Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Reusable Component Identification by Code Understanding through Program Transformation in RRRA

Hongji Yang
Computer Science Department
De Montfort University
England
Email: hjy@dmu.ac.uk

William C. Chu
Department of Information Engineering
Feng Chia University
Taiwan
Email: chu@fcu.edu.tw

Abstract

Code understanding is almost the most essential step in all the post-delivery software activities such as software maintenance, re-engineering and reuse. In Reverse-engineering Reuse Re-engineering Assistant(RRRA), a tool aiming at providing an overall approach for all the post-delivery software activities, code understanding was addressed by reverse engineering through program transformation. The paper proposes a method to deal with this problem and discusses in detail how program transformation techniques, program comprehension techniques and the role of human knowledge are integrated into RRRA, i.e., how they are used by reverse engineering to recognise reusable components and how they are used by semantic interface analysis to formally represent reusable components. The experiments strongly suggest the proposed method is a practical approach to software reuse.

1 Introduction

Code understanding is a question that has to be answered when responding software evolution challenge such as software maintenance, re-engineering and reuse. An obvious reason is that one can not maintain or reuse a piece of software before being absolutely sure about the functionality of the software. For legacy systems, large legacy systems in particular, code understanding is a very difficult task. The only helpful source of information about a legacy system is perhaps the documentation, which is almost certainly missing or incomplete (therefore unreliable) after continuous maintenance of many years. If it is the case, considerable effort is essential in understanding a multi-million line program.

A basic activity to understand existing code is to gather information. When the documentation of a software system became unreliable, the sole source of information from the software is the code itself. This means that code understanding is typically a code activity. Apart from the information directly from the software, other sources of information can also help with code understanding. These sources are domain knowledge of the application, software development and programming knowledge.

One way of understanding code is through reverse engineering, i.e., to reverse code in low level of abstraction to a high-level presentation. Code understanding is eventually a human-centred activities and cognitive research into expert/novice performance differences has indicated that human performance in cognition tasks are significantly better if higher-level abstractions of a problem representation can be exploited.

Program transformation is powerful tool for reverse engineering and it will bring several advantages to code understanding, e.g.:

* Data structures and the implementation of abstract data types can be changed easily.

* Code restructuring changes can be made to the program with the confidence that the functionality is unchanged.

* Formal links between high-level representation (specification or design) of the code and code can be maintained.

* Understanding of code can be carried out at the higher abstraction level.

* Use of code can be incrementally improved — instead of being incrementally degraded.

Code understanding can provide a great help to software reuse, in particular identifying reusable software components. Only a potentially reusable component is understood thoroughly, can the component be confidently reused. In RRRA, a tool aiming at providing an overall approach for all the post-delivery software activities, program understanding was addressed by reverse engineering through program transformation. The approach used by RRRA works like this: a potential reusable code component is reversed into its high-level representation through program transformation; the component is "understood" and identified by being assigned with pre-conditions, post-conditions and obligations; the component will be stored in a reuse library if identified reusable; and finally the component can be reused when its properties will meet the requirements predicates proposed.

There are two basic technical approaches to reuse: parts-based and formal language-based. 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 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.

This paper will first briefly review relevant program understanding approaches. Then the environment of RRRA will be described. Thirdly, the approach of identification of reusable component by reverse engineering through program transformation will be discussed. And finally conclusions will be given to summarise our study.

2 Problems of Code Understanding Related to Our Study

The study described in this paper used a program transformation approach to reverse engineer existing code in order to recognise reusable components in the program transformation environment. In this section, previous work that influences our study is discussed briefly and then the working environment is introduced.

2.1 Code Understanding Theory

To obtain a model of acquiring a high-level abstract representation from code, program understanding techniques, cognitive models and personal experience are decisive factors.

Approaches to program comprehension are summarised by [11], in which three program comprehension problems are discussed.

