JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 15, 173-198 (1999)

Acquisition of Entity Relationship Models for Maintenance–Dealing with Data Intensive Programs in a Transformation System

HONGJI YANG AND WILLIAM C. CHU*
Department of Computer Science
De Montfort University
England
**Department of Information and Computer Science*
TungHai University
Taichung, Taiwan 407, R.O.C.
E-mail: chu@cis.thu.edu.tw

This paper presents results of a research programme on reverse engineering using a transformation system for maintenance and focuses on dealing with data-intensive programs such as those written in COBOL. Problems with data-intensive programs are addressed, our solutions to these problems are discussed and the results of experiments are presented. It is concluded that formal transformations provide a way of combining design decisions which have become instantiated in both the code and the data structures.

We describe a solution to the problem of acquisition Entity Relationship (ER) diagrams from data-intensive source code. In such programs, the relationships between data items are often represented within imperative code as well as within data structures, and we show that reverse engineering can be improved if both are used. This distinguishes our work from other works in the field.

Our method is based on formal transformations. We identify imperative constructs which improve the high-level ER models that can be captured. Suitable transformations are then briefly summarised. A series of experiments with industrial COBOL programs is described. Our results show that code-embedded relations can be usefully incorporated into data intensive reverse engineering, and that they enhance the designs extracted.

Keywords: reverse engineering, program transformations, data-intensive programs, COBOL

1. INTRODUCTION AND PROBLEM DEFINITION

A data-intensive program is one in which much of the complexity and design effort is concentrated in the data structure design, and in which the imperative code is relatively straightforward. Many commercial programs written in COBOL are of this type. In attempting to reverse engineer such programs to a design notation, such as in ER models [5], it is sensible to focus on the data structures in the source code. A number of tools and methods have successfully addressed this problem [1, 10].

However, it is evident when such programs are inspected that important data relationships are often implicitly included in the imperative code as well as in data structures. These need to be included in the reverse engineering process if the maximum amount of information is to be extracted from the source code.

Received August 15, 1996; accepted April 18, 1997.
Communicated by Y. S. Kuo.

The objectives of this research were, therefore, to determine to what extent information from both code and data can be combined in order to extract ER models. The aim of producing an ER model is to aid program comprehension. This paper focuses on five types of implicit information, and case studies using commercial COBOL programs are described.

In our research into reverse engineering, data-intensive programs were in general performed using COBOL as a data-intensive programming language. However the research results can be applied to systems written in other languages. The design of our tool means that language-dependent issues are encapsulated in a front-end translator.

Programs written in COBOL have characteristics which are different to those of typical computation-intensive programs, and there 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 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.

The principal problem which is investigated here is the extraction of ER models from code and data in data-intensive programs. This has stimulated several subsidiary problems.

The first problem is concerned with so-called foreign keys. A data value in one data structure is used to index another data structure, i.e., a[b[i]]. The connection between the data structures is only apparent upon comprehending the code, yet it may be an important relationship in the ER model.

The second problem is concerned with aliasing, in which one piece of memory is used, at different points in the program, for different purposes. Thus, writing to memory may have a side effect. In some languages, memory may be aliased to variables with different types.

The third problem addresses the combination of data and code into abstract data types. This problem has also been studied elsewhere [4], and our own contribution here has focused on extensions to WSL and the transformations needed, as well as on developing a method which is suitable for a transformation approach.

The fourth problem describes a class of code constructs where some variable y depends on others x1, x2, ..., i.e., y := f(x1, x2, ..). For an ER model, the crucial information is that the variables are related; the exact nature of the function f is less important.

The fifth problem addresses input and output, which is often a substantial part of data intensive programs. We have partially solved this by modeling I/O as operations on high-level data types.

The final problem has not yet been addressed in depth; it concerns how to handle pointers in languages such as C.

The organization of the remainder of this paper is as follows. In the next section, a very brief summary of the transformation systems used in the research is given. In section 3, solutions to the above problems are described as well as methods for extracting basic ER models from data structures. Section 4 presents the results of case studies while section 5 summaries our conclusions. Full details of the WSL and transformations are given in [13].

2. BACKGROUND

2.1 Formal Transformation

A program transformation is an operation on a program or part of a program. It transforms the program from one form to another while preserving its semantics. If the program transformation has been proven to be correct, and if the program is expressed in a language which is also formally defined, we can be confident that the semantics are indeed preserved. A trivial example is afforded by assignment merging, so that (y := x+2; z := y + 3) can be replaced by the equivalent z := x + 5. One of the difficult aspects of designing a practical transformation system is the need to check that a transformation is applicable; for example, the above assignment makes a number of assumptions: that "+" is arithmetic, that the variables do not have side effects and so on.

Most of the research on transformation systems, hence, has been heavily concerned with wide spectrum language design since this is the notation for both programs and proofs. The language is wide spectrum because it has to deal simultaneously with constructs at low and high levels of abstraction, and it is inconvenient to introduce several intermediate languages. The design must be aimed at making proofs tractable as well as expressing programming concepts.

Formally, transformations are defined as mappings from initial states to sets of final states. This permits extra flexibility: transformations may provide exact semantic equivalence, or they may refine the semantics [11]. The latter allows, for example, a specification to avoid choices - we may wish to specify a non-deterministic choice while in a concrete implementation, this decision must be made. If we are reverse engineering, we often wish to use refinement transformations in reverse, i.e., use them as abstraction transformations [13].

2.2 Sneed's Work

Sneed and Jandrasics used automated tools to support the retranslation of software code in COBOL back into an application specification by means of reverse engineering [10]. Two steps are needed, firstly to recover a program design from the source code and secondly to recover a program specification from the program design.

In the first step, the code of the COBOL program is translated into an intermediate design schema based on a set of normalised relational tables for the modules, data capsules, and interfaces extracted from the source programs.

Then, two actions are carried out jointly, i.e., data design recovery and program design recovery. The data design part contains five design elements: database structure design, file design, data communication design, data capsule design, and data constant design. The program design part also contains five parts: process structure design, component design, data flow design, module interface design, and module design.

