A Reengineering Assistant and Case Studies

HE Zheng-chu[1, 2]

(1. Department of Business Studies, Hunan College of Finance and Economics, Changsha 410205, China;

2. Post Doctoral Station, Chinese Academy of Social Sciences, Beijing 100732, China)

Abstract: Much of the work in software maintaining and reengineering has concentrated on source transformation, which changes the program source code text or specification source text from one form to another so that these source texts can become more readable and understandable. In this paper we describe our experiences in this area, a convenient tool which can be used to abstract a specification from a program (i.e. from source code text to specification source text) will be introduced, at the same time, a brief introduction of Wide Spectrum Language (WSL) will be given since it will be used to represent our on-screen program in Reengineering Assistant (RA). Additionally, a case study with our RA will be given.

Key words: reengineering assistant; reverse engineering; source to source transformation; program transformation; program understanding; Wide Spectrum Language.

1. Introduction

A program transformation is an operation which modifies a program into a different form which has the same external behavior (it is equivalent under a precisely defined denotational semantics). Many tasks in software engineering and maintenance can be characterized as source to source transformations. Reverse engineering or design recovery [1-2] can be cast as a source transformation from the text of the original source code files to the text of a set of design facts. Software reengineering and restructuring [3] can be cast as a source transformation from the poorly structured original source code text to a better structured new source code. Forward engineering or metaprogramming [4], can be cast as a transformation from the source text of design documents and templates to the instantiated source code files. Platform translation and migration tasks are easily understood as transformations from the original source code files to new source code files in the new language or paradigm. And code reuse tasks can be implemented as a source transformation from existing, tested source code to generic reusable source code modules. While many other methods can be applied to various parts of these problems, at some point each of them must involve dealing with actual source text of some kind at each end of the process[1]. Reengineering assistant, which runs in Windows platform and uses a Wide Spectrum Language (We called it WSL)[7-9] as both program language and specification language, can be used to abstract a program into a specification which is changing one form into another form of the given program without changing its functionality. Certainly, it is used to transform one source text into another which is better than before. This paper introduces the basic features of reengineering assistant, some theoretical foundations used in this tool and how to use it to apply a series of transformations to an existing code to achieve a clear, easily understood code (i.e. derive

HE Zheng-chu (1968-), male, Ph.D., associate professor of Department of Business Studies, Hunan College of Finance and Economics, postdoctor of Chinese Academy of Social Sciences; research fields: enterprise management, enterprise strategy, enterprise crisis management.

a specification from a program).

2. About Reengineering Assistant

The software Reengineering Assistant is a tool to facilitate software maintenance, restructuring, reverse engineering and reengineering, which uses a wide spectrum language to represent both program logics (such as assignment statement, condition statement, and iteration statement etc.) and specification structures. Since both programs and specifications are part of the same language in our system, it makes program transformation applied to a program possible and transformations can be used to demonstrate that a given program is a correct implementation of a given specification, or that a specification correctly captures the behavior of a program. This approach means that very considerable changes can be made to the form of the program without changing its functionality.

Now we successfully finished developing reengineering assistant (RA) of windows version, which is compatible with the previous version (Unix version) taken practice several years, it is said that both versions have the same functionalities and can share result, the following Figure 1 shows the whole structure about reengineering assistant in whole reverse engineering process.

