and Documentation of Software Systems) is an European ESPRIT II project, which ran from 1989 to 1993, and it 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 a part of the REDO project, reverse engineering (reverse-engineering COBOL programs into Z specifications) was carried out at Oxford University [14]. 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.
2. *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 [20]. 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 modularised and restructured as a by-product of the reverse transformation process.
3. *A CASE Tool for Reverse Engineering* - Bachman introduced a CASE tool, DOCMAN, for reverse engineering COBOL programs [2]. His Reengineering Cycle chart provides an architectural view of this CASE tool, which features both forward and reverse engineering. Particularly, reverse engineering begins at the bottom with the definition of existing applications, and later it 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 to be used in the forward engineering process. Then, at the bottom, a new application system becomes an existing application system. It will produce the final product. Basically, this is an informal approach.
4. *TMM* - A method was proposed in [1] 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).
5. *A Concept Recognition-Based Program Transformation System* - This is an approach that applies a transformation paradigm to automate software maintenance activities [10]. 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.
6. *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 (MA). 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 [6, 23, 24].

Most of these approaches have been advocated for reverse engineering, but few have been evaluated in practice on large-scale code. Reverse engineering is often one early part of a software maintenance project.

Many projects have been conducted in this area in seeking a good method to achieve the goal - obtaining specifications from programs.

From the above systems, we know that a great effort is also 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 both availability and accuracy of the design information are presumed. As we know, such information is typically obsolete or lacking in systems, which usually have gone through many years of maintenance. The design information required may be buried deeply in the code and it is hard to be recovered. For such systems, source code is the only reliable source of information. Another problem is that there is not a method for coping with crossing levels of abstraction covering all abstraction levels in these systems.

The state of the art in reverse engineering may be summarised 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 [2].

4 Reverse Engineering and Reusing COBOL Programs

In general, the purpose of reverse engineering is to maintain a software system at a higher level of abstraction. Very often, reuse is also needed as well as reverse engineering. Therefore, based on the research progress on program transformations so far, it may be time to seek a method to integrate reverse engineering and reuse in the same environment.

The work described in this paper aims to extend the REFORM project. Since the REFORM project has made a good effort to use program transformation in tackling software maintenance, to study and develop a method for integrating reverse engineering and reusing COBOL programs in one environment, it will be helpful to have a closer look at the project described in last section (Section 3) and summarise the experience and lessons from the project.

The successes of the REFORM project (including the tool developed in the project - the Maintainer's Assistant) [21] are as follows:

1. use of weakest preconditions expressed in infinitary logic, allowing simple transformation between loops and recursive procedures, which permits very powerful transformations to be included.
2. a small, traceable kernel language, extended via definitional transformations allows very precise and thorough formal semantics to be given.
3. use of an imperative kernel language, with functional constructs added via definitional transformation, rather than a functional kernel language, which allows the technique to be applied to real-world programs, rather than idealised ones.
4. developing the transformation theory in parallel with the wide spectrum language (WSL) development which allowed an incremental style of development of the tool to reflect growing experience with its use.
5. dealing with assembler and similar low-level languages via simple translation followed by automatic restructuring and simplification.
6. developing an interactive, semi-automatic tool, rather than attempting complete automation, thereby making good use of human expert knowledge about the software and its domain.
7. mechanical checking of the correctness conditions on transformation application appearing in the menus.
8. knowledge elicitation: using the prototype and manual case studies to see how the experienced user solves problem, and then implementing these methods and heuristics.

9. the use of generic transformations for merging, moving, separating etc.; these are automatically expanded into the appropriate transformation for each situation.
10. rapid prototyping development, with the system organised as a collection of abstract machines with formally defined interfaces.

However, the prototype of the Maintainer's Assistant in the REFORM Project can only tackle computation intensive program. 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 (often known as typical data intensive programs) though they confirmed the potential capability of the Maintainer's Assistant, because:

1. 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.
2. 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.
3. Most of the program transformations that currently are implemented can only be used for restructuring programs at relatively low levels of abstraction.
4. No representations of types, complex data structures and data design yet exist in WSL.

These facts urged a new research direction to be set up i.e., to employ program transformation technique emphasising data abstraction, to end up with data designs from programs written in e.g. COBOL and to reuse reusable components in reengineering.

It is believed that there are 800 billion lines of COBOL programs existing in the world [15] and the result of the research can be applied to maintain COBOL software. The COBOL language used in this research is unrestricted to any dialect of COBOL. It also covers some features in ANSI COBOL Standard 1985.