A set of transformation rules for mapping COBOL source code back into a 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.

In the second step, the intermediate design representation is retransformed into a specification schema based on the entity/relationship model. The details of how the authors achieve this are not available.

Though the authors have claimed that "it is not only possible to retranslate programs into a program design, but that it is also possible to retranslate a set of program designs into a system specification," the experiments that they carried out were mainly limited to a low level of abstraction, so there is still work to do to reach design abstraction. It seems that the authors did most of their work by hand and have not yet developed a fully automated system yet.

2.3 A CASE Tool for Reverse Engineering

Bachman introduced a CASE tool for reverse engineering of COBOL programs [1]. The Re-Engineering Cycle chart (Fig. 1) provides an architectural view of this CASE tool, which features both forward and reverse engineering. In particular, reverse engineering begins with the definition of existing applications and raises the applications to successively higher levels of abstraction. At the top level, the design objects created by the reverse engineering steps are enhanced and validated so that they become revised design objects used in the forward engineering process. At the lowest level, a new applications system becomes an existing applications system when it goes into production.

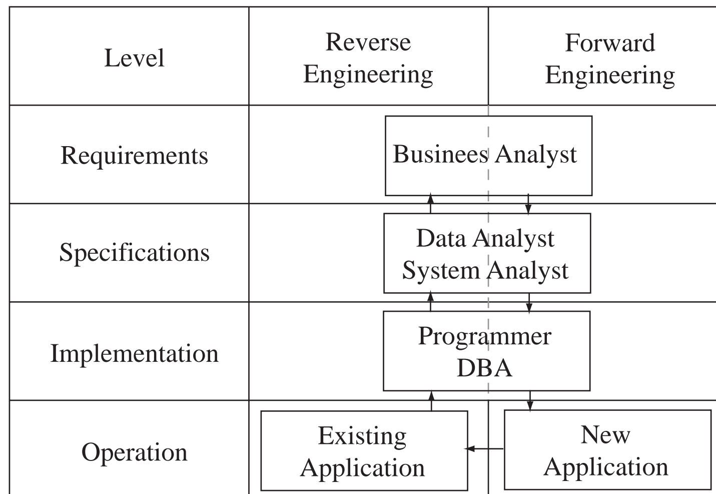


Fig. 1. Re-engineering cycle.

It's claimed that this approach has the following advantages:

- Reverse engineering enables the CASE tool to extract business rules from old applications and use them as the basis for refurbishing and maintaining those applications.
- Reverse engineering also involves the removal of optimization mechanisms and implementation artifacts that were introduced in an earlier implementation of the application.
- It is impossible to reverse engineer a file, database definition, or program automatically because some of the information essential to the task is not present in existing COBOL programs.
- A reverse engineering product built as an expert system can work interactively with the professional user and can identify the missing information, determine its nature, propose alternatives, and insert the user's choice where required to complete the process.

Few actual results and case studies are available for this work.

2.4 Limitations of the Existing Maintainer's Assistant

The Maintainer's Assistant is the name of a system which has been developed over 10 years at Durham University; recently, application of formal transformations to reverse engineering has been undertaken within the REFORM project. It is based on a theory of program equivalence and refinement, and uses semantics-preserving and refining transformations as the underlying foundation. Apart from a language dependent front end translator, the Maintainer's Assistant totally operates using a wide spectrum language, WSL. The prototype of the Maintainer's Assistant is shown in Fig. 2 [14]. This research has concentrated on using transformations for reverse engineering, and full details of our theory, methods and tools, and results can be found in [2, 3, 11, 12, 14, 15]. When the REFORM project progressed to the stage described in [2], the following points were noticed:

- After seeing a demonstration of the prototype of the Maintainer's Assistant, many industrialists were disappointed that the tool was unable to deal with COBOL programs.
- Almost all the program transformations in the transformation library based on Ward's work [11] mainly dealt with functional abstraction (or control abstraction)– most of the transformations operated on the control structures of a program while few transformations were available for data structures. In another words, the system was only suitable for reverse engineer computation-intensive programs, not data-intensive programs.
- Most of the program transformations that are currently implemented can only be used for restructuring of programs at relatively low levels of abstraction.
- No representations of types, complex data structures and data design exist so far in WSL.

These observations suggest that our research should address data-intensive programs and employ program transformation techniques which emphasize data abstraction with the aim of obtaining data designs.

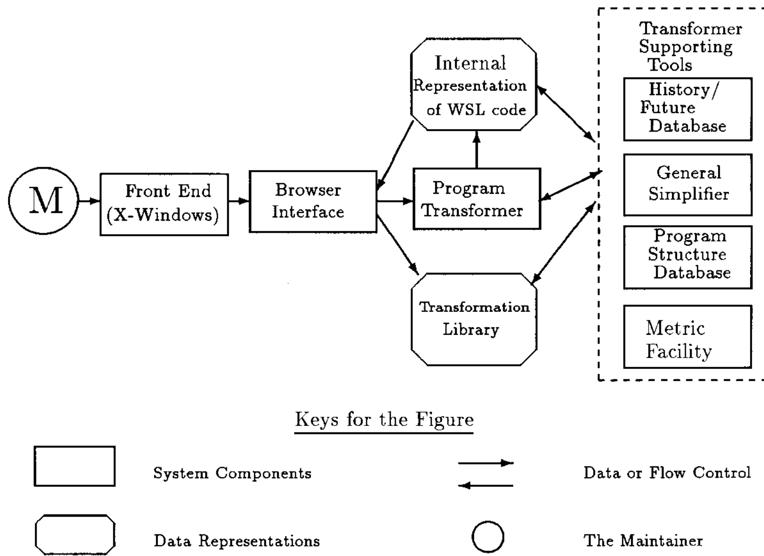


Fig. 2. Prototype of the maintainer's assistant.