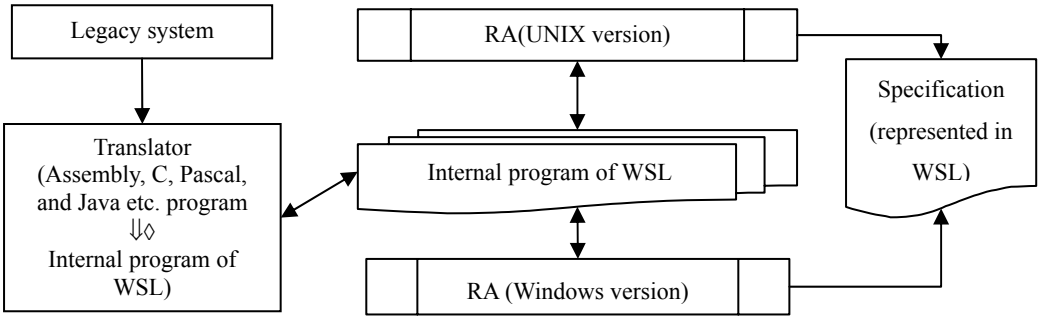


Figure 1 Reverse engineering with RA

From the Figure 1, the reverse engineering almost becomes automatic processing at program code level. Firstly, the legacy system codes can be translated into the internal program codes of WSL form which includes all process logics. For instance, the following program piece can be formed in another form:

```
y:=0;
x:=0;
if (x<>0) then
    skip else
    for i:=1 to 5 step 2 do
        y:=(y+i)
    od
fi
```

The following piece of program describes the above program processing logics.

((ASSIGN (Y 0)) (ASSIGN (X 0)) (COND ((<> X 0) (SKIP)) ((ELSE) (FOR I 1 5 2 (ASSIGN (Y (+ Y I)))))))

Since a wide spectrum language is used in our tool and lots of program processing logics are defined, the reengineering assistant can theoretically work with any programming language if corresponding translator is provided.

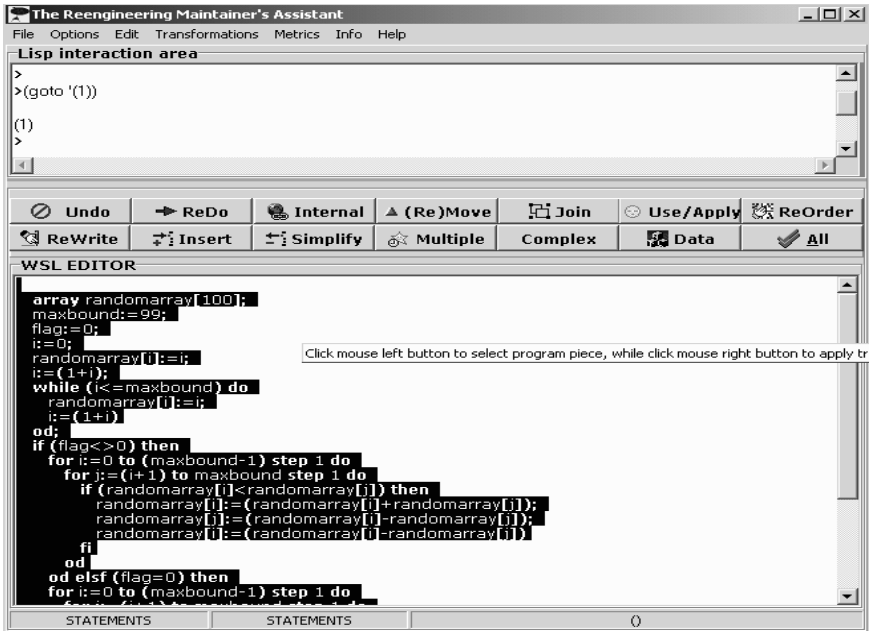


Figure 2 The interface of RA running on windows platform

As the Figure 2 shows, the system incorporates a friendly graphical interface, intelligent program printing algorithm, structure editor, a library of proven transformations and browser. This allows the user to easily move through any branch and leaf of the program tree (the whole program is storied in tree structure in this tool) with mouse left button and four direction keys (left, up, down, right), apply transformations, undo changes he has made, and in special circumstances, edit the program manually, but always in such a way that it is syntactically correct. With a knowledge based heuristics, the system can be used to analyze large programs and suggest suitable transformations as well as carry out the transformations and check the applicability conditions. This means that the correctness of the resulting transformed program is guaranteed by the system rather than being dependent on the user.

Additionally, like most commercial software the reengineering assistant setup file incorporates all the environments in which the tool can be run. And with a help document it is very easy for the user learning how to use this tool.

3. Functionalities of the Reengineering Assistant

With this tool's aid, much work associated with program transformation and understanding may become easy for the user, the followings are some merits of the reengineering assistant.

Structure editor makes the user easily write a new program from scratch or modify an existing program translated from all kinds of legacy system. Those codes the user inputs can be automatically changed into internal form of WSL which is formed in LISP language by system. It is said that statement "i:=(1+i)" can be changed into "(ASSIGN (I (+ 1 I)))" which is available for handling by our transformation system.