Programs written in COBOL have characteristics which are different from those computation-intensive programs. These characteristics are important constraints in reverse engineering the systems written in COBOL. For example, COBOL programs have following characteristics:

1. Important data is represented in the form of records, and operations on data are therefore heavily record-based.
2. COBOL programs are often designed using Entity Relationship Diagrams, rather than process based design methods.
3. 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.
4. COBOL programs usually have external calls to the operating system and database management system.
5. COBOL programs may use many *foreign keys* to represent complex data structures where in other languages pointers would be used.

4.1 Extension of WSL

The motivation for acquiring a data design from COBOL code is still that software can be best understood, altered and enhanced at the conceptual level rather than at the code level where the maintainers's view is often obstructed by implementation details. Also, this is an essential step to reuse the existing code.

One of the characteristics of COBOL 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 data structure to another. 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 in 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 preservation of semantics and suitability for tools, etc.

WSL (a formal specification language used in REFORM) 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. This means that a WSL variable can at different times hold values of different types. Adding type is essential to avoid losing important attributes of the source program, such as logical connections between data. Therefore, data structuring such as records are needed. COBOL is built on a low level model of storage, involving the explicit layout of data in memory, the size of data in characters, etc. A challenging problem for reverse engineering is the use of aliasing to use memory for several purposes. Since COBOL treats all significant data as records, defining "records" in WSL for modelling COBOL records is a clear requirement.

The external calls to the underlining operating system and the embedded database can be modelled 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 analyse the code with foreign keys, and the relations between those modules using foreign keys could be found by these analysis.

Entity Relationship (ER) Diagrams are based on *entity models* [9, 16]. Entity models provide a systematic view of the data structures and data relationships within the system. All systems possess an underlying generic entity model which remains fairly static in all times. The entity model reflects the logic of the system data, not the physical implementation. Entity models provide an excellent graphical representation of the generic data structures and relationships. Therefore, Entity Relationship Diagrams are suitable forms for representing data designs for COBOL programs and a compatible extension of WSL is therefore needed to include Entity Relationship Diagrams.

4.2 A Method for COBOL Program Reverse Engineering and Reuse

As discussed in the earlier part of the paper, reverse engineering existing software to a higher level of abstraction, e.g., recovering design, is a crucial step in software maintenance. And reengineering existing software involves a very important part, i.e., reusing software components. The objectives of our research are to integrate reverse engineering and reuse in one environment, emphasising design recovery and reusable component identification.

A method for design recovery and identifying reusable components through program transformation is proposed as follows:

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 be 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 ER diagrams.
5. The obtained ER diagrams are viewed as reusable components. The ER 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 pre-conditions, 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, a tool needs building (Figure 2), which mainly includes extension of WSL according to the analysis explained in the previous subsection and development of transformations for dealing with COBOL programs.

Figure 2:

4.3 Development of Transformations

WSL was extended by adding more constructs for describing COBOL programs, Entity Relationship Diagrams and pre-conditions and post-conditions of reusable components [23]. When WSL was extended, development of transformations for dealing with COBOL programs started. Transformations developed for extracting Entity Relationship models in our proposed tool are divided into seven categories (see [24] for a recent reference):

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 - a record with just one level of subrecord can be abstracted to an entity with attributes and (2) acquiring a relationship from a record with subrecords - when a subrecord of a record can be abstracted into an entity and the subrecord can be

abstracted into an entity as well, there exists a relationship between the entity derived from the record and the entity derived from the subrecord.

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 a language like REFORM's 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 proposed 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 recognising 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 (or 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 category address how ER models are extracted from code, in particular, from assignment statements, branching structures and loop statements. For example, an assignment statement is a simple but straightforward measure to implement a relation between two data objects, which, at the data design level, may be two entities. Therefore, an assignment can be usually abstracted to a relation. A branching statement, such as **if .. then .. else .. fi** can be abstracted to a sequence of two groups of statement (each group comes from each arm of the **if ... fi** statement). Looping statements, such as **while** and **for**, can be removed just leaving the body of the loop.

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 [7]. Our work, at this stage, is focusing 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 specialised 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 consists 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 [13]; 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.

Annotating Predicates for Reusable Components A systematic abstraction for reusable components is based on formal predicates, i.e.:

- * annotating a predicate to each component,
- * propagating the predicate to a higher level component, and
- * recognise the required predicate conditions for abstraction rules (if the predicate conditions hold the abstraction can be achieved).