- Theories of program comprehension:
  1. examining of the entire program and working out the interactions between various modules,
  2. understanding the program by syntactic and semantic knowledge,
  3. setting a hypothesis of a mapping between the problem domain and the programming domain,
  4. using both top-down and bottom-up strategies at the same time.
- Code reading: The crudest method of understanding program is code reading. Factors affecting code reading are:
  1. the design method employed in the implementation of the program,
  2. the style of writing the program, for example, using meaningful variable names, indentations, comments, etc.
- Program analysis: Static and dynamic analysis — to obtain useful information, such as cross reference listings, call graphs, slicing, and symbolic execution, etc.

Soloway and Eirlich claim [13] that expert programmers have and use two types of programming knowledge: **programming plans**, which are generic program fragments that represent stereotypic action sequences in programming, and **rules of programming discourse**, which capture the conventions in programming and govern the composition of the plans into programs.

Programming knowledge will also play a powerful role in program comprehension [8]. Usually, advanced programmers have strong expectations of what programs should look like and programming knowledge is the base of a programmer's expectation.

2.2 Code Understanding and Tools

Code understanding process is heavily dependent on both individuals and their specific cognitive abilities, and on the set of facilities provided by code understanding [14]. This suggests that tools are one crucial factor to help human to understand code. Though any successful practice of code understanding and software reuse does not require a tool, tools can simplify some tasks especially when tackling a large system. Tools can also be used to locate and identify reusable code, components, and class libraries [15].

It is well accepted that code understanding can be enhanced using reverse engineering technologies. Therefore a reverse engineering tool can be seen as a necessary part in a code understanding system. Also, if one can learn more about how programmers understand code successfully, one can build better tools to support the understanding process [10].

2.3 Data-centred versus Control-centred Code Understanding

People trying to tackle software evolution challenges use a variety of techniques and representations for understanding programs. Most of these representations first focus on the control structure of a program such as call graphs, control flow graphs and paths. This is called control-centred code understanding.

Data centred code understanding is another approach [9] for program understanding – it first focuses on data and data relationships. Whereas the control-based approach gives the control flow model directly and allows the data flow model to be derived, the data-centred approach gives the data flow model directly and allows the control flow model to be derived. In data-centred understanding, people usually first identify and classify important variables in a program, then find the dependence relationships among these variables by tracing the computation done between specific locations in the program. For example, programmers can gain an understanding of a program's variable simply by the names of the variables, e.g., account-name, account-number, etc. This way of program understanding can be supported by automatic variable classification, dependence analysis and program and variable slicing. In business applications, data and variables play a dominant role and the ability to quickly find dependence among variables without having to trace and retrace control flow paths saves considerable time and enhances program understanding.

2.4 Code Understanding and Role of Human Knowledge

Human knowledge also plays an important role throughout the whole process of component reuse in a program transformation system (which is usually an interactive system). This fact is both crucial to the researcher (tool builder) and customer (tool user).

To the reverse engineering researcher, the problem of how to accommodate the use of human knowledge in a tool has to be solved. In fact, the use of a program transformer covers the aspect of static program analysis. Other aspects include presenting useful information (e.g., comments in a program), providing the user with a facility for naming reusable components, etc.

2.5 RRRA — A System Used for Reuse

RRRA (Reverse-engineering Reuse Re-engineering Assistant) is a tool designed for covering aspects of reverse-engineering, resue and re-engineering, and it has addressed the reuse aspect in some extent. The reuse aspect in RRRA has been influenced by both authors' previous research work, semantic interface analysis and reverse engineering.

2.5.1 Semantic Interface Analysis and Reverse Engineering

Semantic interface analysis is a formal approach and it has been used in the development of a reuse tool called the Module Integration and Adaptation Tool for Ada Components (MI-ATAC) where semantic attributes of software components were described by formal notations. Since software reuse includes areas of concern such as representation, retrieval, and adaptation and integration [3] [5], MIATAC attempted to use formal semantic interface predicates in solving these three areas of concern. In particular, MIATAC focuses on the area of integration and adaptation.

In MIATAC, the reusable Ada components are resident in a reuse library and contain formal semantic interface specifications consisting of precondition, postcondition, and obligation predicates represented as specialised Ada 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 may consist of both newly developed components, and previously adapted reuse components all of which may contain formal semantic predicates. This "syntactically integrated" application system is then transformed by a parser which organises the Ada code and the annotated comments into a Descriptive Intermediate Attributed Notation for Ada (DIANA) Tree intermediate form [7]. The DIANA Tree is then traversed and analysed by predicate logic routines to determine how well the candidate component fits into the application system.