3. ACQUIRING ENTITY RELATIONSHIP MODELS

This section describes in detail solutions to the problems raised in section 2 in terms of the source code structures, a summary of the transformations defined, and the ER components that result. Each section takes a similar approach. The new WSL constructs are introduced briefly. Then the transformations that have been designed are described, and it is shown how they are used to extract a particular component of the ER model. In section 5, we shall investigate how well our theory worked out in practice.

3.1 Recovering Data Designs

In order to solve this problem, it was necessary to address the method, the theory of the new approach and a new tool to implement the method developed.

3.1.1 Combining code analysis with data abstraction

The motivation for extracting a data design from data-intensive code is 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. This means that abstraction is needed in order to move from code to a data design.

One of the characteristics of data intensive third generation languages is that high level data designs often translate at the implementation level into 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 examining

the data and the code, not the data alone. Existing reverse engineering techniques have difficulty handling this task. It seems to us that formal transformation offers a possible way to solve this problem.

3.1.2 Program transformations and WSL

A transformational system for extracting data designs which has to cope with data abstraction needs extensions beyond the transformation system that exists in REFORM. A wide spectrum language is the foundation for REFORM, so developments must be represented in WSL. We do not wish to invalidate the proofs and transformations that have already been developed, but rather to build on them.

3.1.3 Analysis of problems with data-intensive programs

WSL currently has declarations which introduce the name of an identifier without its type. Therefore, variables are not typed, but all the 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 the 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 an explicit layout of data in memory, the size of the data in characters, etc. The most 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 modeling of COBOL records is a clear requirement.

There are two issues to address in solving the aliasing problem: it is necessary to determine which records are aliased; and, more challenging, it is necessary to determine a mapping between the different records, based on the memory used by each component of the record. The former can usually be determined using the declarations, and the latter can be done by defining a function which maps from a record to a sequence of bytes (the representation of that record in memory), and from a sequence of bytes to a record. These functions need to know the structure of the record in terms of the number of bytes occupied by each component. Thus, a WRITE to aliased memory is described by a function which maps the COBOL data structure to low level memory; a READ is represented by a function which describes mapping in the reverse direction. In our system, these functions are explicitly inserted in preparation for later simplification using transformations.

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 means of program transformations. These transformations analyze the code with foreign keys, and relations between modules using foreign keys can be found. An example will be presented in section 6.

Entity Relationship Diagrams are based on *entity models* [5-7, 9]. Entity models provide a system view of the data structures and data relationships within the system. All systems possess an underlying generic entity model which remains fairly static over time. 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 forms suitable for representing data designs for data-intensive programs, and WSL needs to be extended to include Entity Relationship Diagrams.

3.1.4 A design recovery method

Based on the above analysis, a method for data design recovery is proposed and is illustrated in Fig. 3. The method consists of following major steps:

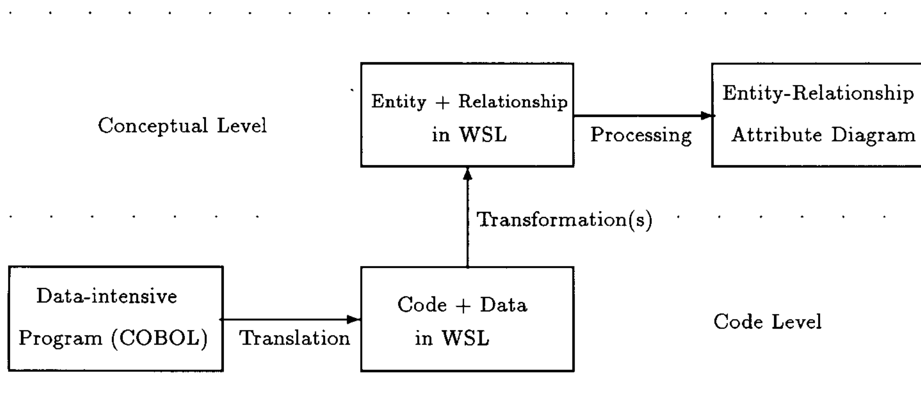


Fig. 3. A data design recovery method.

(1) Translating a data-intensive program (in COBOL in this case) into an intermediate language WSL.

(2) Applying program transformation(s) to the program in the intermediate language; initially, this is at the code level, but as reverse engineering progresses, we shall obtain entities and relationships (in the same intermediate language) but at the conceptual level.

(3) Translating the entities and relationships in the intermediate language into entities and relationships in some notations used in a tool for displaying or printing an Entity Relationship Diagram.

It is stressed that the main reason for using the wide spectrum language WSL is that this language can potentially represent both COBOL programs and Entity Relationship diagrams. As a result, we do not have to deal with the problem of translating between several languages during transformation. Also, program transformations need devising once no matter what language the source code was written in.

3.2 Enhancement of the Maintainer's Assistant

From Fig. 2, it can be seen that the Maintainer's Assistant needs to be substantially enhanced. The enhancements include:

(**1**) **Extension of WSL**
Additional WSL constructs are needed at both the code level and the conceptual level, e.g., constructs for representing data structures at the code level and Entity Relationship languages at the conceptual level.

(**2**) **Extension of Program Transformation Library**
Program transformations are needed to manipulate code and data at both the design and implementation levels, and also for crossing levels of data abstraction.

(**3**) **Extension of Structure Database and Design of Metrics Facility**
The tool itself needs to be extended internally; in particular new program structure database queries are required to support code analysis and transformation implementation.
The objectives of using metrics in REFORM are to help the user to select transformations (to help develop heuristics), to measure the progress made in optimizing the program code and to measure the resulting quality of the program being transformed. These also require extension.

(**4**) **Extension of Browser Interface and Front End**
Corresponding changes have to be made to accommodate the addition of the above extensions to the original prototype.

3.3 Extension of WSL

WSL constructs extended in this research are divided into four categories.

