Formal Transformations and WSL

Part Three

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

$\mathcal{META}$WSL

- Transformations are implemented in an extension of WSL, called $\mathcal{META}$WSL.
- WSL has been developed specifically to be a powerful programming language which is easy to transform.
- $\mathcal{META}$WSL has been developed as a language in which it is easy to implement program transformations.

$\mathcal{META}$WSL

For example:

ifmatch Statement **if** $\sim$?**B** **then** $\sim$?$\mathbf{S}_1$ **else** $\sim$?$\mathbf{S}_2$ **fi**

then **B** := @Not(**B**);

@Paste_Over(**fill** Statement

if $\sim$?**B** **then** $\sim$?$\mathbf{S}_2$ **else** $\sim$?$\mathbf{S}_1$ **fi** **endfill**) **endmatch**

in ASCII form this is:

```
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:

$$\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

If it does, then new local variables `B`, `S1` and `S2` are created. `B` contains the condition from the current **if** statement, `S1` contains the statement sequence from the **then** part and `S2` 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.

$\mathcal{META}$WSL: Extensions to WSL

Moving around the abstract syntax tree:

$\mathcal{META}$WSL: 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)

$\mathcal{META}$WSL: 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:

$\mathcal{META}$WSL: 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

$\mathcal{META}$WSL: 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.)

$\mathcal{META}$WSL: 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.)

$\mathcal{META}$WSL: 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`)

$\mathcal{META}$WSL: 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.

$\mathcal{META}$WSL: 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.

$\mathcal{METAWSL}$ Editing Procedures

Editing the tree:

$\mathcal{META}$WSL Editing Procedures

Editing the tree:

- @Delete, @Clever_Delete, @Cut,

$\mathcal{META}$WSL **Editing Procedures**