The idea is to use data semantics and operation semantics in programs to infer a high level abstraction and semantics. An example in this paper (to be seen in the next subsection) is to use semantics of data structure *sequential-file* and a loop of record copy to infer that the module *file-backup*(*File1*, *File2*) has the post-condition, EQU(*File1*,*File2*), i.e. we have inferred and abstracted semantics of module *file-backup* from its composed components. However this inference can only be achieved when proper predicates are annotated with software components. Although the annotating process may be an overhead, it usually helps to reveal the embedded semantics, which is needed during the comprehension process. In other words, it should reduce guessing work and clarify the semantics during the process. This approach can be achieved with the help the Knowledge Base in the prototype tool. The Knowledge Base plays a vital role in annotating predicates for the reusable components. Knowledge Base contains two major information: software templates, representing knowledge, including domain independent and specific knowledge of software components in the formal predicate format; abstraction rules, representing conditions for abstraction.

4.5 An Illustration

In this subsection, we use an example to illustrate how to translate a COBOL program into WSL, how to transform this program into a data design represented in an Entity Relationship Diagram, how to identify a reusable component and how to acquire semantic interface predicates for a reusable component. The example program is originally in COBOL and its COBOL source code is as follows:

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

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SOURCE-FILE-NAME ASSIGN TO XYZ
        ORGANISATION SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL.
    SELECT TARGET-FILE-NAME ASSIGN TO WXY
```

```

      ORGANISATION SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL.

DATA DIVISION.
  FILE SECTION.
    FD SOURCE-FILE-NAME.
    01 SOURCE-RECORD          PIC X(50).

    FD TARGET-FILE-NAME.
    01 BACKUP-RECORD          PIC X(50).

  WORKING-STORAGE SECTION.
    01 EOF PIC X.

PROCEDURE DIVISION.
MAIN.
  OPEN INPUT SOURCE-FILE-NAME
    OUTPUT TARGET-FILE-NAME
  PERFORM, WITH TEST AFTER, UNTIL EOF = "T"
    READ SOURCE-FILE-NAME NEXT;
    AT END
      MOVE "T" TO EOF
    NOT AT END
      MOVE "F" TO EOF
      MOVE SOURCE-RECORD TO TARGET-RECORD
      WRITE TARGET-RECORD;
  END-PERFORM
* THE STOP RUN STATEMENT CLOSSES THE FILES
  STOP RUN.

```

Table 1. A File-backup Program in COBOL

The program is translated into its equivalent form in WSL (Table 2). The program module was a procedure in the original program and it was called by a (COBOL) PERFORM statement. This program copies the contents in one file to another file. Table 2 shows the format of the program when loaded in to the transformation tool of the prototype.

The identification division in COBOL is translated into a **comment** statement in WSL. Information in the environment division will be used when data division and procedure division are translated, i.e., the files in the code are sequential files. In the data division, COBOL records and files are translated into WSL records and files. COBOL files in this example are sequentially organised and sequentially accessed.

In this program original file operations are translated into WSL as external procedures (denoted by !p which is the WSL function to call an external procedure for which it is known definitely which variables will be changed) or external functions (denoted by !f which the the WSL function to call a named external function).

```

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

```

Table 2. The File-backup Program in WSL

The above program is then dealt with by the Program Transformer, which applies transformations to it. The process to move from code to design/specification level and finally to be annotated with formal pre-conditions, post-conditions and obligations is a process of crossing levels of abstraction and the program will become more abstract when abstraction program transformations are applied. This process is also a process of code understanding.

To understand the module, the program transformation first transform two records in the program into two data entities, i.e., those two records in the module can be viewed as two data entities. The entity keeps the same name as the record.

Operations on a sequential file are treated as operations on a “mathematical” queue: to “read” a record being reading a record from a queue, to “write” a record to a file being writing a record to a queue, and to test whether the file operation pointer is pointing to the end of the file being testing whether the pointer is pointing the end of the queue.

When a record is transformed into an entity, any program statement using this record must be changed accordingly. Because transforming a record into an entity is an abstraction, the statement using the obtained entity must be expressed at a higher abstraction level. In this case, the assignment statement in the module should be viewed as those two entities (originally two records) are related or linked.

Other two assignments in the “if” statement was used as control purpose should be viewed as something that would not exist at the high abstraction level and therefore they can be ignored.

The looping statement, **while**, is also used as a control structure in implementing programs but did not appear in the original program design. A looping statement can be treated as enumerating the same operation on every instance of entities. The condition part of the loop also does not contribute to the Entity Relationship diagram. So a **while** loop can be removed just leaving the body of the loop.

The original program is supposed to implement copying all the records from the original file to the backup file. At the higher abstraction level, it is to say that there is a “copy” relationship between two entities. Each record is one instance of an entity. The original program is transformed into (Table 3):