The transformation engine currently implements over 500 transformations, which are organized into a set of about 20 "generic" transformations (for moving, deleting, inserting, merging, separating etc.) and they consist of 11 types. When a transformation type button is clicked, only the transformations of that type which are correctly applicable to the current selected item are displayed. This organization of the catalogue means that despite the large number of transformations available, only a handful will appear when a button is clicked at any time.

A history/future (undo/redo) control allows backtracking to any previous version of the program, and re-tracing the steps forwards again if required. This allows the operator to retrace his steps if the route he was pursuing does not result in simplified code; all these operations can be finished only click “Undo” or “Redo” menu items or buttons.

The indented program displaying, which is a very good structural programming style, in “WSL EDITOR” area always correctly represents the structure of the program, and makes the program more clear for the operator. For example:

```
for i:=0 to (maxbound-1) step 1 do
    for j:=(1+i) to maxbound step 1 do
        if (randomarray[i]>randomarray[j]) then
            randomarray[i]:=(randomarray[i]+randomarray[j]);
            randomarray[j]:=(randomarray[i]-randomarray[j]);
            randomarray[i]:=(randomarray[i]-randomarray[j])
        fi
    od
od
```

The system can calculate various metrics including McCabe complexity, structural complexity and number of nodes metrics etc., which is very helpful for the user selecting suitable transformations and measuring the progress in improving the quality of the program.

Since we are now working within the formal method and have a rigorous theoretical basis, any suitable sequence of transformations can be applied to the confidence that the semantics of the program will be unchanged (i.e. without changing its functionality or its functionality is reserved).

In addition program tree branches can be “folded” (temporarily hidden) to make the structure clearer or “unfolded” for those folded branches, for instance, the following “for” statement can be folded in WSL EDITOR with clicking “fold” menu.

```
array randomarray[100];
maxbound:=99;
flag:=0;
i:=0;
randomarray[i]:=i;
i:=(1+i);
while (i<=maxbound) do
    randomarray[i]:=i;
    i:=(1+i)
od;
for i:=0 to (maxbound-1) step 1 do
    for j:=(1+i) to maxbound step 1 do
        if (randomarray[i]>randomarray[j]) then
            randomarray[i]:=(randomarray[i]+randomarray[j]);
            randomarray[j]:=(randomarray[i]-randomarray[j]);
            randomarray[i]:=(randomarray[i]-randomarray[j])
        fi
    od
od
```

The following piece of program shows the result when “for” statement is folded in the above piece of program. “----------For----------” means that it is a “for” statement. Like this, when most complicated statements are folded, the program will become a simplifying and clear specification, which is very important for the user analyzing the program to achieve an outline specification.

```
array randomarray[100];
    maxbound:=99;
    flag:=0;
    i:=0;
    randomarray[i]:=i;
    i:=(1+i);
    while (i<=maxbound) do
        randomarray[i]:=i;
        i:=(1+i)
    od;
```

----------For----------

The on-screen representation of WSL is controlled by a simple look-up table and can be changed to resemble a language familiar to the users. In this tool, we use a representation which is some like Pascal language to output WSL program (specification).

4. A Brief Introduction to WSL

The "Wide Spectrum Language" (WSL), which has no restriction to finite values and computable operations, includes low-level programming constructs and high-level abstract specifications within a single language. By working within a single language we are able to prove that a program correctly implements a specification, or that a specification correctly captures the behavior of a program, by means of formal transformations in the language. We don't have to develop transformations between the "programming" and "specification" languages. An added advantage is that different parts of the program can be expressed at different levels of abstraction, if required.

The following sections some information about WSL will be given. In the followings S_1, S_2 etc., are statements, Q, B etc., are formulae, x_1, x_2 etc., are variable and e_1, e_2 etc., represent any valid expressions:

Numeric operators: $e_1 + e_1$, $e_1 - e_1$, $e_1 * , e_1 , e_1 / e_1$ and so on, with the usual meanings like in common programming language.

Sequences: s = $< a_1, a_2, \ldots, a_n >$ is a sequence.

Relations: $e_1 = e_2$, $e_1 > e_2$, $e_1 >= e_2$, $e_1 < e_2$, $e_1 <= e_2$ $e_1 <> e_2$.