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

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.

$\mathcal{META}$WSL **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.

$\mathcal{META}$WSL **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:

$$\text{@Paste\_Over}(\text{@Make}(\texttt{T\_Number}, \text{@V}(\text{@I}) + 1, \langle\rangle)$$

$\mathcal{META}$WSL **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

$\mathcal{META}$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

$\mathcal{META}$WSL **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. ~?x matches any item and puts the matched result into variable x;
2. ~*x matches a sequence of zero or more items and puts the result into x;
3. ~=(e) matches the current item against the value of the expression e.

$\mathcal{META}$WSL **Pattern Matching**

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() $\equiv$

  if `Skip` $\in$ @Stat_Types(@I)

    then @Pass

    else @Fail(“`No ‘SKIP’ statements to delete.`”) **fi**.;

proc @Delete_All_Skips_Code(Data) $\equiv$

  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

Before	After
while B do skip od	$\{\neg$**B**$\}$
if B then skip **else** $x := 0$ **fi**	**if** $\neg$**B then** $x := 0$ **fi**
do skip od	**abort**
var $\langle x := 0 \rangle$: **if B then skip fi end**; $y := 0$	$y := 0$

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

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** $\{y \leqslant 0\}$ **fi**

Meta-Transformations

Part of the source code for WSL to 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:

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

is transformed into:

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:

foreach Statements **do**
    if Depth $= 1$
        then @Down_Last;
                do if @Right? $\wedge$ $\neg$@Is_Proper?
                        then $N := 0$;
                                ateach STS **do**
                                    if Depth $\in$ @Gen_TVs(@I, 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 $\mathcal{META}$WSL