Another work which influenced the design of RRRA is the REFORM (Reverse Engineering using FORmal Methods) project [1] [2] [16]. The aim of the project was to build a prototype tool called the Maintainer's Assistant (MA) which would take existing software written in low-level procedural languages, through a process of successive transformation, and turn it into an equivalent high-level abstract specification expressed in terms of a non-procedural abstract specification language (for example, Z).

MA has both a firm theoretical foundation and a design which has evolved from case studies, which together provide beneficial results when used with real programs; its method combines analysis of data and code, and therefore it can address major problems common to programs written in many programming languages; and it only requires source code as its input and it can be applied to heavily modified code which have been maintained over many years.

2.5.2 The Prototype of RRRA

A main design philosophy of RRRA was to first build the Browser/Interface component (See Figure 1) and use it for integrating other tool components (especially that may exist already). The overall process is as follows:

RRRA takes the source code (in any language) and translates it into its equivalent RRRWSL (Reverse-engineering Reuse Re-engineering Wide Spectrum Language). A user uses the Browser Interface to control the whole tool, i.e., the Browser calls each tool component and displays result on the interface. The Modulariser will check the program, chop the program into smaller programs which are of manageable size, and save it in a database called Program Segments. Then, the user will take a segment of code from the database to work on. The Browser allows the user to look at and alter the code under strict conditions and the user can also select transformations to apply to the code. The Program Transformer works in an interactive mode. It presents RRRWSL on screen in a pretty printed format and searches a catalogue of proven transformations to find applicable transformations for any selected piece of code. Once a transformation is selected it is automatically applied. The code is then transformed to a form at a higher level of abstraction, such as Entity Relationship Diagram(ERD), Data Flow Diagram, Structure Chart, etc. The Object Extractor will use these high abstraction level forms to produce objects and save them in the database named the Objects. During this process formal links have been always kept between a segment of code and its high level abstraction form (e.g. ERD), and between a segment of code and its representation of object. The Semantic Interface Analyser will analyse information in the Program Segments, in the high abstraction level form (e.g., ERD) and in the Objects, give them formal attributes (i.e., they are annotated with predicates) and save them in the Reuse Library. When the re-engineering process starts, the Synthesiser uses the requirements in the New Design/Spec to make queries to the Reuse Library. The Synthesiser will make best use of the reusable components and invoke the Code Writer to generate the part of code which was required by the new system but was not available in the Reuse Library. The synthesised code will be saved in New Code but is still in RRRWSL. Finally, this code is translated into the target code in the required language.

Throughout the whole process the Knowledge Base plays an important role. The Knowledge Base has many functions. In this paper, we only emphasise its role for program transformations (to suggest transformations in a given situation) and predicate annotation and propagation (to provide knowledge information and abstraction rules).

3 Code Understanding in RRRA

3.1 A Method for Identifying Reusable Components

After carefully studying the existing state of the art in this field, a method was proposed for RRRA in identifying reusable components through program understanding. The method consists of the following steps:

1. Translating a program in a source language, such as in ADA or in COBOL, into RRRWSL.
2. Using initial tidy-up transformations to "clean up" the target program in RRRWSL 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 resuable 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 on each time. Program transformations are applied to the module to reverse the module into its high-level representation in Entity Relationship diagrams.
5. The obtained Entity Relationship diagrams together with the original code are used by a Semantic 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 roles 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.

3.2 Implementation of the Method

Problems during the process of implementing the method proposed above are discussed in the section.