- **Constructs for Representing Records and Files**
  These constructs are used to represent records and files in data-intensive programs at the code level. Aliased records can be represented as well.
- **Constructs for Basic Data Types and User-Defined Data Types**
  These constructs are used to enhance WSL in dealing with programs which use basic data types and user defined data types. Basic data types, such as set, sequence, queue and stack, and operations on these types are defined or enhanced.
- **Constructs for Representing Entity Relationship Diagrams**
  These constructs are used to represent data designs of programs in Entity Relationship Diagrams at the conceptual level.
- **Constructs for Implementing Supporting Tools**
  These constructs belong to Meta-WSL, which is a subset of WSL dedicated to implementing program transformations and supporting tools, such as the Program Structure Database and the Metric Facility.

The main limitation of the existing WSL in the prototype of the Maintainer's Assistant is in the representation of COBOL records. Most COBOL statements can be translated into existing WSL statements. Another requirement for extending WSL is to represent Entity Relationship Diagrams. Therefore, the existing WSL was extended. A few examples are given to show how we represent COBOL records and Entity Relationship Diagrams in extended WSL.

Example 1: The following is a segment of a COBOL program:

```
01  STUDENT-INFO.
    05  STUDENT-ID-NO            PIC 9(8).
```

```
    05  NAME.
        10 FIRST-NAME           PIC X(9).
        10 LAST-NAME            PIC X(11).
    05  ADDRESS.
        10 STREET               PIC X(15).
        10 CITY                 PIC X(10).
        10 COUNTY               PIC X(15).
        10 POSTAL-CODE          PIC X(8).
    05  PHONE                   PIC 9(10).
```

It can be translated into WSL:

```
record student-info with
   field student-id-no [8 of int]
   subrecord name with
      field first-name [9 of char]
      field last-name [11 of char]
   end
   subrecord address with
      field street [15 of char]
      field city [10 of char]
      field county [15 of char]
      field postal-code [8 of char]
   end
   field phone [10 of int]
end;
```

Example 2: The corresponding diagrams are shown in Fig. 4 for the following examples of Entity Relationship diagrams in external WSL format:

```
(A)
entity E1 with
    attr A1
    attr A2
    attr A3
    attr A4
end;

(B)
entity E2 end

(C)
paragraph
   entity E3 end
   entity E4 end
   relationship entity E3 has one
      R1 relation with
      many entity E4;
end;
```

```
(D)
paragraph
   entity E5 end;
   entity E6 end;
   relationship entity E5 has one
      R2 relation with
      one entity E6;
end;

(E)
paragraph
   entity E7 with
      attr A5
      attr A6
   end;
   relationship entity E7 has one
      R3 relation with
      one entity E7;
end;

(F)
paragraph
   entity E8 end;
   entity E9 end;
   entity E10 end;
   relationship entity E8 has one
      R4 relation with
      {many entity E9} or {many entity E10};
end;
```

3.4 Extension of the Transformation Library

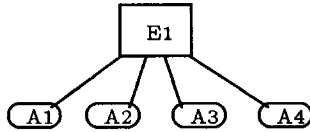
Existing WSL constructs and program transformations can be directly used in conjunction with newly defined WSL constructs and newly developed transformations.

Transformations developed for extension of the Transformation Library are divided into six categories.

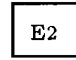
3.4.1 Basic data structures and ER components

It should be noted that some COBOL constructs are irrelevant to extracting ER models and, hence are ignored; these include DISPLAY, REMOVE, INITIALIZE, PERFORM, IF, WHEN, UNTIL, GOTO, SEARCH, and ACCEPT. Initializing of assignment statements is also ignored. In our tool, this is done during translation from COBOL to WSL.

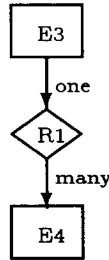
Transformations developed for extracting ER models in our tool are divided into six categories, and each category will be discussed in each of the following subsections.



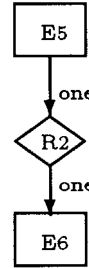
(A) An Entity with Attributes



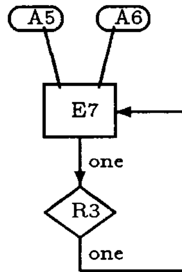
(B) An Entity without Attributes



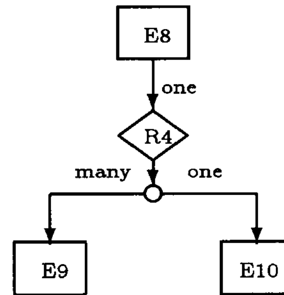
(C) A One-to-Many Relationship



(D) A One-to-One Relationship



(E) A Recursive Relationship



(F) A Mutually Exclusive Relationship

Fig. 4. Entities and relationships in WSL.

From Record to Entity

In forward engineering, a "01 level" COBOL record is usually used to implement an entity in an ER model. Therefore, in reverse engineering, a record with just one level of subrecord (syntactically a subrecord in WSL is still denoted by the keyword **record** for the convenience of defining a sub-sub-record, etc.) can be abstracted to an entity with attributes. For example, a record and an entity abstracted from that record are illustrated below. When the record is transformed into the entity, information such as the length and type of each field is thrown away because this information was not usually in the original data design and was added in by the implementor, e.g:

```
record E1 with
  record A1 [n of char]
  record A2 [n of char]
  record A3 [n of char]
  record A4 [n of char]
end;
```

is abstracted to:

```
 entity E1 with
  attr A1
  attr A2
  attr A3
  attr A4
 end;
```

Acquiring a Relationship from a Record with Subrecords

When a subrecord of a record can be abstracted into an entity and the record 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. For example, in the given record *author*,

```
record author with
 record name [40 of char]
 record address [50 of char]
 record book with
   record title [50 of char]
   record ISBN [20 of char]
 end
end;
```

the subrecord *book* can be abstracted into an entity while the record *author* can be extracted into an entity *author,* according to the knowledge in forward engineering. At the same time, a relationship "write" is abstracted from the logical connection between the record and the subrecord, i.e.:

```
paragraph
entity author with
   attr name
   attr address
end;
entity book with
   attr title
   attr ISBN
end;
relationship entity author has one write relation
     with many entity book;
end;
```

Note that information on the implementation details is also thrown away in the process of abstraction. An ER model for this example is shown in Fig. 5.

