Formal Transformations and WSL

Part Three

Martin Ward
Reader in Software Engineering
martin@gkc.org.uk

Software Technology Research Lab
De Montfort University
Transformations are implemented in an extension of WSL, called $\mathcal{M}_{\text{ETA}}$ WSL.

WSL has been developed specifically to be a powerful programming language which is easy to transform.

$\mathcal{M}_{\text{ETA}}$ WSL has been developed as a language in which it is easy to implement program transformations.
For example:

```plaintext
ifmatch Statement if ~?B then ~?S_1 else ~?S_2 fi
    then B := @Not(B);
        @Paste_Over(fill Statement
            if ~?B then ~?S_2 else ~?S_1 fi endfill) endmatch
```

in ASCII form this is:

```plaintext
IFMATCH Statement IF ~?B THEN ~?S1 ELSE ~?S2 FI
THEN B := @Not(B);
        @Paste_Over(FILL Statement
            IF ~?B THEN ~?S2 ELSE ~?S1 FI ENDFILL) ENDMATCH
```
The IFMATCH construct will test if the currently selected statement matches the pattern:

$$\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

If it does, then new local variables $B$, $S_1$ and $S_2$ are created. $B$ contains the condition from the current if statement, $S_1$ contains the statement sequence from the then part and $S_2$ contains the statement sequence from the else part.

The function @Not will negate and then simplify the condition.

The construct FILL Statement ...ENDFILL creates a new statement by filling in the given pattern with the values of the given variables. This is passed to the @Paste_Over procedure which replaces the current statement with the new one.
METAWSL: Extensions to WSL

Moving around the abstract syntax tree:
MetaWSL: Extensions to WSL

Moving around the abstract syntax tree:

@Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)
MetaWSL: Extensions to WSL

Moving around the abstract syntax tree:

@Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:
Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
MetaWSL: Extensions to WSL

Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
- @GT(I) returns the generic type of the item (Statement, Expression etc.)
METAWSL: Extensions to WSL

Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
- @GT(I) returns the generic type of the item (Statement, Expression etc.)
- @ST(I) returns the specific type (While, Variable etc.)
**METAWSL: Extensions to WSL**

Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
- @GT(I) returns the *generic* type of the item (Statement, Expression etc.)
- @ST(I) returns the *specific* type (While, Variable etc.)
- @V(I) returns the value (eg the name of a Variable)
MetaWSL: Extensions to WSL

Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
- @GT(I) returns the generic type of the item (Statement, Expression etc.)
- @ST(I) returns the specific type (While, Variable etc.)
- @V(I) returns the value (eg the name of a Variable)
- @Cs(I) returns the list of components for the node I.
**METAWSL: Extensions to WSL**

Moving around the abstract syntax tree:

- @Up, @Down, @Left, @Right, @To_Last, @To(n), @Down_Last, @Down_To(n) and @Goto(P)

Examining the tree:

- @Item returns the item at the current position
- @GT(I) returns the *generic* type of the item (Statement, Expression etc.)
- @ST(I) returns the *specific* type (While, Variable etc.)
- @V(I) returns the value (eg the name of a Variable)
- @Cs(I) returns the list of components for the node I.
- @Size(I) returns the number of components for the node I.
MetaWLS Editing Procedures

Editing the tree:
METAWSL Editing Procedures

Editing the tree:

- @Delete, @Clever_Delete, @Cut,
Editing the tree:

- @Delete, @Clever_Delete, @Cut,

- @Paste_Over, @Paste_Before, @Paste_After,
METAWSL Editing Procedures

Editing the tree:

- @Delete, @Clever_Delete, @Cut,
- @Paste_Over, @Paste_Before, @Paste_After,
- @Splice_Over, @Splice_Before, @Splice_After
Editing the tree:

- @Delete, @Clever_Delete, @Cut,
- @Paste_Over, @Paste_Before, @Paste_After,
- @Splice_Over, @Splice_Before, @Splice_After

These are applied to the current position in a tree.
METAWSL Editing Procedures

Editing the tree:

- @Delete, @Clever_Delete, @Cut,
- @Paste_Over, @Paste_Before, @Paste_After,
- @Splice_Over, @Splice_Before, @Splice_After

These are applied to the current position in a tree.

Building a tree:

- @Make(t, v, L) returns a new item with specific type $s$, value $v$ and components $L$ (where $L$ is a list of items). This can be inserted in the tree using the edit operations.
MetaWSL Editing Procedures

Editing the tree:

- @Delete, @Clever_Delete, @Cut,
- @Paste_Over, @Paste_Before, @Paste_After,
- @Splice_Over, @Splice_Before, @Splice_After

These are applied to the current position in a tree.

Building a tree:

- @Make\( (t, v, L) \) returns a new item with specific type \( s \), value \( v \) and components \( L \) (where \( L \) is a list of items). This can be inserted in the tree using the edit operations.

For example:

\[
\text{@Paste}_\text{Over}(\text{@Make}(\text{T\_Number}, \text{@V}(\text{@I}) + 1, \langle \rangle))
\]
METAWSL Editing Requirements

- Keep multiple versions of the program, stored efficiently
- Revert to a previous version very efficiently
- Efficient and transparent caching of program analysis results
META WSL Editing Solution

- Store abstract syntax trees as lisp trees
- Editing functions create a new tree, sharing subtrees
- Caches (of program analysis results) are stored in each tree node
- Use @Edit, @End_Edit and @Undo_Edit for efficiency
**MetaWSL Pattern Matching**

**ifmatch**: match the current node against a WSL program schema

**fill**: creates an abstract syntax tree by filling in the schema variables in a WSL schema.

Within an **ifmatch** construct *pattern variables* are allowed:

1. $\sim ?x$ matches any item and puts the matched result into variable $x$;

2. $\sim *x$ matches a sequence of zero or more items and puts the result into $x$;

3. $\sim \mathit{=} (e)$ matches the current item against the value of the expression $e$. 
Within a `fill` construct,:

1. `$\sim ?x$` pastes in the current value of $x$ at this position;

2. `$\sim *x$` splices the list of items in $x$ over the pattern variable;

3. `$\sim = (e)$` pastes in the value of expression $e$ at this position.
**METAWSL Looping Constructs**

`foreach` and `ateach` enable iteration over all components of the current item.