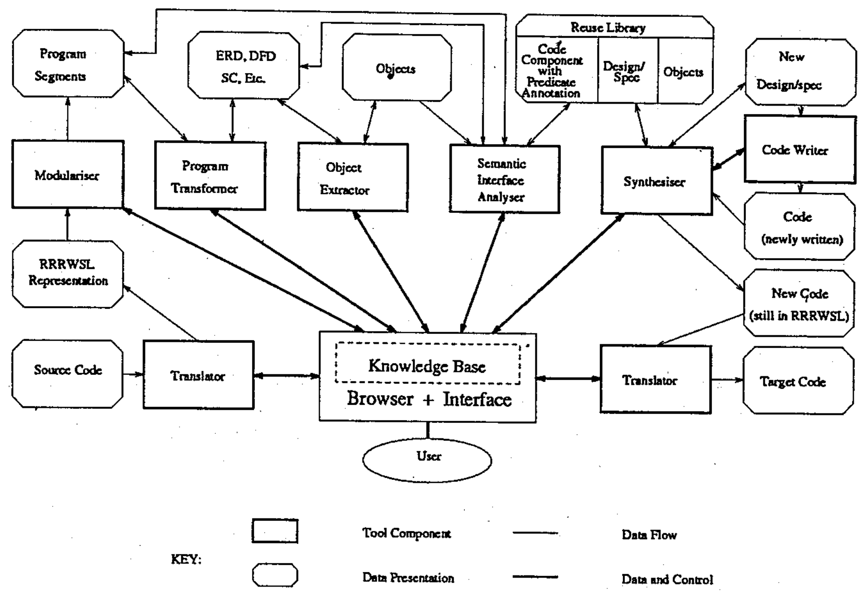

Figure 1: The Reverse-engineering Reuse Re-engineering Assistant (RRRA)

3.2.1 Use of Entity Relationship Model

To reverse engineer code to a high abstraction level representation is to make the original code easier to be understood. Therefore a model at high abstraction level is needed. Because, as discussed earlier, data centred code understanding has many advantages and the Entity Relationship model has been popularly used for many years, the model is used in RRRA.

Entity models provide a system view of the data structures and data relationships within the system [4] [6]. All systems possess an underlying generic entity model which remains fairly static in 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. They provide a clear view of the logical structure of data within the boundary of interest and allow the analyst to model the data without considering its physical form. Entity modelling provides a system view independent of current processing; it is a system-wide view not a functionally decomposed view.

3.2.2 Crossing Levels of Abstraction and Use of Program Transformation

Abstraction techniques are heavily used in reverse engineering. In reverse engineering, abstraction is the process of identifying the important qualities or properties of the phenomenon being modelled. They abstract from irrelevant details, describing only those details that are relevant to the problem at hand, e.g., understanding the design.

Usually, program code and its design are not at the same level of abstraction — the design is more abstract than the code. It is necessary to reduce the amount of complexity that must be considered at any one time, so that certain number of abstraction levels may exist during the specification extraction process. Each "layer" can be considered as a program in a language provided by a virtual computer, implemented by the layer below. At the lowest layer, we have a real machine.

Program transformations are a suitable tool for crossing levels of abstraction because proven transformations when applied will preserve semantics of code before transformed for the transformed code.

A Program Transformer is a good static analyser, because it checks a piece of code thoroughly to see whether the conditions of applying a transformation are met, and only when the conditions are met can a transformation be applied.

3.2.3 Use of Programming and Domain Knowledge

Transformations are design and proven by the tool designer or reverse engineering researcher according to the research results of reverse engineering. Well designed transformations should integrate all relevant knowledge, application domain knowledge, programming knowledge, etc.

Human knowledge assists in the following aspects:

1. Modularisation of source code. The first step in dealing with real software is to modularise the software into manageable sized modules which ought to be functionally independent.
2. Searching for and naming abstract data types. An abstract data type is an important concept of data abstraction. It is the user who guides a program transformation system in searching for an abstract data type and names the obtained abstract data type. The name of an abstract data type affects further abstraction from the abstract data type. Though tools can help in this case, e.g., work of Sneed [12], the role of human is decisive.
3. Searching and naming reusable components. In extracting reusable components from code, it is again the user who directs the search. This includes questions of where to look for, and how to name, reusable components.
4. Making use of any potentially useful information visible in the code, e.g., meaningful variable names, comments, indentation, procedure and function names, etc.
5. Making use of program syntax components, e.g., controlled variable of a loop, assignment statement, etc.