```

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

```

Table 3. An Entity Relationship Diagram for the File-backup Program in WSL

After applying transformations discussed in this section, the final result of the “file-backup” program can be shown by an Entity Relationship diagram (Figure 3).

Figure 3:

When the original program was transformed into an Entity Relationship diagram, the user can easily decide that the program segment can be a good candidate of a reusable component. Therefore, the component in WSL (Table 2 and 3) will also be analysed by the Semantic Interface Analyser(SIA) in order to generate a form annotated with formal pre-conditions, post-conditions and obligations.

To demonstrate how SIA works, we list the software templates and abstraction rules, which are related to the File-Backup program shown in Table 2. in Knowledge Base, as follows:

Software Templates For each type of software component, such as external function, system function, data type, operation, statement, data structure, etc., the Knowledge Base should have a corresponding template for it, which describes its semantics and characteristics.

For the declaration of file structure,

pre-condition

!Sequential-File(file-name, record-name)

post-condition

!EQU(file-name, array[1..N] of record-name)

file file-name with

record record-name with

end;

end;

For function open_file, where “?” represents that any record type make this condition TRUE,

pre-condition

!Sequential-file(file-name, ?)

post-condition

!Index-of-Seq-File(1, file-name)

!Open-File(file-name)

!p open_file (i **var** file-name);

For function *eof?*(*file-name*),

pre-condition

!Open-File(*file-name*)
 !Sequential-file(*file-name*, ?)
 !EQU(*file-name*, array[1..N] of ?)
 !Index-of-Seq-File(*i*, *file-name*)

post-condition

!if *i* > *N* then *eof?*(*file-name*) = TRUE
 !else *eof?*(*file-name*) = FALSE

eof?(*file-name*)

For module read,

pre-condition

!Open-File(*file-name*)
 !Sequential-file(*file-name*, ?)
 !EQU(*file-name*, array[1..N] of ?)
 !Index-of-Seq-File(*i*, *file-name*)

post-condition

!EQU(*record*, *file-name*[*i*])
 !Index-of-Seq-File(*i* + 1, *file-name*)

read(*record*, *file-name*)

For an assignment statement,

pre-condition

!EQU(Type(*id1*), Type(*id2*))

post-condition

!EQU(*id1*, *id2*)

id1 = *id2*

For module write,

pre-condition

!Open-File(*file-name*)
 !Sequential-file(*file-name*, ?)
 !EQU(*file-name*, array[1..N] of ?)
 !Index-of-Seq-File(*i*, *file-name*)

post-condition

!EQU(*file-name*[*i*], *record*)
 !Index-of-Seq-File(*i* + 1, *file-name*)
write(*record*, *file-name*)

Abstraction Rules For the iteration variable i in a loop statement,

pre-condition

$\text{!Iteration-Index}(i)$
 $\text{!EQU}(A, \text{array}[1..N] \text{ of } \text{record-name})$
 $\text{!EQU}(B, \text{array}[1..N] \text{ of } \text{record-name})$
 $\text{!LEQ}(i, N)$ $\implies \forall i \text{ EQU}(A[i], B[i])$

post-condition

$\text{!Iteration-Index}(i + 1)$
 $\text{!}A[i] = B[i]$

and

pre-condition

$\text{!Sequential-File}(\text{File1}, \text{record-type1})$
 $\text{!Sequential-File}(\text{File2}, \text{record-type2})$
 $\text{!EQU}(\text{record-type1}, \text{record-type2})$ $\implies \text{EQU}(\text{File1}, \text{File2})$

post-condition

$\text{!}\forall i \text{ EQU}(\text{file1}[i], \text{file2}[i])$

Table 4 shows the module Backup_File after the analysis and the condition propagation by the Semantic Analyser was done and Table 5 shows the predicates having been propagated from and to be attached to the original program module, which can be used for further checking when it is reused.

comment: “program-id: file-backup”;
pre-condition
 $\text{!Sequential-File}(\text{source-file-name}, \text{source-record})$
 $\text{!EQU}(\text{source-file-name}, \text{array}[1..N] \text{ of } \text{source-record})$
 $\text{file } \text{source-file-name}$ **with**
 $\text{record } \text{source-record}$ **with**
 end;
 end;
post-condition
 $\text{!Sequential-File}(\text{target-file-name}, \text{target-record})$
 $\text{!EQU}(\text{target-file-name}, \text{array}[1..N] \text{ of } \text{target-record})$
 $\text{file } \text{target-file-name}$ **with**
 $\text{record } \text{target-record}$ **with**
 end;
 end;
 $\text{eof} := 0;$
pre-condition
 $\text{!Sequential-file}(\text{source-file-name}, ?)$
post-condition
 $\text{!Index-of-Seq-File}(1, \text{source-file-name})$
 $\text{!Open-File}(\text{source-file-name})$
 $\text{!p } \text{open_file} (i \text{ var } \text{source-file-name});$