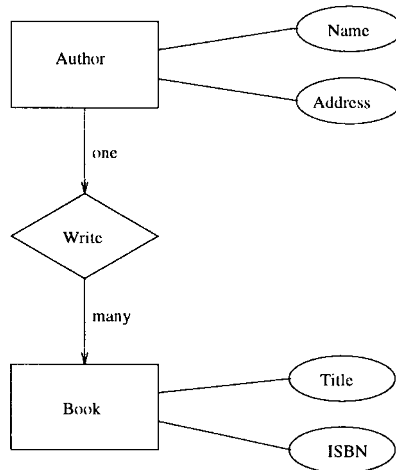


Fig. 5. Deriving a relationship from a record with subrecords.

3.4.2 Files

File input/output is a central problem in most data-intensive programs. File operations in a programming language usually involve access to external storage. For example, in COBOL, a serial file is a sequence of records; i.e., a record is the unit with which a physical file can be accessed. 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 representation 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.

3.4.3 Aliases

There are two purposes in using aliases. The first purpose is to share storage space and the second purpose is to control the storage scope. In languages, which do not have block structure, there is no way to use scope to control storage allocation, so the only technique to use aliasing. Aliasing is, therefore, frequently found in old programs and poses great difficulties for reverse engineering.

As discussed in [15], 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 a mapping the COBOL data structure to a low level memory model and a define function to describe a READ from an aliased record as a mapping in the reverse direction.

For instance, suppose we have the COBOL program shown below:

```
01 X                  PIC X(8).
01 Y REDEFINES X.     PIC X(8).
01 Z                  PIC X(8).
... ...
MOVE Z TO X.
```

It will be translated into WSL:

record *x* [8 **of** *char*];
record *y* [8 **of** *char*];
redefine *x* **with** *y;*
record *z* [8 **of** *char*];
... ...
x := *z*;
y := *read-rec*(*y*; *write-rec*(*x*; *z*));

Here, the WSL struct **redefine** declares that *x* and *y* are two aliased records. The function *write-rec* maps a change of *x* when assigned by *z* to the low level memory model of *x* and *y*; the function *read-rec* describes how the variable *y* is affected when its aliased record *x* is written. The second statement in the above program can be simplified by transformation, i.e.:

y := *read-rec*(*y*; *write-rec*(*x*; *z*));
becomes
y := *z*;

3.4.4 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 means of 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 between two pairs of entity attributes (e.g. assignment statements):

paragraph
entity *employee*
  attr *nhs-number*
  attr *name*
  attr *department*

```
        attr vehicle-num-plate
    end;
    entity car
        attr reg-number
        attr manufacturer
        attr model
        attr driver
    end;
    relate one employee.vehicle-num-plate to
            many car.reg-number;
    relate one car.driver to
            many employee.nhs-number;
    end;
```

Program transformations analyse code as well as data. For example, before these entities and relations were obtained, *employee* was a record variable and *employee:vehicle-num-plate* a field variable. The first **relate** statement records that variable *employee:vehicle-num-plate* ("*one*" variable) was assigned to by an expression typed *car:reg-number* more than once ("*many*" times), and this is later used in deriving a relationship. From the stage, as shown above, it can be identified that the first entity has a foreign key, *employee: vehicle-num-plate,* and that the second entity has a entity *car:driver*. If we start with the entity employee, a one-to-many relationship is derived, i.e., that a person can drive more than one car. When we start with the entity car, a one-to-many relationship is also derived, i.e., that one car can be driven by more than one person. Therefore, the WSL presentation and the Entity Relationship diagram are as shown below and in Fig. 6, respectively:

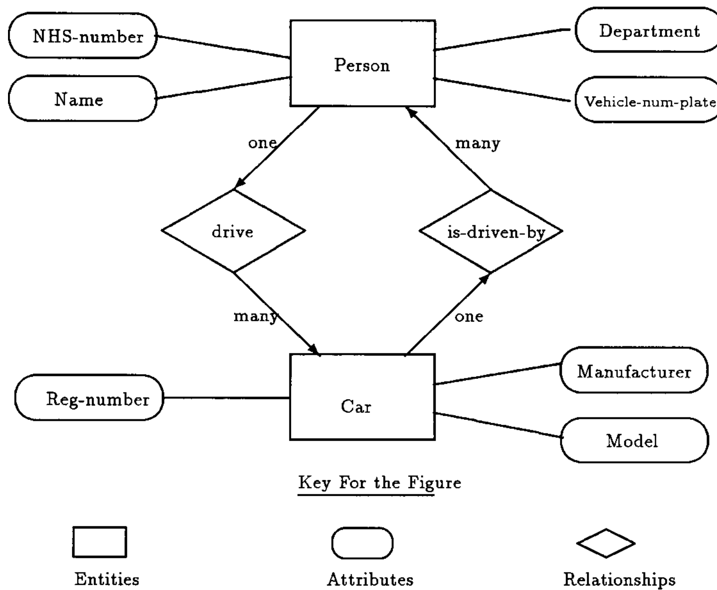


Fig. 6. Deriving a relationship from a foreign key.

```
paragraph
entity employee
      attr nhs-number
      attr name
      attr department
      attr vehicle-num-plate
end;
entity car
      attr reg-number
      attr manufacturer
      attr model
      attr driver
end;
relationship entity employee has one
         drive relation with
         many entity car;
relationship entity car has one
         driven-by relation with
         many entity employee;
end;
```

3.4.5 Abstract data types

Transformations for Forming Abstract Data Type

Transformations in this category deal with recognizing an abstract data type from constituent data declarations and operations on them. 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). Whether or not this closure was originally used for an abstract data type, or the abstract data type was constructed, if an abstract data type is obtained from the closure in the code, it is helpful in viewing the code at a higher abstraction level.