Part of the Absorb_right transformation:

@Right; @Cut; @Left;

. . .

foreach STS **do**

  if Depth $= 0 \vee$ (@ST(@I) $=$ `Exit` $\wedge$ @V(@I) $=$ Depth)

    then if @ST(@I) $=$ `Exit` $\wedge$ Depth > 0

        then @Splice_Over(@Increment(@Buffer,

                AS_Type, Depth, 0))

      elsif @ST(@I) $=$ `Skip`

        then @Paste_Over(@Buffer)

      elsif @ST(@I) $=$ `Exit` $\wedge$ Depth $= 0$

        $\vee$ @Gen_Improper?(@I, AS)

        then skip

      elsif @ST(@I) $=$ `Call` $\wedge$ @V(@I) $=$ "Z"

        then skip

        else @Paste_After(@Buffer) **fi fi od**;

Meta-Transformations

Each time FermaT is rebuilt from source, the Floop_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.

Expression/Condition Simplifier

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 $(\sim ?y * \sim ?x)$ div $\sim ?x$
    then @Paste_Over(y) **endmatch**;

  ifmatch Expression $(\sim ?y * \sim ?x + \sim ?z)$ div $\sim ?x$
    then @Paste_Over(**fill** Expression
      $\sim ?y + (\sim ?z$ div $\sim ?x)$ **endfill**) **endmatch**;

  ...
od;

Expression/Condition Simplifier

foreach Condition **do**

  ifmatch Condition $\sim ?x < \sim ?y \vee \sim ?y < \sim ?x$

    then @Paste_Over(**fill** Condition $\sim ?x \neq \sim ?y$ **endfill**) **endmatch**;

  ifmatch Condition $\sim ?z < \sim ?x \vee \sim ?x \leqslant \sim ?y \vee \sim ?y \leqslant \sim ?z$

    then @Paste_Over(Mth_True) **endmatch**;

  ifmatch Condition $\sim ?y < \sim ?x \wedge \sim ?z < \sim ?y \wedge \sim ?x \leqslant \sim ?z$

    then @Paste_Over(Mth_False) **endmatch**;

  . . .

od;

Expression/Condition Simplifier

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

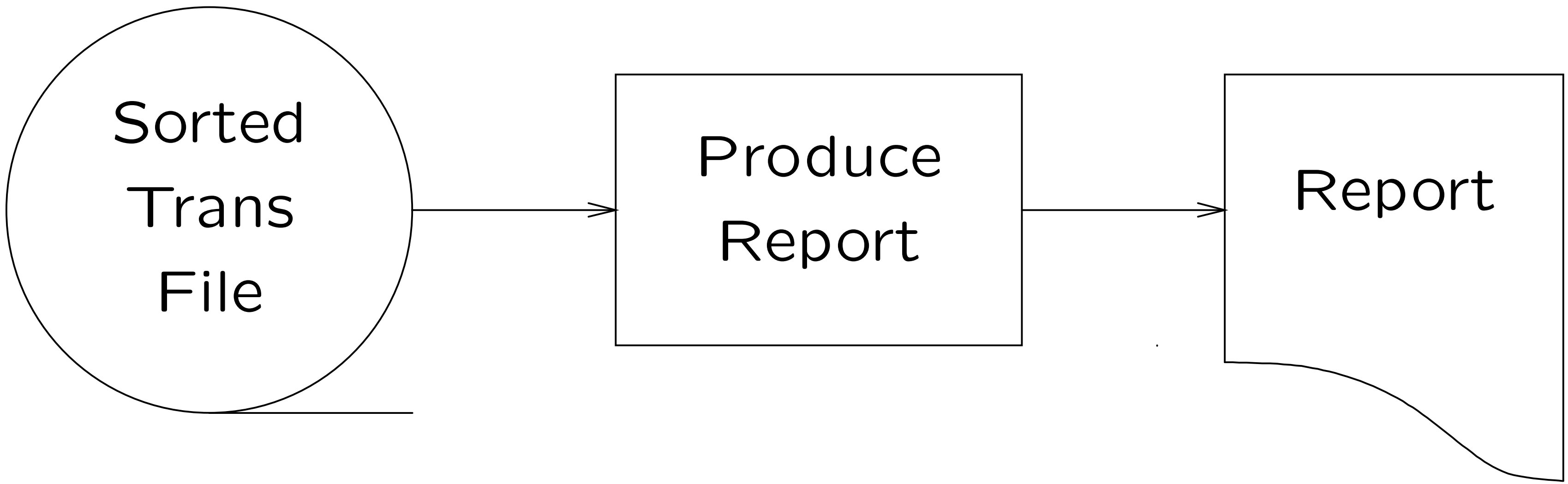
Bolt	+200
Bolt	−150
Bolt	−25
Nut	+100
Nut	−100
Wheel	+40
Wheel	−10
Widget	−500

Management Report

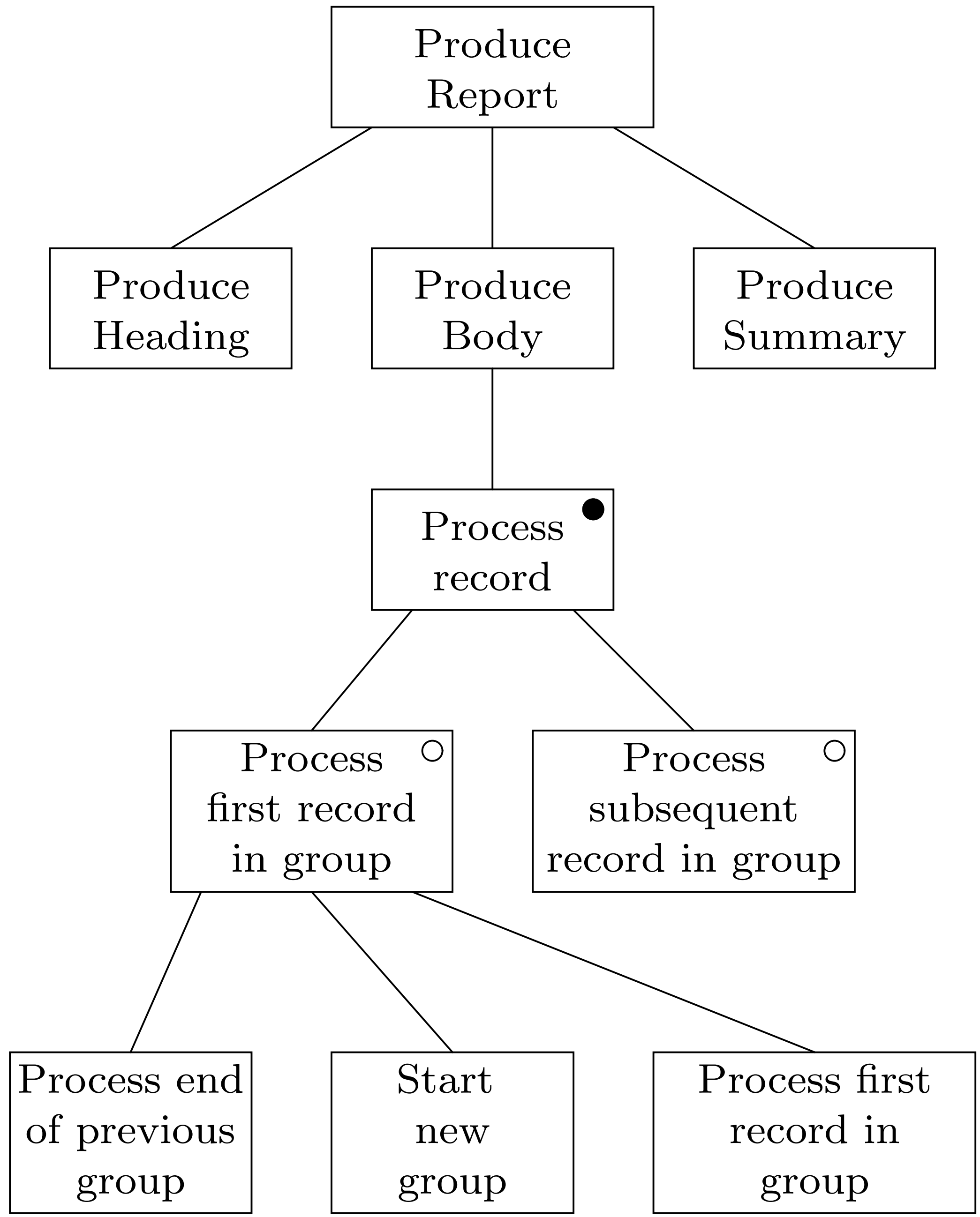
ITEM	Net Change
Bolt	+25
Wheel	+30
Widget	−500

Number Changed: 3

Design Phase



Design Phase



Final Functional Decomposition