Logical operators: $\neg Q$, $Q_1 \vee Q_2$, $Q_1 \wedge Q_2$.

Sequential composition: $S_1 ; S_2 ; \ldots ; S_n$

Assertion: {B}. An assertion is a partial skip statement, it aborts if the condition is false but does nothing if the condition is true.

Simple assignment: $< x_1, \ldots, x_n >:=< e_1, \ldots, e_n >$. This assigns the values of the expressions e_i to the variables x_i. The assignments are carried out simultaneously, for example, **<y:=x, x:=y>** can be used to swap the values of x and y, which equals the following three statements:

```
x:=(x+y);
y:=(x-y);
x:=(x-y)
```

The single assignment $< x > :=< e >$ can be abbreviated to $x := e$.

Deterministic choice: if B then S_1 else S_2 fi. The choice of which statement to execute is determined by the condition B.

Deterministic iteration: while B do S od, do if B then S_1 fi S_2 od. The condition B is tested and S is executed repeatedly until B becomes false in while statement, and generally S_1 includes exit statement in do structure statement.

Local variables: var x: datatype:=initvalue: S endvar. Here x is a local variable which only exists within the statement S.

Counted iteration: for $i := lowvalue$ to $highvalue$ step $stepvalue$ do S od.

Unbounded loops and exits: Statements of the form do S od, where S is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form exit(n) (where n is an integer, not a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow exits which leave a block or a loop other than an unbounded loop. This type of structure is described in Knuth (1974) and more recently in Taylor (1984). [6]

5. Case Studies

In this section, we will use the tool to transform a more complicated program into a clear and simplifying program. The following program is the source code for transforming:

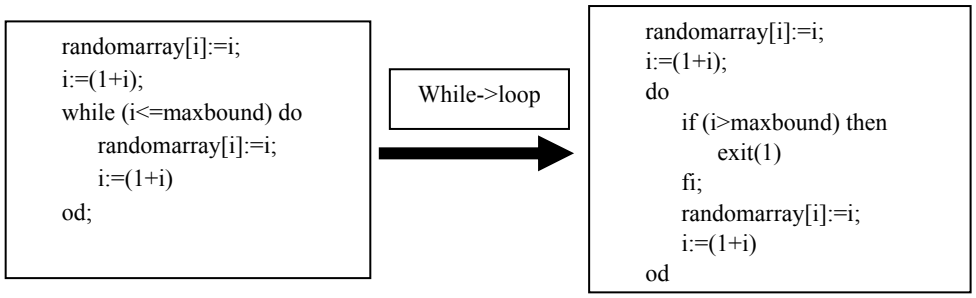
```
array randomarray[100];
maxbound:=99;
flag:=0;
i:=0;
randomarray[i]:=i;
i:=(1+i);
while (i<=maxbound) do
    randomarray[i]:=i;
    i:=(1+i)
od;
if (flag<>0) then
    for i:=0 to (maxbound-1) step 1 do
        for j:=(i+1) to maxbound step 1 do
            if (randomarray[i]<randomarray[j]) then
                randomarray[i]:=(randomarray[i]+randomarray[j]);
                randomarray[j]:=(randomarray[i]-randomarray[j]);
                randomarray[i]:=(randomarray[i]-randomarray[j])
            fi
        od
    od elsf (flag=0) then
    for i:=0 to (maxbound-1) step 1 do
        for j:=(i+1) to maxbound step 1 do
            if (randomarray[i]>randomarray[j]) then
                randomarray[i]:=(randomarray[i]+randomarray[j]);
                randomarray[j]:=(randomarray[i]-randomarray[j]);
                randomarray[i]:=(randomarray[i]-randomarray[j])
            fi
        od
    od
fi
```

Firstly, we can use "simplify" transformation to simplify the whole program, with "simplify" transformation we can remove those redundant or dead code in program source codes and simplify some expressions. After

applying “simplify” transformation to the above program, it will become the following more simplifying program:

```
array randomarray[100];
maxbound:=99;
flag:=0;
i:=0;
randomarray[i]:=i;
i:=(1+i);
while (i<=maxbound) do
    randomarray[i]:=i;
    i:=(1+i)
od;
for i:=0 to (maxbound-1) step 1 do
    for j:=(1+i) to maxbound step 1 do
        if (randomarray[i]>randomarray[j]) then
            randomarray[i]:=(randomarray[i]+randomarray[j]);
            randomarray[j]:=(randomarray[i]-randomarray[j]);
            randomarray[i]:=(randomarray[i]-randomarray[j])
        fi
    od
od
```