The body of the loop is executed at each selected component.

The components iterated over are specified in the keyword after the word `foreach`:

- All statements: `foreach Statement do S od`
- Terminal statements: `foreach TS do S od`
- Simple terminal statements: `foreach STS do S od`
- All expressions: `foreach Expression do S od`
- Conditions, Lvalues, variables, etc.
METAWSL Looping Constructs

**foreach**

- Acts “bottom up” (components of an item are processed before the item itself)
- Works as if the current item is the whole program
- Therefore, editing is efficient, but little context information is available.

**ateach**

- Acts “top down” (process each item, then its components)
- Moves to each component before processing
- Therefore, editing is inefficient, but full context is available.
Looping Example

proc @Delete_All_Skips_Test() ≡
   if Skip ∈ @Stat_TYPES(@I)
      then @Pass
      else @Fail("No ‘SKIP’ statements to delete.") fi.;

proc @Delete_All_Skips_Code(Data) ≡
   foreach Statement do
      if @ST(@I) = Skip then @Delete fi od.

Deleting a skip is always a valid transformation.

The syntax of the edited program is automatically “fixed” if necessary, by the foreach loop.
## Effect of Delete All Skips

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>while B do skip od</td>
<td>( {\neg B} )</td>
</tr>
<tr>
<td>if B then skip</td>
<td>if ( \neg B ) then ( x := 0 ) fi</td>
</tr>
<tr>
<td>else ( x := 0 ) fi</td>
<td></td>
</tr>
<tr>
<td>do skip od</td>
<td>abort</td>
</tr>
<tr>
<td>var ( \langle x := 0 \rangle :</td>
<td>( y := 0 )</td>
</tr>
<tr>
<td>if B then skip fi end;</td>
<td></td>
</tr>
<tr>
<td>( y := 0 )</td>
<td></td>
</tr>
</tbody>
</table>
Effect of Delete All Skips

An example application of Delete_All_Skips:

\[
\text{if } x = 0 \\
\quad \text{then while } y > 0 \text{ do skip od} \\
\text{elsif } z = 0 \\
\quad \text{then begin} \\
\quad \quad \text{if } a = b \text{ then skip fi} \\
\quad \text{where} \\
\quad \text{proc } F(x) \equiv y := y + x. \\
\text{end} \\
\text{else skip fi}
\]
Effect of Delete All Skips

An example application of Delete_All_Skips:

if $x = 0$
  then while $y > 0$ do skip od
elsif $z = 0$
  then begin
    if $a = b$ then skip fi
  where
  proc $F(x) \equiv y := y + x.$
  end
else skip fi

The result is:

if $x = 0$ then $\{y \leq 0\}$ fi
Meta-Transformations

Part of the source code for WSL to Scheme:

```scheme
ifmatch Statements ∼*S1; if ∼?B then ∼*S; exit(1) fi; ∼*S2
   then @Up;
       if @Gen_Proper?(@Make(T_Statements,
           ⟨⟩, S1 + S + S2), AS)
           then B := @Not(B);
               @Splice_Over(@Cs(fill Statements
                   ∼*S1;
                   while ∼?B do
                       ∼*S2; ∼*S1 od;
                       ∼*S endfill)))
           else @Trans(TR_Floop_To_While, "" ) fi
           else @Up; @Trans(TR_Floop_To_While, "" ) endmatch
```
Meta-Transformations

An example of applying this transformation:

```plaintext
do  read(file, record);
    if  eof?(file)
        then close(file);
            exit(1) fi;
    process(record, total) od
```

is transformed into:

```plaintext
read(file, record);
while ¬eof?(file) do
    process(record, total);
    read(file, record) od;
close(file);
```
Meta-Transformations

Part of the source code for the transformation Floop_To_While. This tries to make a statement reducible without duplicating code:

```plaintext
foreach Statements do
  if Depth = 1
    then @Down_Last;
    do if @Right? ∧ ¬@Is_Proper?
      then N := 0;
      ateach STS do
        if Depth ∈ @Gen_TVs(@l, ASType)
          then N := N + 1 fi od;
      if N > 1 then exit(1) fi;
      PRINFLUSH("a"); done := 0;
      @Trans(TR_Fully_Absorb_Right, "");
      exit(1) fi;
  if @Left? then @Left else exit(1) fi od fi od
```
A Sample of METAWSL

Part of the Absorb_right transformation:
@Right; @Cut; @Left;
...
foreach STS do
  if Depth = 0 ∨ (@ST(@I) = Exit ∧ @V(@I) = Depth)
    then if @ST(@I) = Exit ∧ Depth > 0
      then @Splice_Over(@Increment(@Buffer,
                   AS_Type, Depth, 0))
    elsif @ST(@I) = Skip
      then @Paste_Over(@Buffer)
    elsif @ST(@I) = Exit ∧ Depth = 0
      ∨ @Gen_Improper?(@I, AS)
       then skip
    elsif @ST(@I) = Call ∧ @V(@I) = “Z”
       then skip
    else @Paste_After(@Buffer) fi fi od;
Meta-Transformations

Each time FermaT is rebuilt from source, the Flop_To_While transformation is applied to its own source code!
Expression/Condition Simplifier

For the industrial strength FermaT transformation system the requirements for an expression and condition simplifier were:

1. Efficient execution: especially on small expressions;

2. Easily extendible by adding new pattern match and replacement rules: extensive searching based on a small set of rules is too expensive