```

pre-condition
!Sequential-file(target-file-name, ?)
post-condition
!Index-of-Seg-File(1, target-file-name)
!Open-File(target-file-name)
!p open_file (o var target-file-name);
while (eof  $\neq$  1) do
  pre-condition
  !Open-File(source-file-name)
  !Sequential-file(source-file-name, ?)
  !EQU(file-name, array[1..N] of ?)
  !Index-of-Seg-File(i, file-name)
  post-condition
  !if i > N then eof?(file-name) = TRUE
  !else eof?(file-name) = FALSE
  if non-empty? (!f eof? (source-file-name))
    then eof := 1
    else eof := 0;
    pre-condition
    !Open-File(source-file-name)
    !Sequential-file(source-file-name, ?)
    !EQU(source-file-name, array[1..N] of ?)
    !Index-of-Seg-File(i, source-file-name)
    post-condition
    !EQU(source-record, source-file-name[i])
    !Index-of-Seg-File(i + 1, source-file-name)
    !p read_file (source-record var source-file-name);
    pre-condition
    !EQU(Type(target-record), Type(source-record))
    post-condition
    !EQU(target-record, source-record)
    target-record := source-record;
    pre-condition
    !Open-File(target-file-name)
    !Sequential-file(target-file-name, ?)
    !EQU(target-file-name, array[1..N] of ?)
    !Index-of-Seg-File(i, target-file-name)
    post-condition
    !EQU(target-file-name[i], target-record)
    !Index-of-Seg-File(i + 1, target-file-name)
    !p write_file (target-record var target-file-name);
  fi;
od;

```

Table 4. The File-backup Program with annotated predicates

Based on predefined templates in the Knowledge Base, predicate analysis and propagation, we can infer that the module *file-backup* contains the following predicates:

```

pre-condition
!Sequential-file(File 1, record-type1)
!Sequential-file(File 2, record-type2)

```

```

!EQU(record-type1, record-type2)
post-condition
!EQU(File1, File2)

