Restructuring and Destructuring

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

Software Technology Research Lab
De Montfort University
Restructuring Transformations

An unstructured program (action system) can be made more structured using these transformations:

- **Expand Call**: Replace an action call by a copy of the action body

- **Substitute and Delete**: Apply Expand_Call to all the calls of the selected action, and then delete the action (provided the action does not call itself!)

- **Remove Recursion in Action**: In a regular action system, an action which calls itself can be transformed into an action which does not call itself by introducing loops

- **Floop to While**: a suitable Floop can be transformed directly to a while loop. In the general case, a flag may be needed.
Restructuring Transformations

- **Merge Calls in Action:** Attempt to merge two or more calls to the same action into a single call.

- **Delete Rest:** In a regular action system, no action call can return, so all the rest of the statements after an action call can be deleted.

- **Delete Item:** An action which is never called is “dead code” and can be deleted.

- **Simplify Action System:** Applies the above transformations to simplify an action system as much as possible.

- **Simplify Item:** An action system containing a single action can be converted to a loop.

The next few slides illustrate each of these transformations.
inhere \equiv \text{inhere( var ); } \textbf{call more end} \\
more \equiv \text{if } m = 1 \\
\hspace{1cm} \textbf{then } p := \text{number}[i]; \text{ line := line } \text{+ } \text{“, ” } \text{+ } p \text{ fi; } \\
\hspace{1cm} \text{last := item}[i]; \textbf{call l end}

becomes:
Expand Call

\[
\text{inhere} \equiv \text{inhere( } \text{var } \); \textbf{call more} \text{ end}
\]
\[
\text{more} \equiv \textbf{if } m = 1 \\
\hspace{1cm} \textbf{then } p := \text{number}[i]; \text{ line } := \text{line } + \text{ "", " } + p \text{ fi;}
\hspace{1cm} \text{last } := \text{item}[i]; \textbf{call l end}
\]

becomes:
\[
\text{inhere} \equiv \text{inhere( } \text{var } \); \\
\hspace{1cm} \textbf{if } m = 1 \\
\hspace{2cm} \textbf{then } p := \text{number}[i]; \text{ line } := \text{line } + \text{ "", " } + p \text{ fi;}
\hspace{2cm} \text{last } := \text{item}[i]; \textbf{call l end}
\]
\[
\text{more} \equiv \textbf{if } m = 1 \\
\hspace{1cm} \textbf{then } p := \text{number}[i]; \text{ line } := \text{line } + \text{ "", " } + p \text{ fi;}
\hspace{1cm} \text{last } := \text{item}[i]; \textbf{call l end}
\]

If this was the only call to more, then the action can be deleted.
Substitute and Delete

If an action does not call itself, then Substitute_and_Delete applies Expand_Call to each call of the action, and then deletes the action.
more \equiv \textbf{if} \ m = 1 \ \textbf{then} \ p := \text{number}[i];
\text{line} := \text{line} + "\ , \" + p \ \textbf{fi};
\text{last} := \text{item}[i];
i := i + 1;
\textbf{if} \ i = (n + 1) \ \textbf{then} \ \text{call} \ \text{alldone} \ \textbf{fi};
p_{-1}(\textbf{var});
\boxed{\text{call more}} \ \textbf{end}

becomes:
Remove Recursion in Action

more ≡ if \( m = 1 \) then \( p := \text{number}[i] \);

\[
\text{line} := \text{line} + "", " \; + p \; \text{fi} ;
\]

last := \text{item}[i];

\( i := i + 1 ; \)

if \( i = (n + 1) \) then call alldone fi;

\( p_{-1}(\text{var}); \)

\textbf{call more} end

becomes:

more ≡ \textbf{do} if \( m = 1 \) then \( p := \text{number}[i] \);

\[
\text{line} := \text{line} + "", " \; + p \; \text{fi} ;
\]

last := \text{item}[i];

\( i := i + 1 ; \)

if \( i = (n + 1) \) then call alldone fi;

\( p_{-1}(\text{var}) \) od end
Sometimes a double loop is needed.