From the result of “simplify” transformation operation, we will amazedly find that “if” statement becomes a “for” statement which is one of the branches in the previous “if” statement. In fact, presumed flag equals 0, the program will always execute the branch when condition is true (i.e. flag=0), while another branch will never be executed (i.e. dead code). The result program really captures the functionality of the previous program and the result program becomes more readable and clearer. The further transformation can be used to restructure the above program, we can transform “while” statement into “do…od” statement through “while−>loop” transformation operation.



The above figure shows the result after applying “while−>loop” transformation to “while” statement, moreover, we can change the new result source into a friendly structure for the user.

The below figure shows the whole result when a “while” structure is changed into a “do…od” structure. Through these restructuring transformations, the program becomes another form in structure which perhaps can be more easily understood for some users, but without any change of the program functionality. Certainly, further transformation operations can be applied to the program, the following figure shows the result when “Merge−−>>” is applied to the three statements “maxbound**:=**99”, “flag**:=**0”, “i**:=**0” and “Replace_all_values_ optimally” transformation operation is applied to the whole program. From which the new program becomes clear, structural, easily understood with its functionality is reserved. It is more easily for the user to master the modified program.

```
 randomarray[i]:=i;
 i:=(1+i);
do
    if (i>maxbound) then
        exit(1)
    fi;
    randomarray[i]:=i;
    i:=(1+i)
od
```

Merge−>>

```
do
    randomarray[i]:=i;
    i:=(1+i);
    if (i>maxbound) then
        exit(1)
    fi
od
```

```
array randomarray[100];
<maxbound:=99, flag:=0, i:=0>;
do
    randomarray[i]:=i;
    i:=(1+i);
    if (i>99) then
        exit(1)
    fi
od;
for i:=0 to (99-1) step 1 do
    for j:=(1+i) to 99 step 1 do
        if (randomarray[i]>randomarray[j]) then
            randomarray[i]:=(randomarray[i]+randomarray[j]);
            randomarray[j]:=(randomarray[i]-randomarray[j]);
            randomarray[i]:=(randomarray[i]-randomarray[j])
        fi
    od
od
```

6. Conclusions

The reengineering assistant can theoretically work with any programming language if corresponding translator is provided. The objective of reengineering assistant is developing a tool unrelated to concrete programming language to help software engineers or maintainers understand their program more easily and convenient, so different translators which are used to translate different languages program into the internal program of WSL and translate the internal program of WSL into different languages' program are required in the further research.

Reference:

[1] Javes R. Cordy. Source Transformation in Software Engineering Using the TXL Transformation System. *European Journal of Operational Research*, 2004, 17: 35-44.

[2] T.J. Biggerstaff. Design Recovery for Maintenance and Reuse. *IEEE Computer 22.7*, 1989(7): 34-49.

[3] R.S. Amold,. Software Restructuring. *Proceedings of the IEEE 77*, 1989(4): 607-617.

[4] J.R. Cordy, M. Shukla. Practical MetaProgramming. *Proceedings of the 1992 IBM Centre for Advanced Studies Conference*, Toronto, Canada, 1992: 215-224.

[5] Tim Bull. A Transformation System for Maintenance—Turning Theory into Practice, *Proceedings of the Conference on Software Maintenance*, 2005.

[6] H.A. Priestley, M.P. Ward. *A Multipurpose Backtracking Algorithm*. Oxford University, DPhil Thesis, 2005.

[7] Martin Ward. *Proving Program Refinements and Transformations*. Oxford University, DPhil Thesis, 2005.

[8] Martin Ward. Specifications and Programs in a Wide Spectrum Language. *J. Assoc. Comput. Mach.*, 2005(4).

[9] Martin Ward. A Model for Partial Programs. *J. Assoc. Comput. Mach.*, 2004(11).

(Edited by Gavin Dai, Shirley Hu and Jimmy)