6. Allowing user's hypothesis made according to software engineering knowledge and domain knowledge. The user's hypothesis can be continuously updated all the time as the process of applying transformations is going on.
7. Providing help information from a tool, that may have a built-in manual facility for all the transformations available.

3.2.4 Combining Data and Structure Analysis

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

3.2.5 Abstract Data Type and Reusable Component

An abstract data type is usually a suitable candidate as a reusable component. 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). No matter whether 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 above method of identifying a user-defined abstract data type can be implemented satisfactorily in the program transformation approach because a program transformer is usually a powerful analyser in searching for a closure.

3.2.6 Keeping Formal Links between Reusable Components and Their High-Level Representation

The process of transforming a module at the code level to its high abstraction level representation may takes a number of steps. The Program Transformer has a built-in facility to keep a full history of the transformation process. The advantage of doing this is that when a route of abstraction fails the Program Transformer can "retreat" to a suitable point and move forward again. At the mean time, this facility maintains a formal link between the original module to every intermediate stage of the transformed module as well as the final version of the module.

3.2.7 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 is to use semantics of data structure *sequential-file* and a loop of record copy to infer that the module *file-backup*(*File*1, *File*2) 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 RRRA.

The Knowledge Base in RRRA 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.

3.3 An Illustration

The example program used in this illustration was taken from a COBOL text book 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 CLOSES THE FILES
   STOP RUN.
```

Table 1. A File-backup Program in COBOL

The program is translated into its equivalent form in RRRWSL (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 RRRA prototype.

The identification division in COBOL is translated into a comment statement in RRRWSL. 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 RRRWSL records and files. COBOL files in this example are sequentially organised and sequentially accessed.*

In this program original file operations are translated into RRRWSL as external procedures (denoted by !p which is the RRRWSL 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 RRRWSL 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 RRRWSL

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 form 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 RRRWSL

After applying transformations discussed in this section, the final result of the "file-backup" program can be shown by an Entity Relationship diagram (Figure 2).

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 RRRWSL (Table 2 and 3) will also be analysed by the Semantic Analyser(SA) in order to generate a form annotated with formal pre-conditions, post-conditions and obligations.

To demonstrate how SA 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
!Iteration-Index(i)
!EQU(A, array[1..N] of record-name)
!EQU(B, array[1..N] of record-name)
                                  ⟹ ∀ i EQU(A[i], B[i])
!LEQ(i, N)
post-condition
!Iteration-Index(i + 1)
!A[i] = B[i]
```

and

```
pre-condition
!Sequential-File(File1, record-type1)
!Sequential-File(File2, record-type2)
!EQU(record-type1, record-type2)
                                  ⟹ EQU(File1, File2)
post-condition
!∀ i EQU(file1[i], 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
!Sequential-File(source-file-name, source-record)
!EQU(source-file-name, array[1..N] of source-record)
  file source-file-name with
     record source-record with
     end;
  end;
post-condition
!Sequential-File(target-file-name, target-record)
!EQU(target-file-name, array[1..N] of target-record)
  file target-file-name with
     record target-record with
     end;
  end;
  eof := 0;
pre-condition
!Sequential-file(source-file-name, ?)
post-condition
!Index-of-Seq-File(1, source-file-name)
!Open-File(source-file-name)
  !p open_file (i var source-file-name);
pre-condition
!Sequential-file(target-file-name, ?)
post-condition
!Index-of-Seq-File(1, target-file-name)
!Open-File(target-file-name)
  !p open_file (o var target-file-name);
while(eof ≠ 1)do
pre-condition
!Open-File(source-file-name)
!Sequential-file(source-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
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-Seq-File(i, source-file-name)
         post-condition
         !EQU(source-record, source-file-name[i])
         !Index-of-Seq-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-Seq-File(i, target-file-name)
         post-condition
         !EQU(target-file-name[i], target-record)
         !Index-of-Seq-File(i + 1, target-file-name)
         !p write_file (target-record var target-file-name);
fi;
od;
```