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). Whether or not a closure was originally used for an abstract data type, if an abstract data type is obtained from this closure in the code, it is helpful in viewing the code at a higher abstraction level.

The way to implement this idea is to first provide a structure in WSL. Five constructs are defined for the definition of a user-defined data type, user-defined data type procedure call and user-defined data type function call. The key words for these constructs are: **user-adt**, **user-adt-funct**, **user-adt-proc**, **user-adt-funct-call** and **user-adt-proc-call**.

Transformations in this class address looking for an abstract data type, i.e., a closure. A search for a closure can be started with a procedure definition (or function definition) as well as with a variable, and the result is the same.

It should be pointed out that the above method for identifying a user-defined abstract data type can be implemented satisfactorily in the program transformation approach because the program transformer is a powerful analyzer for searching for a closure. Other methods of identifying an abstract data type, such as using a *retrieve function*, though they have been considered, still need further study before they can be implemented in the tool.

From User-Defined Data Types to Data Design

An abstract data type usually involves a data object and a number of operations on this object. The operations are implemented in terms of procedures and functions, which take parameters. At the data design level, the data object can be viewed as an entity; each parameter can be viewed as a different entity; and the operations can be viewed as relations between the data object and other data objects that are represented by the parameters.

Therefore, a user-defined abstract data type can be abstracted to an entity while all the statements accessing this abstract data type have to be changed accordingly, and "ADT-> Entity" is such a transformation. By applying this transformation, the following program will be abstracted from

```
var ...
    begin
      ......
      user-adt-proc-call intset.insert (int1);
      user-adt-proc-call intset.idelete (int2);
      ......
    where
      user-adt intset (iset := 0; x := 0; y := 0) (nil)
        user-adt-proc iinsert (x)
          == iset := iset v{x};
        user-adt-proc idelete (y)
          = = iset −{y};
      ......
to
paragraph
entity int1
entity int2
entity intset
......
relate int1 to intset;
comment: insert int1 to intset;
relate int2 to intset;
comment: delete int1 from intset;
......
end
```

In this process, the abstract data type *intset* becomes an entity, and so do the two parameters involved. Two procedure call statements become two relations. In order to abstract the program further, useful information is recorded using comment statements.

Acquiring ER Models from Abstract Data Type

We shall continue with the example in the previous subsection. A relate statement has two components, one of which is an entity derived from an abstract data type. Then, the other component is a variable or a record without any subrecord (these two are equivalent, and transformations are available in the prototype for transforming one into the other), so this **relate** statement can be transformed into a relationship. The name of the relationship has to be provided by the user according to the information existing in the code. The program can be transformed into the following WSL form:

```
paragraph
entity intset;
entity int1
entity int2
......
relationship entity int1
      has one is-member-of relation
         with one entity intset;
relationship entity int2
      has one is-not-member-of relation
         with one entity intset;
......
end
```

3.4.6 Functional relationships

Transformations in this class address how ER models are extracted from code, in particular, from assignment statements, branching structures and loop statements.

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 usually be abstracted to a relation.

A branching statement, such as **if .. then .. else .. fi**, is used as a control structure in implementing programs. But the structure itself did not appear in the original data design and neither did the condition part of the branching structure. Therefore, these parts will not contribute to the Entity Relationship diagram. Information appearing in both arms of an **if .. then .. else .. fi** statement may exist in the Entity Relationship diagram. Therefore, an **if .. then .. else .. fi** statement can be abstracted to a sequence of two groups of statements (each group comes from each arm of the **if ... fi** statement).

Looping statements, such as **while** and **for**, are also used as a control structure in implementing programs but do not appear in the original data design. A looping statement can be treated to enumerate operation on every instance of an entities. The condition part of the loop also does not contribute to the Entity Relationship diagram. Therefore, a **while** loop or a **for** loop can be removed just leaving the body of the loop.

4. CASE STUDIES

4.1 Introduction to the Case Studies

The extensions to MA described in the previous section 3.2 were all implemented, and a number of case studies were used in experiments with our tool. For example, in one case study, the process of transforming a COBOL program, which copies one file to another, into an Entity Relationship Diagram was tried. This case study shows how we obtain ER models by analyzing COBOL statements in the "Procedure Division". It provides a basic test-bed for our ideas. In a second case study, a program where an alias was used to control storage scope is dealt with. In a third case study, a real program (of 3750 lines in COBOL) used in a national telephone company was thoroughly investigated. In a fourth case study, the COBOL program used was also from the same telephone company (the example code used in this case was approximately 7000 lines of COBOL source), and in a fifth case study, a section of a COBOL program of a Public Library Administration System, which involved using embedded system calls (CICS/TOTAL calls), was also used.

The intention in selecting these case studies was to cover as many applications of the method developed (or, in terms of the tool, as many transformations) as possible and, in particular, to find out what really happens to a real program at a scale of a few thousand lines.

4.2 Problems Addressed

In the subsection, general problems encountered during the case studies are described. Due to the limited length of this paper, quantified details are omitted.

4.2.1 Modulising the program by inspection

The first step in the analysis of the program is to inspect the code and divide it into smaller manageable blocks. Clues to doing this can be obtained first from the division information in the COBOL code.

Translating file and record declarations into WSL is straightforward; when they become WSL records, they are considered as candidates for entities. Not all WSL records are loaded into the prototype at the same time. A record is only loaded in conjunction with a block of code in which the record will be used.

The initialisation section normally opens files declared earlier and initialise data parameters and initialises input and output devices and have therefore not been dealt with.

The main processing section is usually composed of a number of functionally independent subprograms. One of these subprograms acts as the "main" subprogram which coordinates the remaining subprograms. These subprograms were translated into WSL, but at each stage, one subprogram was loaded into the prototype tool together with the records (which were translated earlier) used by the subprogram.