more ≡ \( i := i + 1; \)

\[
\begin{align*}
& \text{if } i < n + 1 \text{ then call more } \\
& \text{elsif } B1? (i) \text{ then } p_1( \text{ var } ) \\
& \text{elsif } B2? (i) \text{ then call more } \text{ fi; } \\
& p_3( \text{ var } ); \\
& \text{call alldone} \text{ end}
\end{align*}
\]
Sometimes a double loop is needed.

more $\equiv i := i + 1$

\[
\begin{align*}
\text{if } i < n + 1 & \text{ then call more } \\
\text{elsif } B1? (i) & \text{ then } p.1(\text{ var }) \\
\text{elsif } B2? (i) & \text{ then call more } \text{ fi; } \\
p.3(\text{ var }) ; \\
call \text{ alldone } \text{ end}
\end{align*}
\]

becomes:

more $\equiv$ do do $i := i + 1$;

\[
\begin{align*}
\text{if } i < n + 1 & \text{ then exit } \\
\text{elsif } B1? (i) & \text{ then } p.1(\text{ var }) \\
\text{elsif } B2? (i) & \text{ then exit } \text{ fi; } \\
p.3(\text{ var }) ; \\
call \text{ alldone } \text{ od od end}
\end{align*}
\]
Take Outside Loop

do if \( X = 1 \) then \( Y := 1; \ X := 0 \); \text{exit}(2) 

elsif \( X = 2 \)

then \( Y := 1; \ X := 0 \); \text{exit}(2) 

else \( X := X - Y \) fi od

becomes
Take Outside Loop

\[
\begin{align*}
\text{do if } & X = 1 \text{ then } Y := 1; \ X := 0; \ \text{exit}(2) \\
\text{elseif } & X = 2 \\
\text{ then } & Y := 1; \ X := 0; \ \text{exit}(2) \\
\text{else } & X := X - Y \ \text{fi od}
\end{align*}
\]

becomes

\[
\begin{align*}
\text{do do if } & X = 1 \text{ then } \text{exit}(1) \\
\text{elseif } & X = 2 \\
\text{ then } & \text{exit}(1) \\
\text{else } & X := X - Y \ \text{fi od}; \\
Y := & 1; \ X := 0; \ \text{exit}(1) \ \text{od}
\end{align*}
\]
Double To Single Loop

do do i := i + 1;
    if i < n + 1 then exit
    elsif B1?(i) then p_1( var ); exit(2)
    elsif B2?(i) then exit
        else exit(2) fi od od

becomes:
Double To Single Loop

do do i := i + 1;
    if i < n + 1 then exit
    elsif B1?(i) then p_1( var ); exit(2)
    elsif B2?(i) then exit
    else exit(2) fi od od

becomes:

do i := i + 1;
    if i < n + 1 then skip
    elsif B1?(i) then p_1( var ); exit
    elsif B2?(i) then skip
    else exit fi od
do  \( i := i + 1; \)
    if  \( i < n + 1 \) then skip
    elsif \( B1? (i) \) then \( p_1( \text{ var }); \) exit
    elsif \( B2? (i) \) then skip
    else exit fi od

becomes:
Floot to While

do \ i := i + 1;
    if \ i < n + 1 \ then \ skip
    elsif B1?(i) \ then \ p.1( \ var \ ); \ exit
    elsif B2?(i) \ then \ skip
        else \ exit \ fi \ od

becomes:
fl_flag1 := 0;
while fl_flag1 = 0 do
    i := (i + 1);
    if i < (n + 1) then fl_flag1 := 0
    elsif B1?(i) \ then \ p.1( \ var \ ); \ fl_flag1 := 1
    elsif B2?(i) \ then \ fl_flag1 := 0
        else \ fl_flag1 := 1 \ fi \ od;
Floop to While

Simpler loop:

\begin{verbatim}
do  \ i := (i + 1); 
    if (n + 1) \leq i \land B1?(i) 
      then exit(1)
    elsif (n + 1) \leq i \land \neg B2?(i) 
      then exit(1)
    elsif i < (n + 1) 
      then skip fi od;
\end{verbatim}

becomes:
Floop to While

Simpler loop:

\[
\textbf{do } i := (i + 1); \\
\quad \textbf{if } (n + 1) \leq i \land B1?(i) \\
\quad \quad \textbf{then exit}(1) \\
\quad \textbf{elsif } (n + 1) \leq i \land \neg B2?(i) \\
\quad \quad \textbf{then exit}(1) \\
\quad \textbf{elsif } i < (n + 1) \\
\quad \quad \textbf{then skip} \textbf{ fi} \textbf{ od}; \\
\]

becomes:

\[
i := (i + 1); \\
\textbf{while } \neg B1?(i) \land B2?(i) \lor i < (n + 1) \textbf{ do} \\
\quad i := (i + 1) \textbf{ od;} \\
\]