3. Easy to prove correct. If the simplifier is to be easily extended, then it is important that we can prove the correctness of the extended simplifier equally easily.
foreach Expression do
    ifmatch Expression \((-(-\sim?x))\)
        then @Paste_Over(x) endmatch;

    ifmatch Expression \(1/(1/\sim?x)\)
        then @Paste_Over(x) endmatch;

    ifmatch Expression \((\sim?y \ast \sim?x) \div \sim?x\)
        then @Paste_Over(y) endmatch;

    ifmatch Expression \((\sim?y \ast \sim?x + \sim?z) \div \sim?x\)
        then @Paste_Over(fill Expression
            \sim?y + (\sim?z \div \sim?x) endfill) endmatch;
        ... od;
foreach Condition do
    ifmatch Condition \( \sim ?x \lessdot \sim ?y \lor \sim ?y \lessdot \sim ?x \)
        then @Paste_Over(fill Condition \( \sim ?x \neq \sim ?y \) endfill) endmatch;
    ifmatch Condition \( \sim ?z \lessdot \sim ?x \lor \sim ?x \lessdot \sim ?y \lor \sim ?y \lessdot \sim ?z \)
        then @Paste_Over(Mth_True) endmatch;

    ifmatch Condition \( \sim ?y \lessdot \sim ?x \land \sim ?z \lessdot \sim ?y \land \sim ?x \lessdot \sim ?z \)
        then @Paste_Over(Mth_False) endmatch;

    ...
od;
Specify the simplifier as a list of pattern match and replacement rules, using `ifmatch` and `fill`. This meets requirement (2).

implement the simplifier as a large, deeply-nested set of `if` statements which test the specific type of the current item, the number of components and the types of the components. This meets requirement (1).

Automatically Transform the specification into the implementation via a meta-transformation (which transforms the source code of one transformation into source code for an equivalent, but more efficient, transformation). This meets requirement (3).

As new rules are added to the specification of the transformation, the implementation is generated automatically.
Expression/Condition Simplifier

- Specification: 19,465 bytes of WSL
- Implementation: 85,786 bytes of efficient, lower level WSL
- Scheme Translation: 274,032 bytes of Scheme
- Macro Expansion: 884,943 bytes of Scheme
- C Translation: 1,089,448 byte C file plus associated 108,098 byte header file

So the original WSL source file expands into a C implementation which is 60 times larger.

Which would you rather maintain?
Using FermaT in Reverse Engineering

The next few slides look at

- A typical software development process
- A typical “maintenance phase” and its output
- Using FermaT to reverse engineer the program and produce an efficient and maintainable improved version
- Using FermaT to raise the abstraction level of the program all the way to a formal specification
### Problem Specification

#### Input Data

<table>
<thead>
<tr>
<th>ITEM</th>
<th>Net Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bolt</td>
<td>+200</td>
</tr>
<tr>
<td>Bolt</td>
<td>−150</td>
</tr>
<tr>
<td>Bolt</td>
<td>−25</td>
</tr>
<tr>
<td>Nut</td>
<td>+100</td>
</tr>
<tr>
<td>Nut</td>
<td>−100</td>
</tr>
<tr>
<td>Wheel</td>
<td>+40</td>
</tr>
<tr>
<td>Wheel</td>
<td>−10</td>
</tr>
<tr>
<td>Widget</td>
<td>−500</td>
</tr>
</tbody>
</table>

#### Management Report

<table>
<thead>
<tr>
<th>ITEM</th>
<th>Net Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bolt</td>
<td>+25</td>
</tr>
<tr>
<td>Wheel</td>
<td>+30</td>
</tr>
<tr>
<td>Widget</td>
<td>−500</td>
</tr>
</tbody>
</table>

Number Changed: 3
Design Phase

- Produce Report
  - Produce Heading
  - Produce Body
  - Produce Summary
- Process record
  - Process first record in group
  - Process subsequent record in group
- Process end of previous group
- Start new group
- Process first record in group
Final Functional Decomposition

\[ \text{proc Management\_Report} \equiv \]
\[ \text{Produce\_Heading;} \]
\[ \text{read(stuff);} \]
\[ \text{while NOT eof(stuff) do} \]
\[ \text{if First\_Record\_In\_Group} \]
\[ \text{then Process\_End\_Of\_Previous\_Group;} \]
\[ \quad \text{Process\_Start\_Of\_New\_Group;} \]
\[ \quad \text{Process\_Record} \]
\[ \text{else} \]
\[ \quad \text{Process\_Record} \]
\[ \text{fi;} \]
\[ \text{read(stuff)} \]
\[ \text{od;} \]
\[ \text{Produce\_Summary.} \]
Problem

Users complain about the line of garbage which appears at the top of each report.
Problem

Users complain about the line of garbage which appears at the top of each report.

This is because we call Process_End_Of_Previous_Group (which prints a report line) before any records have been processed.

What is the solution to this “first time through” problem?
First Quick Fix

\[
\text{proc Management\_Report} \equiv \\
\begin{array}{l}
\text{var} \langle \text{SW1} := 0 \rangle : \\
\text{Produce\_Heading;}
\text{read(stuff);}
\textbf{while} \ NOT \ \text{eof(stuff)} \ \textbf{do}
\textbf{if} \ \text{First\_Record\_In\_Group}
\textbf{then} \textbf{if} \ \text{SW1} = 1
\textbf{then} \text{Process\_End\_Of\_Previous\_Group}
\textbf{fi;}
\text{SW1} := 1;
\text{Process\_Start\_Of\_New\_Group;}
\text{Process\_Record}
\textbf{else}
\text{Process\_Record}
\textbf{fi;}
\text{read(stuff)}
\textbf{od;}
\text{Produce\_Summary}
\end{array}
\text{end.}
\]
Problem

The new *Zymometers* have been on sale for quite a while now, and selling quite well, but they don't appear on the report.
Problem

The new *Zymometers* have been on sale for quite a while now, and selling quite well, but they don't appear on the report.