```
proc Management_Report ≡
   Produce_Heading;
   read(stuff);
   while NOT eof(stuff) do
      if First_Record_In_Group
         then Process_End_Of_Previous_Group;
              Process_Start_Of_New_Group;
              Process_Record
         else
            Process_Record
      fi;
      read(stuff)
   od;
   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

```
proc Management_Report ≡
 var ⟨SW1 := 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
      fi;
      read(stuff)
    od;
    Produce_Summary
  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

```
proc Management_Report ≡
  var ⟨SW1 := 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
      fi;
      read(stuff)
    od;
    Process_End_Of_Last_Group;
    Produce_Summary
  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.

Third Quick Fix

```
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!

Problem

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.

Fourth Quick Fix

```
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 ⟨SW1 := 0, SW2 := 0⟩ :
   !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 Constant_Propagation and Remove_Redundant_Vars to remove SW2:

```
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:

```
!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:

```
!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
             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 Floop_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

Concrete Call	Abstract Code
Produce_Heading	!P write(“`Management Report...`”)
read	last := record; $i := i + 1$; record := records$[i]$
eof?(stuff)	$i > \ell$(records)
Process_Start_Of_Group	total := 0
Process_Record	total := total + record.number
First_Record_In_Group?	last.name $\neq$ record.name
Process_End_Of_Group	**if** total $\neq 0$ **then** write(last.name, total); changed := changed + 1 **fi**
Produce_Summary	!P write(“`Changed items:`” , changed)

Second Stage: Abstract Data Types

var $\langle i := 0, \text{last} := \text{``''}, \text{record} := \text{``''}, \text{changed} := 0\rangle$:
  !P write("`Management Report...`");
  last := record; $i := i + 1$; record := records$[i]$;
  while $i \leqslant \ell(\text{records})$ **do**
    total := 0;
    do total := total + record.number;
      last := record; $i := i + 1$; record := records$[i]$;
      if $i > \ell(\text{records}) \vee \text{last.name} \neq \text{record.name}$
        then **exit** **fi** **od**;
    if total $\neq 0$
      then !P write(last.name, total);
        changed := changed + 1 **fi** **od**;
  !P write("`Changed items:`", changed) **end**

Third Stage: Restructure and Simplify

First, we can replace record by records[i] throughout:

$$
\begin{array}{l}
\textbf{var}\ \langle i := 0, \mathsf{last} := \text{``''}, \mathsf{changed} := 0\rangle : \\
\quad \mathsf{!P\ write}(\text{``}\mathtt{Management\ Report...}\text{''}); \\
\quad \mathsf{last} := \mathsf{records}[i];\ i := i + 1; \\
\quad \textbf{while}\ i \leqslant \ell(\mathsf{records})\ \textbf{do} \\
\qquad \mathsf{total} := 0; \\
\qquad \textbf{do}\ \mathsf{total} := \mathsf{total} + \mathsf{records}[i].\mathsf{number}; \\
\qquad\quad \mathsf{last} := \mathsf{records}[i];\ i := i + 1;\ ; \\
\qquad\quad \textbf{if}\ i > \ell(\mathsf{records})\ \vee\ \mathsf{last.name} \neq \mathsf{records}[i].\mathsf{name} \\
\qquad\qquad \textbf{then exit fi od}; \\
\qquad \textbf{if}\ \mathsf{total} \neq 0 \\
\qquad\quad \textbf{then}\ \mathsf{write}(\mathsf{last.name}, \mathsf{total}); \\
\qquad\qquad\quad \mathsf{changed} := \mathsf{changed} + 1\ \textbf{fi od}; \\
\quad \mathsf{!P\ write}(\text{``}\mathtt{Changed\ items:}\text{''}, \mathsf{changed})\ \textbf{end}
\end{array}
$$

Now move the assignments to last forwards, and replace this variable by its value.

Third Stage: Restructure and Simplify

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

Convert the inner loop to a **while** loop by duplicating code to move the test to the beginning.

Third Stage: Restructure and Simplify

The restructured abstract program:

var $\langle i := 0, \text{changed} := 0\rangle$:
  !P write("`Management Report...` " **var**);
  $i := 1$;
  while $i \leqslant \ell(\text{records})$ **do**
    total $:=$ records$[i]$.number;
    $i := (i + 1)$;
    while records$[(i-1)]$.name $=$ records$[i]$.name $\wedge\ i \leqslant \ell(\text{records})$ **do**
      total $:=$ (total $+$ records$[i]$.number);
      $i := (i + 1)$ **od**;
    if total $\neq 0$
      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