Note that the statement \(i := i + 1\) had to be copied.
Merge Calls in Action

$$K \equiv \textbf{if } \text{item}[i] \neq \text{last}$$

then \( !P \) write(line var os);

line := "";

m := 0;

inhere(var);

\fbox{\textbf{call more}\textbf{ fi}};

\fbox{\textbf{call more}\textbf{ end}}

Merge the two calls into one:
Merge Calls in Action

\[ K \equiv \text{if } \text{item}[i] \neq \text{last} \]
\[ \quad \text{then } \text{!P write}(\text{line var os}); \]
\[ \quad \text{line} := ""; \]
\[ \quad m := 0; \]
\[ \quad \text{inhere( var );} \]
\[ \quad \text{call more fi;} \]
\[ \quad \text{call more end} \]

Merge the two calls into one:

\[ K \equiv \text{if } \text{item}[i] \neq \text{last} \]
\[ \quad \text{then } \text{!P write}(\text{line var os}); \]
\[ \quad \text{line} := ""; \]
\[ \quad m := 0; \]
\[ \quad \text{inhere( var ) fi;} \]
\[ \quad \text{call more end} \]
In a **regular** action system, any statements immediately following a **call** can be deleted.

Similarly, any statements following an **exit** can be deleted.

\[
x := y; \textbf{call} \ A; \ x := x + 1
\]

becomes:
\[
x := y; \textbf{call} \ A
\]
If a regular action system contains a single action, then there can only be two types of call:

- Calls to the action itself
- Calls to the terminating action (Z)

The action system is replaced by a double loop:

- Calls to the action itself are replaced by `exit`
- Calls to Z are replaced by `exit(2)`

In simple cases, the double loop may be converted to a single loop, or eliminated altogether (if there are no calls to the action itself).
An Intel assembler program to compute a GCD:

.model small
.code
    mov ax,12
    mov bx,8
compare:
    cmp ax,bx
    je theend
    ja greater
    sub bx,ax
    jmp compare
greater:
    sub ax,bx
    jmp compare
theend:
    nop
end
WSL Translation

var \langle \text{flag}_z := 0, \text{flag}_c := 0 \rangle :

actions \text{A_S\_start} :

\text{A_S\_start} \equiv \text{ax} := 12;
    \text{bx} := 8;
    \textbf{call compare end}

\text{compare} \equiv \text{if ax = bx then flag}_z := 1 \text{ else flag}_z := 0 \text{ fi;}
    \text{if ax < bx then flag}_c := 1 \text{ else flag}_c := 0 \text{ fi;}
    \text{if flag}_z = 1 \text{ then call theend fi;}
    \text{if flag}_z = 0 \land \text{flag}_c = 0
    \text{ then call greater fi;}
    \text{if bx = ax then flag}_z := 1 \text{ else flag}_z := 0 \text{ fi;}
    \text{if bx < ax then flag}_c := 1 \text{ else flag}_c := 0 \text{ fi;}
    \text{bx} := \text{bx} - \text{ax};
    \textbf{call compare};
    \textbf{call greater end}

\text{greater} \equiv \text{if ax = bx then flag}_z := 1 \text{ else flag}_z := 0 \text{ fi;}
    \text{if ax < bx then flag}_c := 1 \text{ else flag}_c := 0 \text{ fi;}
    \text{ax} := \text{ax} - \text{bx};
    \textbf{call compare};
    \textbf{call theend end}