4.2.2 Initial "tidy-up"

When translating a COBOL program into WSL, some of the "data" statements which are not able to contribute to the eventual ER diagrams are omitted. For example, COBOL statements, such as MOVE 0 TO A-VARIABLE, MOVE SPACES TO A-VARIABLE,

MOVE "NOTHING" TO A-VARIABLE, ADD 0 TO A-VARIABLE, SUBTRACT 0 FROM A-VARIABLE, and SET AVARIABLE TO 0, INSPECT statements, COMPUTE statements, etc., are omitted. This represents information not needed in abstraction.

After initial tidying-up, the COBOL program translated into WSL was ready for further transformation. It is worth mentioning that information that could be useful in future abstraction was recorded by means of WSL comment statements.

4.2.3 Obtaining *relate* statements

Almost all the assignment statements that were originally translated from the MOVE statements in COBOL were abstracted to relate statements, and 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 would be used to decide the degree of the relationship between the two entities which would be obtained from the two records linked by the relate statement later on.

4.2.4 Aliased records

The REDEFINES statements in the original COBOL program were translated into **redefine** statements in WSL. According to observations, all the original records to be redefined were of the same data types as that of the redefining records. The conclusion about how to apply those functions to deal 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 in this case study.

4.2.5 Obtaining entities

Entities were abstracted from records, and this was the starting point for moving from the code level to the conceptual level. This was done when restructuring work at the code level had been completed.

4.2.6 Obtaining relationships

Relationships were mainly derived from the **relate** statements, and information recorded by the **comment** statements was used to decide the degree for each relationship by means of transformation with the help of human expertise.

The **comment** statement was translated from the OCCURS construct in the COBOL program and was used to decide the degree of the relationship.

4.2.7 Final tidying-up and result

Finally, duplicate entity relationships which might be obtained from different places of the program, were checked and removed. All the **comment** statements were removed. Usually:

- One line of COBOL code is typically translated into one line of WSL.
- A number of entities and relationships were successfully derived from the code in each case study. Some entities were extracted from the original 01 level records, and other entities were derived from file operations and from abstract data types; relationships were derived from those records which had subrecords, abstract data types and foreign keys).
- Data designs obtained from programs made these programs much more comprehensible.
- Deriving data designs using the tool was faster and more reliable process than doing it manually (even when that is possible).
- Metric graphs monitoring the process of transforming programs showed that the programs had been considerably simplified, i.e., by becoming more abstract.

4.3 Summary

The case study described above is a typical example of existing data intensive code which may need reverse engineering. Other case studies showed a situation similar to that in this case study where a certain number of ER models were extracted. Through these case studies, we observed, in particular, a few facts:

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

Information obtained during the reverse engineering process needs to be used carefully. For example, relationships obtained, such as "file-backup" in one case study, emerged from reverse engineering but were not really a part of the application domain. Such information must be discarded, which requires that a decision be made by the reverse engineer.

In one of the case studies, embedded CICS/TOTAL calls were included. Fortunately, most of the CICS/TOTAL calls in this COBOL source code were used to handle simple "interfacing" operations, so that they could be translated into external procedure (function) calls in WSL (these are WSL functions to call external procedures (functions) which it is known definitely which variables would be changed) for further processing. However, if these CICS/TOTAL calls had not been comprehensible, they could not have been processed since the source code of these CICS calls did not literally appear in the COBOL program.

When the source code scales up, the method developed in the paper would still work well provided that program segments of a manageable size can still be found. The scale of the source code will affect the method in a situation where smaller program segments are not all comparatively self-contained; i.e., self-contained segments are not of manageable size. 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.

5. CONCLUSIONS

The research described in this paper has focused on extracting ER models from data intensive source code by combined analysis of data structures and code. Suitable formal transformations for this purpose have been developed, in particular, transformations for dealing with COBOL records and files, aliased records, foreign keys, user-defined abstract data types and functional relationships. These transformations have been used in the case studies. This research has increased our understanding in this field, and the main problems of extracting ER models have been solved.

One of the features of this approach is the extraction of ER models to represent the final products (data designs). The existing WSL has been extended to represent the original programs and ER diagrams. Therefore, the original objects and the target objects can both be represented and manipulated in the same language. In order to capture useful information for data abstraction, all constructs associated with data are used, i.e., records, assignment statements, etc. Common data structures used in programming, data-intensive programming in particular, have been investigated, for example, foreign keys, aliasing problem, file operations, ADTs, etc. However, we have not found examples, in practice, of aliased records incorporating type conversion. This suggests that our underlying memory model, at least for COBOL, is excessively complicated, and that the great majority of cases could be handled by means of static analysis of the code.

It has been demonstrated that it is viable to extract ER models from code and data structures using the method developed. ER diagrams have been produced from several tens of COBOL programs. The development of the method and the implementation of the prototype show that this approach has covered a broad range from theory to practice [2]. We have not found other systems that can derive data designs represented by ER models in this way [8].

It appears at first that the method developed here could be applied manually (i.e., by manipulating COBOL code by hand and extracting entities and their relationships from the code directly) and therefore, that it might not necessary to build a tool to implement the method. This argument might be true when a program is fairly small (like the program in the first case), but when the size of the program to be maintained increases, building a tool becomes essential.

Industrial case studies, such as the ones described in this paper, were carried out during the research. With the help of the method developed and the tool built in this research, practical reverse engineering could be achieved.

A number of conclusions can be reached from this work to extend the original prototype of the Maintainer's Assistant:

- The abstracted 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. During the duration of this project, experts in COBOL programming were consulted to identify the features of data-intensive programs and to evaluate the resultant Entity Relationship diagrams obtained from the example programs using the prototype tool. The experts agreed that the obtained Entity Relationship Diagrams represented viable designs for these example programs and could be used as possible data designs for subsequent for-

ward engineering of the programs. In some cases, this could be done in an inverse way, i.e., by checking whether the original program was an implementation of the obtained Entity Relationship Diagram. Of course, the eventual aim would be to assure correctness by proving that the transformations are semantic preserving. This has been done for some transformations.