We only call `Process_End_Of_Previous_Group` when we detect the start of the *next* group. So the very last group of all is missed off the report.
Second Quick Fix

\texttt{proc Management\_Report \equiv}
\begin{verbatim}
\hspace{1em} \texttt{var \langle SW1 := 0 \rangle :}
\hspace{1em} \texttt{Produce\_Heading;}
\hspace{1em} \texttt{read(stuff);} \\
\hspace{1em} \texttt{while NOT eof(stuff) do}
\hspace{2em} \texttt{if First\_Record\_In\_Group}
\hspace{3em} \texttt{then if SW1 = 1}
\hspace{4em} \texttt{then Process\_End\_Of\_Previous\_Group}
\hspace{5em} \texttt{fi;}
\hspace{4em} \texttt{SW1 := 1;}
\hspace{5em} \texttt{Process\_Start\_Of\_New\_Group;}
\hspace{5em} \texttt{Process\_Record}
\hspace{3em} \texttt{else}
\hspace{4em} \texttt{Process\_Record}
\hspace{5em} \texttt{fi;}
\hspace{2em} \texttt{read(stuff)}
\hspace{1em} \texttt{od;}
\hspace{1em} \texttt{Process\_End\_Of\_Last\_Group;}
\hspace{1em} \texttt{Produce\_Summary}
\end{verbatim}
\texttt{end.}
Problem

The line of garbage has re-appeared at the top of the report!
Problem

The line of garbage has re-appeared at the top of the report!

It turns out that there was a strike at the warehouse this week: no items came in our out. So there were no records in the file. But the program calls Process_End_Of_Last_Group anyway.
proc Management_Report ≡
  var ⟨SW1 := 0, SW2 := 0⟩ :
  Produce_Heading;
  read(stuff);
  while NOT eof(stuff) do
    if First_Record_In_Group
      then if SW1 = 1
        then Process_End_Of_Previous_Group
        fi;
        SW1 := 1;
        Process_Start_Of_New_Group;
        Process_Record
      else
        Process_Record; SW2 := 1
        fi;
    read(stuff)
  od;
  if SW2 = 1 then Process_End_Of_Last_Group
  fi;
  Produce_Summary
end.
Problem

The *Zymometers* have disappeared off the report again!
The **Zymometers** have disappeared off the report again!

It turns out that the warehouse manager decided to consolidate all orders for each item each day into a single transaction. He also decided to run the report once a day, instead of once a week. So there was never more than one record in each group.

So SW1 never gets set, and the last group never gets processed.
proc Management_Report ≡
var ⟨SW1 := 0, SW2 := 0⟩ :
    Produce_Heading;
    read(stuff);
while NOT eof(stuff) do
    if First_Record_In_Group
        then if SW1 = 1
            then Process_End_Of_Previous_Group
                fi;
            SW1 := 1;
            Process_Start_Of_New_Group;
            Process_Record;
            SW2 := 1
        else
            Process_Record; SW2 := 1
            fi;
    read(stuff)
    od;
if SW2 = 1 then Process_End_Of_Last_Group
    fi;
    Produce_Summary
end.
After Four Quick Fixes... 

Now we can all rest assured that it works, right?
After Four Quick Fixes…

Now we can all rest assured that it works, right?

What can FermaT do with this program?
WSL Version of Procedure Body

var \langle SW1 := 0, SW2 := 0 \rangle :

!P Produce_Heading( var sys);
!P read( var stuff, sys);
while ¬!XC eof(stuff) do
  if !XC First_Record_In_Group?(stuff)
    then if SW1 = 1
      then !P Process_End_Of_Group( var sys) fi;
      SW1 := 1;
      !P Process_Start_Of_Group( var sys);
      !P Process_Record( var sys);
      SW2 := 1
    else !P Process_Record( var sys); SW2 := 1 fi;
    !P read( var stuff) od;
  if SW2 = 1
    then !P Process_End_Of_Group( var sys) fi;
!P Produce_Summary( var sys) end
Restructure: Remove SW2

Unroll the loop, absorb the following statement and use ConstantPropagation and Remove_Redundant_Vars to remove SW2:

```plaintext
var ⟨SW1 := 0⟩:

!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then if !XC First_Record_In_Group?(stuff)
        then SW1 := 1;
            !P Process_Start_Of_Group( var sys) fi;
        !P Process_Record( var sys);
        !P read( var stuff);
    while ¬!XC eof?(stuff) do
        if !XC First_Record_In_Group?(stuff)
            then if SW1 = 1
                then !P Process_End_Of_Group( var sys) fi;
                SW1 := 1;
                !P Process_Start_Of_Group( var sys) fi;
            !P Process_Record( var sys);
            !P read( var stuff) od;
        !P Process_End_Of_Group( var sys) fi;
!P Produce_Summary( var sys) end
```
Restructure: Remove SW1

Note that !XC First_Record_In_Group?(stuff) is true for the very first record read. FermaT cannot deduce this from the information in the program, so we have to edit the code. With this test fixed, we can remove SW1 in the same way:

!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then !P Process_Start_Of_Group( var sys);
         !P Process_Record( var sys);
         !P read( var stuff);
         while ¬!XC eof?(stuff) do
             if !XC First_Record_In_Group?(stuff)
                 then !P Process_End_Of_Group( var sys);
                     !P Process_Start_Of_Group( var sys) fi;
             !P Process_Record( var sys);
             !P read( var stuff) od;
         !P Process_End_Of_Group( var sys) fi;
!P Produce_Summary( var sys)

The next stage is to merge the two copies of Process_End_Of_Group...
Restructure: Merge Duplicated Code

Convert the **while** loop to a **do ... od** loop and absorb the second copy of Process_End_Of_Group (to move it closer to the first copy). This also allows us to absorb read and Process_Record into the loop:

```plaintext
!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then !P Process_Start_Of_Group( var sys);
        do !P Process_Record( var sys);
            !P read( var stuff);
            if !XC eof?(stuff)
                then !P Process_End_Of_Group( var sys);
                    exit(1) fi;
            if !XC First_Record_In_Group?(stuff)
                then !P Process_End_Of_Group( var sys);
                    !P Process_Start_Of_Group( var sys) fi od fi;
!P Produce_Summary( var sys)
```
Restructure: Merge Duplicated Code

In order to absorb Process_Start_Of_Group into the loop, we need the other copy to appear at the end of the loop body.