\text{theend} \equiv \textbf{call } Z \textbf{ end end actions end}
Flag Removal

actions A_S_start :
A_S_start ≡ ax := 12;
bx := 8;
call compare end

compare ≡ if ax = bx
then if ax < bx
then call theend
else call theend fi
else if ax ≥ bx
then call greater fi fi;
bx := (bx − ax);
call compare;
call greater end

greater ≡ ax := (ax − bx);
call compare;
call compare;
call theend end

theend ≡ call end endactions
ax := 12;
bx := 8;
do if ax = bx
    then if ax < bx
        then exit(1)
        else exit(1) fi
    else if ax ≥ bx
        then ax := (ax − bx)
        else bx := (bx − ax) fi fi od
Simplify

\[ ax := 12; \]
\[ bx := 8; \]
\[ \textbf{while} \ ax \neq bx \ \textbf{do} \]
\[ \quad \textbf{if} \ ax \geq bx \]
\[ \quad \quad \textbf{then} \ ax := ax - bx \]
\[ \quad \quad \textbf{else} \ bx := bx - ax \] 
\[ \quad \textbf{fi} \]
\[ \textbf{od} \]
## Program Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Raw</th>
<th>WSL</th>
<th>Flags</th>
<th>Collapse</th>
<th>Simplify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>36</td>
<td>18</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Expressions</td>
<td>40</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>McCabe</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Control/Data Flow</td>
<td>45</td>
<td>23</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Branch–Loop</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Structural</td>
<td>242</td>
<td>143</td>
<td>58</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Destructuring Transformations

A *restructuring transformation* changes the structure of a program without changing the sequence of state changes which occur during the execution of the program.

Such a transformation preserves the *operational semantics* of the program.

For example:

\[
\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}
\]

is equivalent to:

\[
\text{if } \neg B \text{ then } S_2 \text{ else } S_1 \text{ fi}
\]
Destructuring Transformations

Assignment merging is *not* a restructuring transformation:

\[ x := e_1; x := e_2 \]

is equivalent to:

\[ x := e_2[e_1/x] \]

For example:

\[ x := 2 \times x; x := x + 1 \]

is equivalent to:

\[ x := 2 \times x + 1 \]

The first program has two state changes, but the second has only one, so these are not *operationally* equivalent.
Destructuring Transformations

One method to prove the correctness of a proposed restructuring transformation:

1. Convert the first program to a regular action system with no structured statements
2. Convert the second program to a regular action system with no structured statements
3. Transform the two action systems to a common format

This can sometimes be easier than trying to transform one program directly into the other.
Convert to an Action System

Any program $S$ is equivalent to the regular action system:

```
actions start :
start $\equiv S; \text{call } Z \text{ end endactions}
```

Now, process structured statements in $S$ from the top down, adding new actions to the action system as required.
Suppose we have an action containing a sequence of statements:

\[ A_0 \equiv S_1; S_2; \ldots; S_n; \text{call } B \text{ end} \]

This is equivalent to the set of actions:

\[ A_0 \equiv S_1; \text{call } A_1 \text{ end} \]
\[ A_1 \equiv S_2; \text{call } A_2 \text{ end} \]
\[ A_{n-1} \equiv S_n; \text{call } B \text{ end} \]
Destructuring a Conditional

\[ A_0 \equiv \text{if } B_1 \text{ then } S_1 \]
\[ \quad \text{elsif } B_2 \text{ then } S_2 \]
\[ \quad \text{elsif } \ldots \]
\[ \quad \text{else } S_n \text{ fi;} \]
\[ \quad \text{call } B \text{ end} \]

This is equivalent to:

\[ A_0 \equiv \text{if } B_1 \text{ then call } A_1 \]
\[ \quad \text{elsif } B_2 \text{ then call } A_2 \]
\[ \quad \text{elsif } \ldots \]
\[ \quad \text{else call } A_n \text{ fi end} \]
\[ A_1 \equiv S_1; \text{ call } B \text{ end} \]
\[ \ldots \]
\[ A_n \equiv S_n; \text{ call } B \text{ end} \]
Destructuring a While Loop

\[ A_0 \equiv \text{while } B \text{ do } S \text{ od; call } B \text{ end} \]

This is equivalent to:

\[ A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } B \text{ fi end} \]

\[ A_1 \equiv S; \text{ call } A_0 \text{ end} \]
Destructuring Floops

To destructure an Floop, first absorb the following call into the loop:
\[ A_0 \equiv \text{do } S \text{ od; call } B \text{ end} \]

is transformed to:
\[ A_0 \equiv \text{do } S' \text{ od end} \]

where \( S' \) is \( S \) with each \( \text{exit}(n) \) with terminal value 1 replaced by \( \text{call } B \) (i.e. every \( \text{exit} \) which could terminate the loop).