file-backup(File1, File2)

```

Table 5. Semantic Interface Predicates Generated

5 Concluding Remarks and Future Investigations

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

In the overall process of software engineering, software maintenance is still the most expensive stage in the software life cycle. The enormous maintenance problems desperately await their solutions. COBOL programs constitute a large proportion of existing programs in the industries and their maintenance should be given a close attention.

Program transformations are a powerful tool in reverse engineering for existing COBOL programs and they also provide a facility for reuse of these COBOL programs. Our approach of dealing with COBOL programs is to derive a program data design from a COBOL program through program transformations, to represent designs in Entity Relationship Diagrams and to annotate reusable components with pre-conditions, post-conditions and obligations through Semantic Interface Analysis techniques. Our method is sufficiently general to apply to programs written in other source languages besides COBOL, although COBOL programs have been mainly used in our experiments. Our method only requires source code as its input and it can be applied to heavily modified codes, e.g. the programs have been maintained over many years.

Our proposed method has an implementation of the prototype. This shows that our approach has a scope from theory to practice, i.e. it has a potential to be used by industries.

The abstract Entity Relationship Diagrams are able to represent the designs of the original programs. The correctness of the obtained ER diagrams is at present checked manually based on human knowledge and expertise in our method. The role of human knowledge is to guide the whole application process of the method. When COBOL programs are translated into WSL (in the case of a COBOL-to-WSL translator having not been built or even when building the translator), human knowledge has to be used to ensure that WSL programs are semantically equivalent to the original COBOL programs. It can be seen that human knowledge plays a decisive role in choosing transformations, in particular, abstraction transformations. It is also human knowledge and domain expertise that determine whether the obtained Entity Relationship Diagrams represents a reasonable data design of the original COBOL program.

Research work on dealing with COBOL 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 be useful in incorporating other techniques, such as Object Oriented technology.

Combining program transformations with object orientation will also be an very important issue. Initial thoughts about introducing an object-oriented approach into the existing systems; for example, we need to extend the existing kernel and re-build a WSL kernel language in order to develop an object-oriented WSL. We also need to develop transformations under the object-oriented approach for reverse engineering COBOL programs, to develop a new method or to extend the existing method for COBOL program reuse, etc.

When the source code is very large, the method developed in this paper would still work well provided that program segments are a manageable size. However, the scale of the source code would affect the method in a situation where smaller program segments are not self-contained, i.e., self-contained segments are already not in a manageable size. Nevertheless, one possible solution could be to build an Information Database. When the Program Transformer is working on a program segment which has many calls to other segments, it is the Information Database that collects all the necessary information from the called segments for the Transformer so that the Transformer does not need to load in those segments.

Experiments carried out so far have only been focused on capturing reusable COBOL program components. Future work needs to be conducted in adapting and integrating these reusable components into a new system being developed or being reengineered.

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

6 Acknowledgement

We would like to thank Dr. P. McParland of Queens University of Belfast for his comments on an early version of this paper.

References

- [1] Arango, G., Baxter I., Freeman P. and Pidgeon C., “TMM: Software Maintenance by Transformation”, IEEE Software, May, 1986.
- [2] Bachman R., “A CASE for Reverse Engineering”, reprinted from DATAMATION, Cahners Publishing Company, July, 1988.
- [3] Balzer R., “A 15 Year Prospective on Automatic Programming”, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985.
- [4] 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.
- [5] Bauer F., Moller 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, February 1989.
- [6] Bennett K., Bull T. and Yang H., “A Transformation System for Maintenance — Turning Theory into Practice”, IEEE Conference on Software Maintenance-1992, Orlando, Florida, November, 1992.
- [7] Biggerstaff T. and Ritcher C., “Reliability Framework, Assessment and Direction”, IEEE Software, Vol. 14, No. 4, 1987.
- [8] Burstall R. and Darlington J., “A Transformation System for Developing Recursive Programs”, Journal of the ACM, Vol 24, pp. 44–67, 1977.
- [9] Chen P., “The Entity-Relationship Model — Toward a Unified View of Data”, ACM Transaction on Database Systems, Vol. 1, No. 1, March, 1976.
- [10] Engberts A., Kozaczynski W. and Ning J., “Concept Recognition-Based Program Transformation”, IEEE Conference on Software Maintenance-1991, Sorrento, Italy, 1991.
- [11] Feather M., “A Survey and Classification of Some Program Transformation Techniques, Program Specification and Transformation”, Elsevier Science Publishers, The Netherlands, pp.165–195, 1987.
- [12] Fickas S. “Automating the Transformational Development of Software”, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November, 1985.
- [13] Frakes W. and Pole P., “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.
- [14] Lano K. and Breuer P., “From Programs to Z Specifications”, Technical Report, Programming Research Group, Oxford University, 1990.

- [15] Layzell P., "The Identification and Management of Latent Software Assets", International Journal of Information Management, Vol. 14, No. 6, 1994, pp.427–442.
- [16] Martin J. and McClure C., *Structured Techniques for Computing*, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1985.
- [17] Partsch H., *Specification and Transformation of Programs*, Springer-Verlag, London, 1990.
- [18] Partsch H and Steinbruggen R., "Program Transformation Systems", Computing Surveys, Vol. 15, No. 3, September, 1983, pp.198–236.
- [19] Sannella D and Tarlecki A., "Toward Formal Development of Programs from Algebraic Specification: Implementation Revisited", ACTA Informatica, No. 25, pp.233–281, 1988.
- [20] Sneed H. and Jandrasics G., "Inverse Transformation of Software from Code to Specification", IEEE Conference on Software Maintenance-1988, Phoenix, Arizona, 1988.
- [21] Ward M. and Bennett K., "A Practical Program Transformation System for Reverse Engineering", IEEE Conference on Software Maintenance-1993, Montreal, Canada, 1993.
- [22] Whysall P., "Refinement", *Software Engineer's Reference Book*, Butterworth Heinemann, 1991.
- [23] Yang H., "Acquiring Data Designs from Existing Data Intensive Programs", Ph.D. Thesis, Durham University, 1994.
- [24] Yang H. and Bennett K., "Acquiring Entity-Relationship Attribute Diagrams from Code and Data through Program Transformation", IEEE International Conference on Software Maintenance (ICSM '95), Nice, France, October, 1995.