Convert the loop to a double loop (via Make_Loop), increment the call to Process_Start_Of_Group and take it out of the inner loop using Take_Out_Of_Loop:

!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then !P Process_Start_Of_Group( var sys);
        do do !P Process_Record( var sys);
            !P read( var stuff);
            if !XC eof?(stuff)
                then !P Process_End_Of_Group( var sys);
                    exit(2) fi;
            if !XC First_Record_In_Group?(stuff)
                then !P Process_End_Of_Group( var sys);
                    exit(1) fi od;
    !P Process_Start_Of_Group( var sys) od fi;
!P Produce_Summary( var sys)
Restructure: Merge Duplicated Code

Join the two if statements, move the (hidden) else skip clause to the top and use Elsif_To_Else_If to create a nested if statement:

```plaintext
!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then do !P Process_Start_Of_Group( var sys);
        do !P Process_Record( var sys);
            !P read( var stuff);
            if ¬!XC First_Record_In_Group?(stuff) ∧ ¬!XC eof?(stuff)
                then skip
            else if !XC eof?(stuff)
                then !P Process_End_Of_Group( var sys);
                    exit(2)
            else !P Process_End_Of_Group( var sys);
                    exit(1)
        fi
    od od fi;
!P Produce_Summary( var sys)
```

This statement can be taken out of the loop via Take_Out_Of_Loop. Then take Process_End_Of_Group out of the if statement.
Restructure: Merge Duplicated Code

!P Produce_Heading( var sys);
!P read( var stuff, sys);
if ¬!XC eof?(stuff)
    then do !P Process_Start_Of_Group( var sys);
        do !P Process_Record( var sys);
            !P read( var stuff);
            if !XC First_Record_In_Group?(stuff) ∨ !XC eof?(stuff)
                then exit(1) fi od;
        !P Process_End_Of_Group( var sys);
    if !XC eof?(stuff) then exit(1) fi
    od fi;
!P Produce_Summary( var sys)
Restructure and Simplify

Finally, the outer loop can be converted to a **while** loop via **Loop_To_While**. This will note that the test is at the *end* of the loop, but there is a surrounding **if** statement with the same test. So this **if** statement can be deleted:

```
P Produce_Heading( var sys);
P read( var stuff, sys);
while ¬!XC eof?(stuff) do
  P Process_Start_Of_Group( var sys);
  do P Process_Record( var sys);
     P read( var stuff);
     if !XC First_Record_In_Group?(stuff) ∨ !XC eof?(stuff)
        then exit(1) fi od;
  P Process_End_Of_Group( var sys) od;
P Produce_Summary( var sys)
```

In this version, there are no flag variables and no duplicated statements.
Second Stage: Abstract Data Types

We have processed this program about as far as possible at this abstraction level.

The next stage is to raise the abstraction level by defining abstract data types for the input and output files. The abstraction models the input file as a sequence of records, where each record has two fields: name (a string) and number (an integer). The abstract variable $i$ is an index into the sequence of records.

To recognise the start of a new group, the variable last stores a copy of the last record processed.
## Second Stage: Abstract Data Types

<table>
<thead>
<tr>
<th>Concrete Call</th>
<th>Abstract Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce_Heading</td>
<td>!\text{P} \ \text{write} (&quot;Management Report...&quot;)</td>
</tr>
<tr>
<td>read</td>
<td>last := record; i := i + 1; record := records[i]</td>
</tr>
<tr>
<td>eof?(stuff)</td>
<td>i &gt; \ell(\text{records})</td>
</tr>
<tr>
<td>Process_Start_Of_Group</td>
<td>total := 0</td>
</tr>
<tr>
<td>Process_Record</td>
<td>total := total + record.number</td>
</tr>
<tr>
<td>First_Record_In_Group?</td>
<td>last.name \neq record.name</td>
</tr>
<tr>
<td>Process_End_Of_Group</td>
<td>if total \neq 0</td>
</tr>
<tr>
<td></td>
<td>\quad \text{then} \ \text{write}(\text{last.name, total}); \ \text{changed} := \text{changed} + 1 \ \text{fi}</td>
</tr>
<tr>
<td>Produce_Summary</td>
<td>!\text{P} \ \text{write} (&quot;Changed items:&quot;, \text{changed})</td>
</tr>
</tbody>
</table>
Second Stage: Abstract Data Types

\texttt{var}\ \langle i := 0, \textasciitilde\textasciitilde, record := \textasciitilde\textasciitilde, changed := 0\rangle:\ \\
!P\ \texttt{write(\"Management Report\ldots\"}); \\
last := record; \ i := i + 1; record := records[i]; \\
\textbf{while} \ i \leq \ell(\text{records}) \ \textbf{do} \\
\hspace{1em} total := 0; \\
\hspace{1em} \textbf{do} total := total + record\texttt{.number}; \\
\hspace{1em} \hspace{1em} last := record; \ i := i + 1; record := records[i]; \\
\hspace{1em} \hspace{1em} \textbf{if} \ i > \ell(\text{records}) \ \vee \ last\texttt{.name} \neq \text{record}\texttt{.name} \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{then} \texttt{exit fi od}; \\
\hspace{1em} \textbf{if} \ total \neq 0 \\
\hspace{1em} \hspace{1em} \textbf{then} \!P\ \texttt{write(last.name, total)}; \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \texttt{changed := changed + 1 fi od}; \\
\hspace{1em} \!P\ \texttt{write(\"Changed items:\", changed)} \ \textbf{end}
First, we can replace record by records[i] throughout:

```plaintext
var ⟨i := 0, last := "", changed := 0⟩ :
  !P write("Management Report...");
  last := records[i]; i := i + 1;
  while i ≤ ℓ(records) do
    total := 0;
    do total := total + records[i].number;
      last := records[i]; i := i + 1;
      if i > ℓ(records) ∨ last.name ≠ records[i].name
         then exit fi od;
    if total ≠ 0
       then write(last.name, total);
          changed := changed + 1 fi od;
  !P write("Changed items:", changed) end
```