In other words, any \( \text{exit} \) which can terminate the outermost loop is replaced by \( \text{call } B \).

Then replace the loop with a call to the action:
\[ A_0 \equiv S'; \text{ call } A_0 \text{ end} \]

This is the opposite of Remove_Recursion_In_Action.
Destructuring Floops

An example:

\[ A_0 \equiv \text{do } \text{inhere( var )}; \]

\[
\text{do if } m = 1 \\
\hspace{1em} \text{then } p := \text{number}[i]; \text{ line } := \text{line } + \text{ “, ” } + p \text{ fi;}
\]

\[
\text{last } := \text{item}[i]; \text{ } i := (i + 1);
\]

\[
\text{if } i = (n + 1) \\
\hspace{1em} \text{then } !P \text{ write(line var os); } \text{exit(2)} \text{ fi;}
\]

\[
m := 1; \text{ if item}[i] \neq \text{last} \\
\hspace{1em} \text{then } !P \text{ write(line var os);}
\]

\[
\hspace{2em} \text{line } := \text{“”};
\]

\[
\hspace{3em} m := 0;
\]

\[
\text{exit(1) fi od od;}
\]

\[
\text{call } Z \text{ end}
\]
Destructuring Floops

Absorb the call:

\[ A_0 \equiv \text{do inh}( \text{var} ); \]

\[
\begin{align*}
&\quad \text{do if } m = 1 \\
&\quad \quad \text{then } p := \text{number}[i]; \text{ line := line } + \ "", \ "" + p \ \text{fi;}
&\quad \text{last := item}[i]; \ i := (i + 1);
&\quad \text{if } i = (n + 1) \\
&\quad \quad \text{then } !P \ \text{write(line var os); \ call Z fi;}
&\quad m := 1;
&\quad \text{if item}[i] \neq \text{last} \\
&\quad \quad \text{then } !P \ \text{write(line var os);}
&\quad \quad \quad \text{line := ""};
&\quad \quad \quad m := 0;
&\quad \text{exit(1) fi od od end}
\end{align*}
\]
Destructuring Floops

Remove the loop:

\[ A_0 \equiv \text{inhere( } \text{var } \); \]

\[ \text{do if } m = 1 \]

\[ \text{then } p := \text{number}[i]; \text{ line := line } + \text{ "", } + + p \text{ fi; } \]

\[ \text{last := item}[i]; \quad i := (i + 1); \]

\[ \text{if } i = (n + 1) \]

\[ \text{then } !P \text{ write(line var os); call Z fi; } \]

\[ m := 1; \]

\[ \text{if item}[i] \neq \text{last} \]

\[ \text{then } !P \text{ write(line var os); } \]

\[ \text{line := ""; } \]

\[ m := 0; \]

\[ \text{exit(1) fi od; } \]

\[ \text{call } A_0 \text{ end } \]
Destructuring Floops

Processing the inner loop.

Process the sequence and then absorb the call:

\[
A_0 \equiv \text{inhere( var ); call } A_1 \text{ end}
\]

\[
A_1 \equiv \text{do if } m = 1
\]

\[
\text{then } p := \text{number}[i]; \text{ line := line } + \text{ "", } + p \text{ fi;}
\]

\[
\text{last := item}[i]; \; i := (i + 1);
\]

\[
\text{if } i = (n + 1)
\]

\[
\text{then } !P \text{ write(line var os); call } Z \text{ fi;}
\]

\[
m := 1;
\]

\[
\text{if item}[i] \neq \text{last}
\]

\[
\text{then } !P \text{ write(line var os);}
\]

\[
\text{line := "";}
\]

\[
m := 0;
\]

\[
\text{call } A_0 \text{ fi od end}
\]
Destructuring Floops

Processing the inner loop. Remove the loop:

\[
A_0 \equiv \text{inhere( var ); call } A_1 \text{ end}
\]

\[
A_1 \equiv \text{if } m = 1
\]

\[
\quad \text{then } p := \text{number}[i]; \text{ line := line } + \text{ " , " } + p \text{ fi;}
\]