Table 4. 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(File1, record-type1)
!Sequential-file(File2, record-type2)
!EQU(record-type1, record-type2)
post-condition
!EQU(File1, File2)

file-backup(File1, File2)
```

Table 5. Semantic Interface Predicates Generated

This example illustrates one of many experiments that have been carried out.

4 Discussions and Concluding Remarks

Identifying reusable components from existing code with the help of coding understand through program transformations is a practical approach. The use of formal program transformation will give users confidence about the reusable components identified. The use of semantic interface analysis will make the selection of a reusable component more accurate. And after all, the whole process is formal and therefore has been properly integrated into a tool. The approach used in the RRRA has several features:

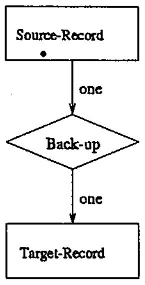

Figure 2: Entity Relationship Diagram for File-Backup Program

- Program understanding techniques have been fully used by the proposed method and integrated in building the tool. Main examples include:
  1. Code reading is used in modularising a program system.
  2. The use of Entity Relationship diagrams is based on the idea of code centred understanding.
  3. The use of program transformation represents the use of formal methods.
  4. Programming knowledge is used in many places, such as dealing with file operations, etc.
  5. Static analysis is used for checking applicability of transformation.
- Reverse engineering code into high level abstraction represented in Entity Relationship diagrams provides the user with a good chance to understand a program module existed in the old code and this will help the user to decide more confidently whether the module is reusable or not.
- In addition to helping the user to comprehend a program module formally, the proposed approach will still use a formal means to examine the module and generate formal semantic interface predicates for reuse.
- Experiments have demonstrated the feasibility of the method proposed in this paper. Since the method always starts with looking for manageable-sized code blocks, it can be applied to large programs, i.e., the method can scale up.

Example programs used in the experiments were mainly written in ADA, COBOL and C, and they were translated into RRRWSL. It is expected that more experiments will be carried out on programs written in other languages. Also, in the future work of the research in RRRA, the results of applying those reusable components identified by the approach described in the paper will give useful feedback on improving the process of identification of reusable components.

References

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

[2] Bennett K., Bull T. and Yang H., "A Transformation System for Maintenance — Turning Theory into Practice", IEEE Conference of Software Maintenance-1992 (CSM '92), Orlando, Florida, November, 1992.

[3] Biggerstaff T. and Ritcher C., "Reusability Framework, Assessments and Direction", IEEE Software, Vol.4, No. 7, pp. 252-257, 1987.

[4] Chen P. P., "The Entity Relationship Model — Toward a Unified View of Data", ACM Transaction on Database Systems, Vol. 7, No. 1, March, 1976.

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

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

[7] Evans A., Bulter K., Goos G. and Wulf W., *DIANA Reference Manual*, Tartan Laboratories, Inc., Pittsburgh, 3rd Edition, 1983.

[8] Gilmore D. J., "Models of Debugging", Fifth European Conference on Cognitive Ergonomics, September, 1990, Urbino, Italy.

[9] Joiner J.K., Tsai W. T., Chen X. P. Subramanian S. and Gandamaneni H., "Data-Centred Program Understanding", IEEE International Conference on Software Maintenance-1994, Victoria, Canada, 1994.

[10] von Mayrhauser A. and Vans A. M., "From Code Understanding Needs to Reverse Engineering Tool Capacities", Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering (CASE '93), Singapore, July, 1993.

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

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

[13] Soloway E. and Ehrlich K., "Empirical Studies of Programming Knowledge", *IEEE Transaction on Software Engineering*, SE-10, September 1984.

[14] Tilley S. R., Wong K., Storey M. D. and Muller H. A., "Programmable Reverse Engineering", IEEE International Conference on Software Maintenance-1994 (ICSM '94), Victoria, British Columbia, Canada, September, 1994.

[15] Wappler T. and Yglesias K. P., "What A Reuse Tool Can Do for You", Object Magazine, Vol. 4, No. 8, Jan., 1995..

[16] Yang H., "The Supporting Environment for A Reverse Engineering System — The Maintainer's Assistant", IEEE Conference on Software Maintenance-1991 (CSM '91), Sorrento, Italy, October, 1991.