Now move the assignments to last forwards, and replace this variable by its value.
Third Stage: Restructure and Simplify

\[
\begin{align*}
\text{var} & \langle i := 0, \text{changed} := 0 \rangle : \\
& \quad \text{!P write(“Management Report...” var );}
\end{align*}
\]
\[
\begin{align*}
i & := 1; \\
\textbf{while} & \ i \ \leq \ \ell(\text{records}) \ \textbf{do} \\
& \quad \text{total} \ := \ 0; \\
& \quad \textbf{do} \quad \text{total} \ := \ (\text{total} + \text{records}[i].\text{number}); \\
& \quad \quad \ i \ := \ (i + 1); \\
& \quad \quad \textbf{if} \ i \ > \ \ell(\text{records}) \ \vee \ \text{records}[(i - 1)].\text{name} \ \neq \ \text{records}[i].\text{name} \\
& \quad \quad \quad \textbf{then} \ \textbf{exit}(1) \ \textbf{fi} \ \textbf{od}; \\
& \quad \textbf{if} \ \text{total} \ \neq \ 0 \\
& \quad \quad \textbf{then} \quad \text{!P write(\text{records}[(i - 1)].\text{name}, \text{total} \ \text{var });} \\
& \quad \quad \quad \ \text{changed} \ := \ (\text{changed} + 1) \ \textbf{fi} \ \textbf{od}; \\
& \quad \text{!P write(“Changed items:”, \ \text{changed} \ \text{var }) \ \textbf{end}}
\end{align*}
\]

Convert the inner loop to a \textbf{while} loop by duplicating code to move the test to the beginning.
Third Stage: Restructure and Simplify

The restructured abstract program:

```plaintext
var ⟨i := 0, changed := 0⟩ :
  !P write("Management Report... " var );
i := 1;
while i ≤ ℓ(records) do
  total := records[i].number;
i := (i + 1);
  while records[(i − 1)].name = records[i].name ∧ i ≤ ℓ(records) do
    total := (total + records[i].number);
i := (i + 1) od;
if total ≠ 0
  then !P write(records[(i − 1)].name, total var );
    changed := (changed + 1) fi od;
!P write("Changed items:", changed var ) end
```
Fourth Stage: Specification Level
Fourth Stage: Specification Level

\[
\text{proc } \text{Management\_Report } \equiv \\
\begin{align*}
\text{begin} \\
&P \text{ write(“Management Report...”);} \\
&\text{var} \langle q := \text{split(records, same\_name?)} \rangle : \\
&\quad q := \text{summarise } \ast q; \\
&\quad q := \text{filter}(q, \text{change?}); \\
&P \text{ write } \ast q; \\
&P \text{ write(“Changed items:”, } \ell(q) \text{) end}
\end{align*}
\]

where

\[
\begin{align*}
\text{funct } \text{same\_name?}(x, y) & \equiv \\
& x.\text{name} = y.\text{name}. \\
\text{funct } \text{summarise}(g) & \equiv \\
& \langle g[1].\text{name}, +/(.\text{number} \ast g) \rangle. \\
\text{funct } \text{change?}(a, b) & \equiv \\
& b \neq 0.
\end{align*}
\]
A Method For Reverse Engineering

1. Establish the reverse engineering environment
2. Collect the software to be reverse engineered
3. Produce a high-level description of the system
4. Translate the source code into WSL
5. “Inverse Engineering”, i.e. reverse engineering through formal transformations. We do this by iterating over the four steps on the next slide.
6. Acceptance test: We now have a high-level specification of the whole system which should go through the usual Q.A. and acceptance tests
A Method For Reverse Engineering

The Inverse Engineering Stage: Iterate over the following four steps:

1. Restructuring transformations

2. Analyse the resulting structures to determine suitable higher-level data representations and control structures

3. Redocument: record the discoveries made so far and any other useful information about the code and its data structures

4. Implement the higher-level data representations and control structures using suitable transformations
Inverse Engineering

Inverse engineering is the process of extracting high-level abstract specifications from source code using program transformations. This is important in the following areas:

- Specifications are more compact and expressed in a problem-oriented notation
- Specifications are easier to understand, modify and enhance than source code
- Increases the programmer’s understanding of the program
- Translation between programming languages becomes possible
Inverse Engineering

Inverse engineering is the process of extracting high-level abstract specifications from source code using program transformations. This is important in the following areas:

- The transformations are proved to be correct: this allows a high degree of confidence to be placed in the resulting specifications
- Errors and inefficiencies are exposed and easily corrected
- Executable code can be generated automatically, or semi-automatically from the specifications
Any practical program transformation system for reverse engineering has to meet the following requirements:

- It has to be able to cope with all the usual programming constructs: loops with exits from the middle, `gotos`, recursion etc.

- It cannot be assumed that the code was developed (or maintained) according to a particular programming method: real code ("warts and all") must be acceptable to the system.

- Significant restructuring may be required before the real reverse engineering can take place, and it is important that this restructuring can be carried out automatically.
Any practical program transformation system for reverse engineering has to meet the following requirements:

- It should be based on a formal language and formal transformation theory, so that it is possible to prove that all the transformations used are semantic-preserving.

- The formal language should ideally be a wide spectrum language which can cope with both low-level constructs such as `gotos`, and high-level constructs, including nonexecutable specifications.

- Translators are required from the source language(s) to the formal language.
Tool Requirements (continued)

- It must be possible to apply transformations without needing to understand the program first.

- It must be possible to extract a module, or smaller component, from the system for analysis and transformation, with the transformations guaranteed to preserve all the interactions of that component with the rest of the system.

- An extensive catalogue of proven transformations is required, with mechanically checkable correctness conditions and some means of composing transformations to develop new ones.
An interactive interface which pretty-prints each version on the display will allow the user to instantly see the structure of the program from the indentation structure.

The correctness of the transformation system itself must be well-established, since all results depend of the transformations being implemented correctly.