\[
\quad \text{last := item}[i]; \quad i := (i + 1);
\]

\[
\quad \text{if } i = (n + 1)
\]

\[
\quad \quad \text{then } !P \text{ write(line var os); call } Z \text{ fi;}
\]

\[
\quad m := 1;
\]

\[
\quad \text{if } \text{item}[i] \neq \text{last}
\]

\[
\quad \quad \text{then } !P \text{ write(line var os); }
\]

\[
\quad \quad \quad \text{line := "";}
\]

\[
\quad \quad \quad m := 0;
\]

\[
\quad \quad \text{call } A_0 \text{ fi;}
\]

\[
\text{call } A_1 \text{ end}
\]
Some transformations can be proved correct by converting both programs to action systems and analysing the action systems using Expand_Call (and its inverse), case analysis, renaming etc.

For example, to prove that $P_1$:

\[
\begin{align*}
&\text{do } S_1; \\
&\quad \text{if } B \text{ then exit fi; } \\
&\quad S_2 \text{ od}
\end{align*}
\]

is equivalent to $P_2$:

\[
\begin{align*}
& S_1; \\
& \text{while } \neg B \text{ do } \\
&\quad S_2; \\
&\quad S_1 \text{ od}
\end{align*}
\]

where $S_1$ and $S_2$ are both *proper sequences*. 
Loop Inversion

$P_2$ translates to this action system:

**actions** $A_0$:

$A_0 \equiv S_1; \text{ call } A_1 \text{ end}$

$A_1 \equiv \text{ if } B \text{ then call } Z \text{ else call } A_2 \text{ fi end}$

$A_2 \equiv S_2; \text{ call } A_3 \text{ end}$

$A_3 \equiv S_1; \text{ call } A_1 \text{ end \ endactions}$

$P_1$ translates to this action system:

**actions** $A_0$:

$A_0 \equiv S_1; \text{ call } A_1 \text{ end}$

$A_1 \equiv \text{ if } B \text{ then call } Z \text{ else call } A_2 \text{ fi end}$

$A_2 \equiv S_2; \text{ call } A_0 \text{ end \ endactions}$
Expand the call $A_0$ in $A_2$:

$$A_2 \equiv S_2; \ S_1; \ call \ A_1 \ end$$

Destructure the sequence:

$$A_2 \equiv S_2; \ call \ A_3 \ end$$
$$A_3 \equiv S_1; \ call \ A_1 \ end$$

The two action systems are now identical.
Loop Unrolling

To prove that the program $P_1$:

\[ \text{while } B \text{ do } S \text{ od} \]

is equivalent to $P_2$:

\[ \text{while } B \text{ do } S; \text{ if } B \land Q \text{ then } S \text{ fi od} \]

$P_1$ as an action system:

**actions** $A_0$ :

\[ A_0 \equiv \text{while } B \text{ do } S \text{ od}; \text{ call } Z \text{ end endactions} \]

Destructure the action system:

**actions** $A_0$ :

\[ A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end} \]

\[ A_1 \equiv S; \text{ call } A_0 \text{ end endactions} \]
Loop Unrolling

$P_2$ as an action system:

```plaintext
actions $A_0$:
$A_0 \equiv \text{while } B \text{ do } S; \text{ if } B \land Q \text{ then } S \text{ fi od}; \text{ call } Z \text{ end endactions}
```

Destructure the action system:

```plaintext
actions $A_0$:
$A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$
$A_1 \equiv S; \text{ call } A_2 \text{ end}$
$A_2 \equiv \text{if } B \land Q \text{ then } A_3 \text{ else call } A_0 \text{ fi end}$
$A_3 \equiv S; \text{ call } A_0 \text{ end endactions}$
```
Consider the $P_1$ action system again:

**actions** $A_0$ :

$A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$

$A_1 \equiv S; \text{ call } A_0 \text{ end end actions}$

$P_2$ has an action $A_3 \equiv S; \text{ call } A_0 \text{ end}$ so we add this action to $P_1$ and note that any call to $A_1$ can be replaced by a call to $A_3$.

In particular, $A_0$ is equivalent to:

$A'_0 \equiv \text{if } B \text{ then call } A_3 \text{ else call } Z \text{ fi end}$

Also, $P_2$ has $\text{if } B \land Q \text{ then } \ldots \text{ fi}$ where $P_1$ has $\text{call } A_0$.