```
proc Management_Report ≡
  begin
    !P write("Management Report...");
    var ⟨q := split(records, same_name?)⟩ :
      q := summarise * q;
      q := filter(q, change?);
      !P write * q;
      !P write("Changed items:", ℓ(q)) end
  where
  funct same_name?(x, y) ≡
    x.name = y.name.
  funct summarise(g) ≡
    ⟨g[1].name, +/(.number * g)⟩.
  funct change?(a, b) ≡
    b ≠ 0.
  end
```

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

Tool Requirements

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, **goto**s, 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

Tool Requirements (continued)

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 **goto**s, 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

Tool Requirements (continued)

- 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 $\mathcal{META}$WSL
- 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:

  $$x := 1;\ y := 2 \ \approx \ y := 2;\ x := 1$$

  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:

$$x := x + 1;\ x := 2 * x \ \approx\ x := 2 * (x + 1)$$

is valid in WSL but not in C.

Problems with Existing Languages

- **The Replacement Property.** For example:

$$x := x + 1;\ x := 2 * x \ \approx\ x := 2 * (x + 1)$$

  is valid in WSL but not in C.

  Consider the C statements:

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

  and:

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

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

Metrics

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

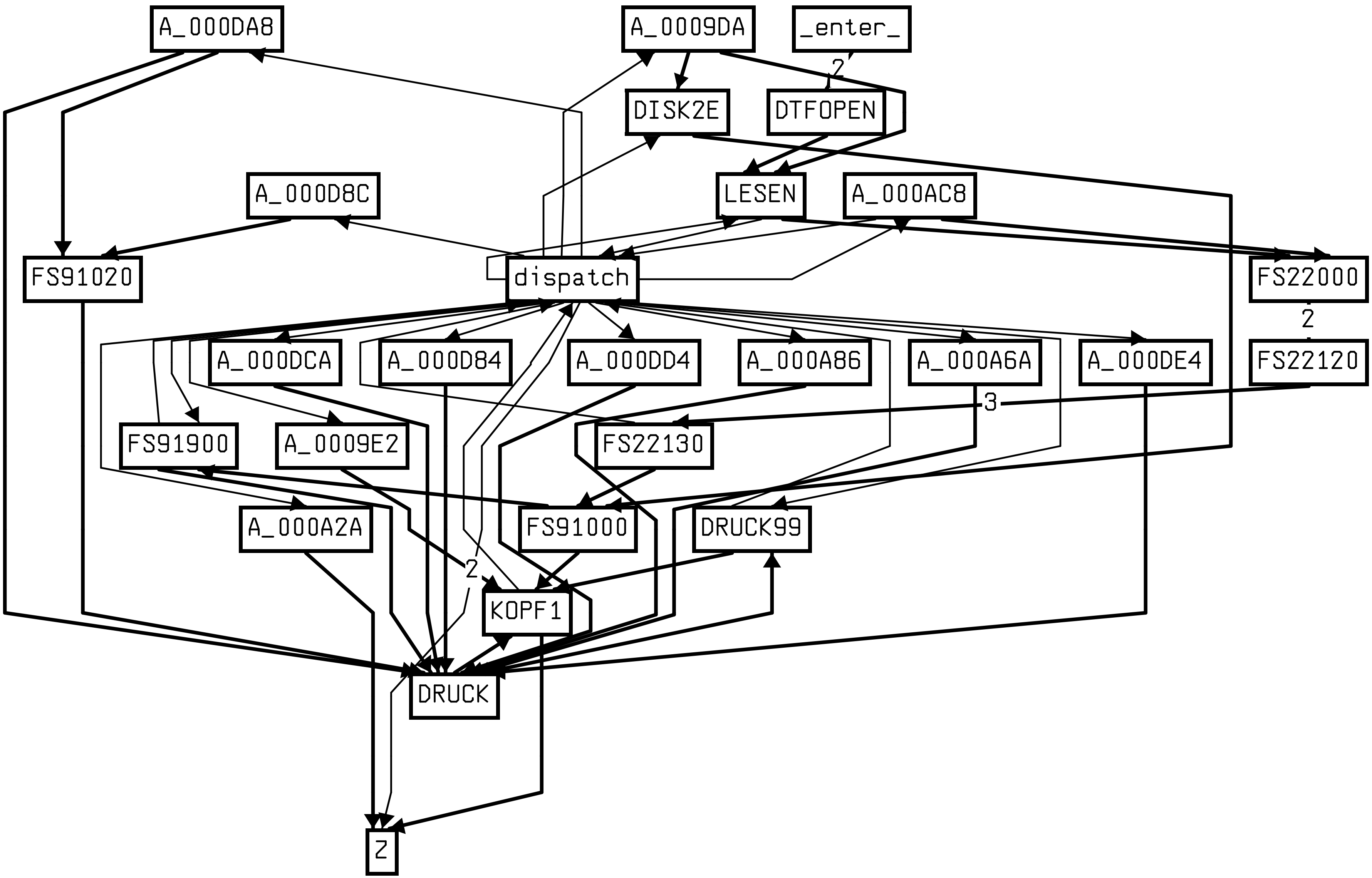
McCabe Cyclomatic is the usual McCabe cyclomatic complexity

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

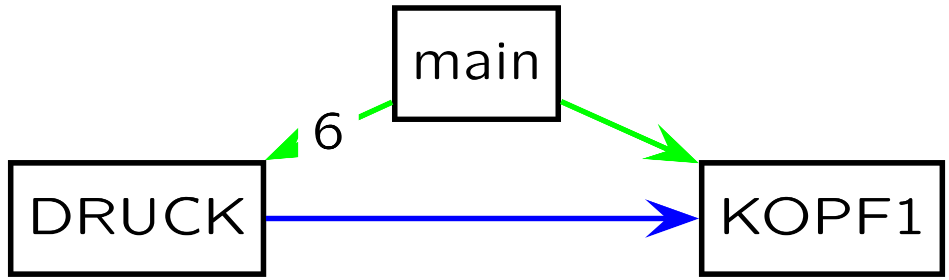
Branch/Loop is a metric which counts the size of loops

Structural is a metric which gives a weighted sum of the structural features of the program.

Call Graph: Before



Call Graph: After



More Case Study Results

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

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

More Case Study Results

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