A method for reverse engineering by program transformation needs to be developed alongside the transformation system.
Features of FermaT

- Source code is translated into WSL, then automatically restructured and simplified
- Transformations are written in $\text{MetaWSL}$
- The tool validates transformation choice and offers a menu of valid transformations according to the context
- A transformation engine carries out the transformations and records the history
- Documentation and comments can be attached to the code
- Edits and modifications are recorded in the history
- A front end displays a pretty-printed version of the current program
- The system calculates various metrics (McCabe, structural complexity, size) to monitor progress and quality
The Wide Spectrum Language WSL

- A *formal* language with supporting formal development method
- Includes both low-level programming constructs and high-level specifications within one language
- Refinement of specifications into programs and reverse engineering of programs into specifications can be carried out within a single language
- Program transformations have been developed to extract specifications from source code
- Defined in terms of an imperative kernel language
The Wide Spectrum Language WSL

- Extended by definitional transformations into a practical programming language
- Automatic translators have been developed to translate programs from other programming languages into WSL
- Translators from IBM Assembler, JOVIAL and a propriety 16 bit assembler to WSL, and translators from WSL to COBOL and C have been developed and used successfully
- A practical program transformation system, called FermaT, has been developed based on WSL and the transformation theory
Why invent another language?

Why not ADA, . . . or C, . . . or YFL?

- Simple semantics with tractable reasoning methods
- Specifications and low-level programming constructs
- Results are language independent
- Existing programming languages were not designed to be transformable
- All existing programming languages have limitations
Problems with Existing Languages

- **Differences between compilers.** Some transformations will be valid for one compiler, but not for another. In practice, this would mean our transformation system could only be used with one of the many incompatible versions of the language;

- **Side effects in expressions.** For example:

\[
y := f(x) + f(x) \approx y := 2.f(x)
\]

is a valid WSL transformation. But in a language with side effects, we would need to check the definition of \( f(x) \), and everything it calls, for possible non-idempotent side-effects;
Problems with Existing Languages

Variable Aliases. For example:

\[ \begin{align*}
  x &:= 1; \quad y := 2 \quad \approx \quad y := 2; \quad x := 1 \\
\end{align*} \]

is a valid WSL transformation. But in a language with the possibility of aliasing, \( x \) and \( y \) may refer to the same memory location. In this case, the transformation is invalid. We would need to determine if the two variables were aliased: but in the general case, this is a non-computable problem.
Problems with Existing Languages

The Replacement Property. For example:

\[
x := x + 1; \quad x := 2 \times x \quad \approx \quad x := 2 \times (x + 1)
\]

is valid in WSL but not in C.
The Replacement Property. For example:

\[ x := x + 1; \quad x := 2 \times x \quad \approx \quad x := 2 \times (x + 1) \]

is valid in WSL but not in C. Consider the C statements:

```c
if (y == 0)
    x = 2*(x+1);
```

and:

```c
if (y == 0)
    x = x + 1; \quad x = 2\times x;
```

Are they equivalent?
Program Transformation Applications

- Deriving algorithms in a systematic way from their specifications
- Improving the efficiency of programs
- Deriving the specification of an unstructured program from the source code ("Inverse Engineering")
- Discovering bugs in a program by attempting to transform it into a specification
- Restructuring "spaghetti" Assembler programs into a hierarchy of self-contained modules.
Benefits of Formal Transformations

- The transformations used in inverse engineering have been proved correct according to the formal method.
- Large restructuring changes can be made to the program with the confidence that the functionality is unchanged.
- During the inverse engineering process, bugs and inconsistencies are revealed. This leads to increased reliability.
- The formal links between specification and code can be recorded and kept up to date.
- Maintenance can be carried out at the specification level.
- Programs can be re-expressed in problem-oriented notation.
- Programs can be incrementally improved—instead of being incrementally degraded!
Modelling Assembler in WSL

Our approach involves three types of modelling:

1. Complete model: Each assembler instruction is translated into WSL statements which capture all the effects of the instruction, including condition codes and registers;

2. Partial model: Branches to register are modelled by attempting to determine all possible targets of such a branch, associating a value with each target, and calling a “dispatch” routine which finds the target for the given value;

3. Self-modifying code: Some cases are detected and handled (overwriting a NOP/branch, modifying a length field etc.) but general self-modifying code require human intervention: usually to renovate the assembler using more standard programming practices!
## Typical Case Study Results

Typical results from a case study of a 442 line IBM Assembler module, taken from a large commercial system. In this case study, no manual transformations were required to get a compilable C program.

<table>
<thead>
<tr>
<th>Stage</th>
<th>No. of Statements</th>
<th>McCabe Cyclomatic</th>
<th>Control Flow /Data Flow</th>
<th>Branch /Loop</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>958</td>
<td>133</td>
<td>806</td>
<td>405</td>
<td>10,449</td>
</tr>
<tr>
<td>Data tr.</td>
<td>916</td>
<td>107</td>
<td>688</td>
<td>335</td>
<td>6,856</td>
</tr>
<tr>
<td>Fix Assem</td>
<td>336</td>
<td>18</td>
<td>222</td>
<td>20</td>
<td>2,059</td>
</tr>
</tbody>
</table>
Metrics

**No. of Statements** is the number of executable statements in the parse tree

**McCabe Cyclomatic** is the usual McCabe cyclomatic complexity

**Control Flow/Data Flow** counts the number of control flow lines and data flow lines

**Branch/Loop** is a metric which counts the size of loops

**Structural** is a metric which gives a weighted sum of the structural features of the program.
Call Graph: Before
Call Graph: After

Diagram showing a call graph with nodes labeled "main", "DRUCK", and "KOPF1". There are directed edges from "main" to "DRUCK" and from "DRUCK" to "KOPF1".
More Case Study Results

Four IBM Assembler modules containing up to 4,000 lines of source code:

<table>
<thead>
<tr>
<th>Stage</th>
<th>No. of Statements</th>
<th>McCabe Cyclomatic</th>
<th>Control Flow/Data Flow</th>
<th>Branch/Loop</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMD-1</td>
<td>7,704</td>
<td>1,949</td>
<td>5,945</td>
<td>2,412</td>
<td>83,535</td>
</tr>
<tr>
<td>FMD-6</td>
<td>2,042</td>
<td>33</td>
<td>319</td>
<td>35</td>
<td>11,017</td>
</tr>
<tr>
<td>RTM-1</td>
<td>24,178</td>
<td>10,110</td>
<td>23,965</td>
<td>5,337</td>
<td>329,372</td>
</tr>
<tr>
<td>RTM-6</td>
<td>4,533</td>
<td>1,031</td>
<td>4,748</td>
<td>439</td>
<td>31,316</td>
</tr>
<tr>
<td>TLO-1</td>
<td>4,240</td>
<td>372</td>
<td>2,235</td>
<td>1,780</td>
<td>35,174</td>
</tr>
<tr>
<td>TLO-6</td>
<td>101</td>
<td>4</td>
<td>73</td>
<td>1</td>
<td>746</td>
</tr>
<tr>
<td>TSM-1</td>
<td>21,457</td>
<td>8,149</td>
<td>20,443</td>
<td>5,184</td>
<td>262,223</td>
</tr>
<tr>
<td>TSM-6</td>
<td>3,173</td>
<td>492</td>
<td>2,356</td>
<td>281</td>
<td>20,005</td>
</tr>
</tbody>
</table>
More Case Study Results

Two IBM Assembler modules, 1,452 and 6,361 source lines:

<table>
<thead>
<tr>
<th>Stage</th>
<th>No. of Statements</th>
<th>McCabe Cyclomatic</th>
<th>Control Flow/Data Flow</th>
<th>Branch/Loop</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS0022c-1</td>
<td>6,183</td>
<td>2,518</td>
<td>6,086</td>
<td>1,563</td>
<td>77,842</td>
</tr>
<tr>
<td>SAS0022c-6</td>
<td>1,313</td>
<td>169</td>
<td>980</td>
<td>93</td>
<td>8,867</td>
</tr>
<tr>
<td>SAS002c-1</td>
<td>35,483</td>
<td>16,876</td>
<td>39,460</td>
<td>7,424</td>
<td>481,173</td>
</tr>
<tr>
<td>SAS002c-6</td>
<td>11,633</td>
<td>2,130</td>
<td>11,355</td>
<td>2,082</td>
<td>84,251</td>
</tr>
</tbody>
</table>
## Metrics From Migration Projects

<table>
<thead>
<tr>
<th>Org</th>
<th>Raw WSL McCabe</th>
<th>Restructured WSL</th>
<th>Assembler McCabe</th>
<th>COBOL McCabe</th>
<th>Data Access bugs/MLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,605</td>
<td>467</td>
<td>651</td>
<td>283</td>
<td>550</td>
</tr>
<tr>
<td>B</td>
<td>245</td>
<td>38</td>
<td>70</td>
<td>33</td>
<td>274</td>
</tr>
<tr>
<td>C</td>
<td>392</td>
<td>52</td>
<td>96</td>
<td>27</td>
<td>302</td>
</tr>
</tbody>
</table>
## Metrics From Migration Projects

<table>
<thead>
<tr>
<th>Org</th>
<th>Complex Subr Linkage</th>
<th>EXecute Instr/MLOC</th>
<th>Self-Modifying Code/MLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>73.6%</td>
<td>24</td>
<td>2,347</td>
</tr>
<tr>
<td>B</td>
<td>36.9%</td>
<td>745</td>
<td>590</td>
</tr>
<tr>
<td>C</td>
<td>53.0%</td>
<td>1,169</td>
<td>127</td>
</tr>
</tbody>
</table>
For any kernel language statement \( S : V \rightarrow W \), and formula \( R \) whose free variables are all in \( W \), we define \( \text{WP}(S, R) \) as follows:

1. \( \text{WP}(\{P\}, R) =_{\text{DF}} P \land R \)
2. \( \text{WP}([Q], R) =_{\text{DF}} Q \Rightarrow R \)
3. \( \text{WP}(\text{add}(x), R) =_{\text{DF}} \forall x. R \)
4. \( \text{WP}(\text{remove}(x), R) =_{\text{DF}} R \)
5. \( \text{WP}((S_1 ; S_2), R) =_{\text{DF}} \text{WP}(S_1, \text{WP}(S_2, R)) \)
6. \( \text{WP}((S_1 \sqcap S_2), R) =_{\text{DF}} \text{WP}(S_1, R) \land \text{WP}(S_2, R) \)
7. \( \text{WP}((\mu X.S), R) =_{\text{DF}} \bigvee_{n<\omega} \text{WP}((\mu X.S)^n, R) \)

where \((\mu X.S)^0 = \text{abort}\) and \((\mu X.S)^{n+1} = S[(\mu X.S)^n/X]\) which is \( S \) with all occurrences of \( X \) replaced by \((\mu X.S)^n\).
**Proof Theoretic Refinement**

Proof theoretic refinement is defined from the weakest precondition formula $\text{WP}$, applied to the special postcondition $x \neq x'$ where $x$ is a list of all the variables assigned in either statement, and $x'$ is a list of new variables.

If $S, S' : V \rightarrow W$ have no free statement variables and $x$ is a sequence of all variables assigned to in either $S$ or $S'$, and the formulae

$$\text{WP}(S, x \neq x') \Rightarrow \text{WP}(S', x \neq x')$$

and

$$\text{WP}(S, \text{true}) \Rightarrow \text{WP}(S', \text{true})$$

are provable from the set $\Delta$ of sentences, then we say that $S$ is refined by $S'$ and write:

$$\Delta \vdash S \leq S'$$
The FermaT Transformation System is available under the GNU GPL (General Public Licence) from the following web site:

http://www.cse.dmu.ac.uk/~mward/fermat.html
http://www.gkc.org.uk/fermat.html