So replace $\text{call } A_0$ in $A_1$ by the equivalent statement:

$\text{if } B \land Q \text{ then call } A'_0 \text{ else call } A_0 \text{ fi}$
Loop Unrolling

Unroll call $A'_0$ in $A_1$:

actions $A_0$ :

$A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$

$A_1 \equiv S; \text{ if } B \land Q \text{ then if } B \text{ then call } A_3 \text{ else call } Z \text{ fi}$

else call $A_0$ fi end

$A_3 \equiv S; \text{ call } A_0 \text{ end end actions}$

Simplify:

actions $A_0$ :

$A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$

$A_1 \equiv S; \text{ if } B \land Q \text{ then call } A_3$

else call $A_0$ fi end

$A_3 \equiv S; \text{ call } A_0 \text{ end end actions}$
Loop Unrolling

Destructure:

\textbf{actions} \( A_0 \) :

\( A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end} \)
\( A_1 \equiv S; \text{ call } A_2 \text{ end} \)
\( A_2 \equiv \text{if } B \land Q \text{ then call } A_3 \text{ else call } A_0 \text{ fi end} \)
\( A_3 \equiv S; \text{ call } A_0 \text{ end end actions} \)

This is identical to the destructured version of \( P_2 \).
Entire Loop Unrolling

To prove that $P_1$:
\[ \text{while } B \text{ do } S \text{ od} \]
is equivalent to $P_3$:
\[ \text{while } B \text{ do } S; \text{ while } B \land Q \text{ do } S \text{ od od} \]

Convert $P_3$ to an action system:

**actions** $A_0$ :

$A_0 \equiv \text{if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$

$A_1 \equiv S; \text{ call } A_2 \text{ end}$

$A_2 \equiv \text{if } B \land Q \text{ then call } A_3 \text{ else call } A_0 \text{ fi end}$

$A_3 \equiv S; \text{ call } A_2 \text{ end endactions}$

This is the same as $P_2$ (which we have proved to be equivalent to $P_1$) except that there is a call $A_2$ in the body of $A_3$ instead of call $A_0$. 
Entire Loop Unrolling

actions $A_0$:

$A_0 \equiv \text{ if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end}$

$A_1 \equiv S; \text{ call } A_2 \text{ end}$

$A_2 \equiv \text{ if } B \land Q \text{ then call } A_3 \text{ else call } A_0 \text{ fi end}$

$A_3 \equiv S; \text{ call } A_2 \text{ end endactions}$

Case analysis to prove call $A_2$ is equivalent to call $A_0$ in $A_3$:

1. If $B$ is false or $Q$ is false, then call $A_2$ leads to call $A_0$

2. If $B$ is true and $Q$ is true, then

   (a) call $A_2$ leads, via call $A_3$, to execute $S$ and call $A_2$, while

   (b) call $A_0$ leads, via call $A_1$, to execute $S$ and call $A_2$
Another way to prove that \texttt{call } A_2 \text{ is equivalent to } \texttt{call } A_0 \text{ in } A_3 \text{ is to replace } \texttt{call } A_2 \text{ by the equivalent statement:}

\begin{verbatim}
if B \land Q \text{ then call } A_2
  \text{ else call } A_2 \text{ fi}
\end{verbatim}

Expand each call and simplify:

\textbf{actions} \ A_0 :
\begin{align*}
A_0 & \equiv \text{ if } B \text{ then call } A_1 \text{ else call } Z \text{ fi end} \\
A_1 & \equiv S; \text{ call } A_2 \text{ end} \\
A_2 & \equiv \text{ if } B \land Q \text{ then call } A_3 \text{ else call } A_0 \text{ fi end} \\
A_3 & \equiv S; \\
& \quad \text{ if } B \land Q \text{ then } S; \text{ call } A_1 \\
& \quad \text{ else call } A_0 \text{ fi end end actions}
\end{align*}

Replace \texttt{S; call A_1} by \texttt{call A_0} (since \texttt{B} is true here). We have:
\begin{align*}
A_3 & \equiv S; \text{ call } A_0 \text{ end}
\end{align*}