Stage	No. of Statements	McCabe Cyclomatic	Control Flow /Data Flow	Branch /Loop	Structural
SAS0022c-1	6,183	2,518	6,086	1,563	77,842
SAS0022c-6	1,313	169	980	93	8,867
SAS002c-1	35,483	16,876	39,460	7,424	481,173
SAS002c-6	11,633	2,130	11,355	2,082	84,251

Metrics From Migration Projects

Org	Raw WSL McCabe	Restructured WSL	Assembler McCabe	COBOL McCabe	Data Access bugs/MLOC
A	1,605	467	651	283	550
B	245	38	70	33	274
C	392	52	96	27	302

Metrics From Migration Projects

Org	Complex Subr Linkage	EXecute Instr/MLOC	Self-Modifying Code/MLOC
A	73.6%	24	2,347
B	36.9%	745	590
C	53.0%	1,169	127

Weakest Preconditions

For any kernel language statement $\mathbf{S}\colon V \to W$, and formula $\mathbf{R}$ whose free variables are all in W, we define $\mathsf{WP}(\mathbf{S}, \mathbf{R})$ as follows:

1. $\mathsf{WP}(\{\mathbf{P}\}, \mathbf{R}) \ =_{\text{DF}}\ \mathbf{P} \wedge \mathbf{R}$
2. $\mathsf{WP}([\mathbf{Q}], \mathbf{R}) \ =_{\text{DF}}\ \mathbf{Q} \Rightarrow \mathbf{R}$
3. $\mathsf{WP}(\mathsf{add}(\mathbf{x}), \mathbf{R}) \ =_{\text{DF}}\ \forall \mathbf{x}.\, \mathbf{R}$
4. $\mathsf{WP}(\mathsf{remove}(\mathbf{x}), \mathbf{R}) \ =_{\text{DF}}\ \mathbf{R}$
5. $\mathsf{WP}((\mathbf{S}_1;\ \mathbf{S}_2), \mathbf{R}) \ =_{\text{DF}}\ \mathsf{WP}(\mathbf{S}_1, \mathsf{WP}(\mathbf{S}_2, \mathbf{R}))$
6. $\mathsf{WP}((\mathbf{S}_1 \ \sqcap \ \mathbf{S}_2), \mathbf{R}) \ =_{\text{DF}}\ \mathsf{WP}(\mathbf{S}_1, \mathbf{R}) \wedge \mathsf{WP}(\mathbf{S}_2, \mathbf{R})$
7. $\mathsf{WP}((\mu X.\mathbf{S}), \mathbf{R}) \ =_{\text{DF}}\ \bigvee_{n<\omega} \mathsf{WP}((\mu X.\mathbf{S})^n, \mathbf{R})$

where $(\mu X.\mathbf{S})^0 = \textbf{abort}$ and $(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n / X]$ which is $\mathbf{S}$ with all occurrences of X replaced by $(\mu X.\mathbf{S})^n$.

Proof Theoretic Refinement

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

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

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

and

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

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

$$\Delta \vdash \ \mathbf{S} \ \leq \ \mathbf{S}'$$

FermaT

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