- The method combines analysis of data and of code; therefore, it can address two major problems common to data-intensive programs, i.e., aliases and relations (like foreign keys) that affect both data and code. This distinguishes our approach from others, e.g., Bachman's work.
- The method is sufficiently general to apply to data intensive programs written in other source languages though only COBOL programs have been used in experiments to date. For example, a program written in the C programming language can be reverse engineered using the approach as long as the C program is translated into WSL. In fact, a few examples in the experiments were derived from C programs, using manual translation to WSL.
- This method only requires source code as its input and it can be applied to heavily modified code of the sort which is typical in systems which have been maintained over many years.
- There are few restrictions on the approach developed in this research. Perhaps the only restriction is that the user needs to supply the source code which is to be reverse engineered. The approach is, however, fundamentally interactive, and the tool supports the expert maintainer in restructuring code and then extracting designs. It is not automatic. The user-driven mode is in common with the approaches used in almost all other reverse engineering methods. The user's role was taken into consideration when the approach was studied, and a user-friendly interface was designed and implemented when the prototype tool was built.

Further research is needed to solve the remaining side problems of extracting ER models from source code. For example, more experiments should 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. The method developed in this research is general enough to cope with data-intensive programs in other languages. However, at present, the prototype is mainly COBOL-specific, and it can only deal with data structures that exist in COBOL. For example, programming languages such as C provide data structures such as pointers that are not supported by COBOL; therefore, transformations for dealing with pointers are still needed. Another area for future research is to find out whether the approach developed in this research can be used to extract specifications (e.g., specifications written in Z) from programs, which was the original aim of the REFORM project.

Metric measures need to be defined for measuring both codes and specifications to meet the needs of reverse engineering. It is perhaps important that a metric measure can reflect the process of crossing levels of data abstraction.

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

ACKNOWLEDGEMENTS

The authors thank the referees for their valuagle comments. This work was support in part by the National Science Council of the Republic of China under grant NSC87-2213-E-035-009.

REFERENCES

1. R. Bachman, *A CASE for Reverse Engineering*, reprinted from DATAMATION, Cahners Publishing Company, 1988.
2. K. Bennett, T. Bull and H. Yang, “A transformation system for maintenance — turning theory into practice”, in *Proceedings of IEEE Conference on Software Maintenance* 1992, pp. 146-155.
3. T. Bull, “An introduction to the WSL program transformer”, in *Proceedings of IEEE Conference on Software Maintenance* 1990, pp. 242-250.
4. F. Calliss, M. Ward and M. Munro, “The maintainer’s assistant”, in *Proceedings of IEEE Conference on Software Maintenance* 1989, pp. 168-177.
5. G. Canfora, A. Cimitile and M. Munro, “A reverse engineering method for identifying reusable abstract data type”, Technical Report, DUR-CS-9208, Department of Computer Science, Durham University, 1992.
6. P. Chen, “The entity-relationship model — toward a unified view of data”, *ACM Transaction on Database Systems*, Vol. 1, No. 1, 1976, pp. 1-16.
7. Cutts, *Structured Systems Analysis and Design Methodology*, Paradigm Publishing Company, London, 1987.
8. C. Date, *An Introduction to Database Systems*, Vol I, Addison-Wesley Publishing Company, Manchester, 1986.
9. K. Lano and P. Breuer, “Reverse-engineering and validating COBOL”, Technical Report, 2487-TN-PRG-1049, Programming Research Group, Oxford University, 1991.
10. J. Martin and C. McClure, *Structured Techniques for Computing,* Prentice-Hall International Inc., Englewood Cliffs, 1985.
11. H. Sneed and G. Jandrasics, “Inverse transformation of software from code to specification”, in *Proceedings of IEEE Conference on Software Maintenance* 1988, pp. 126-135.
12. M. Ward, “Proving program refinements and transformations”, Ph.D. Thesis, Department of Mathematics, Oxford University, 1989.
13. H. Yang, “Acquiring data designs from existing data intensive programs”, Ph.D. Thesis, Computer Science Department, Durham University, 1994.
14. H. Yang, “The supporting environment for a reverse egineering system — the maintainer’s assistant”, in *Proceedings of IEEE Conference on Software Maintenance* October, 1991, pp. 13-22.
15. H. Yang and K. Bennett, “Extension of a transformation system for maintenance — dealing with data-intensive programs”, in *Proceedings of IEEE International Conference on Software Maintenance* , 1994, pp. 344-353.

Hongji Yang received his B.Sc. (from Jilin University, China), M.Phil. (from Jilin University, China) and Ph.D. (from Durham University, England) in computer science in 1982, 1985 and 1995, respectively. He was an assistant professor from 1985 to 1987 and a lecturer from 1987 to 1988 at Jilin University, China; he was a senior research assistant from 1989 to 1993 at Durham University, England; and he was a lecturer from 1993 to 1994, and a senior lecturer from 1994 to 1996 at De Montfort University, England. His research interests include software engineering, computer networks and computer architecture.

William Cheng-Chung Chu (¦¶¥Ç©¾) received his M. S. and Ph. D. degrees from Nothwestern University, Evanston, Illinois, in 1987 and 1989, respectively, both in computer science. Since 1998, he has been with the Department of Computer and Information Science at the TungHai University, Taiwan, where he is now an Associate Professor. From 1994 to 1998, he had been with the Department of Information Engineering at Feng Chia University, he was a research scientist at the Software Technology Center of the Palo Alto Research Laboratories of the Lockheed Missiles and Space Company, Inc., where served as a member of the Software Reverse Engineering and Software Reuse projects team and was a principal investigator of Converting Programming Languages for the Ada project. He received special contribution awards from Lockheed in 1992 and 1993. In 1992, he was a Visiting Scholar in the Department of Engineering Economic System at Standford University, where he was involved in projects related to the design and development of an Intelligent Knowledge Based Expert System.

His current research interests include the fields of software re-engineering, maintenance reuse, software quality, OO, and distributed computing systems.