Appendix: Examples of UNISEX Sessions

This appendix contains three pedagogical examples session to demonstrate the use of the UNISEX system. These transcripts are unedited except for setting user input in boldface.

The first example demonstrates the use of the system for testing a program using numeric values, a mix of numeric and symbolic values, and only symbolic values.

```
% unisex power.p
compiling power.p
loading unisex.....

 1  program power;
 2  var
 3     a,b,raise,temb: integer;
 4  begin
 5     temb := b;
 6     raise := 1;
 7     while temb>0 do
 8        begin
 9           raise := raise*a;
10           temb := temb-1
11        end
12  end.

Execution mode ?
Enter (t)est or (v)erify: t
initialize debug functions
# go
initialize variables
$ a=3, b=5

pascal program execution completed
# vars
Local variables:
a = 3
b = 5
raise = 243
temb = 0

pc: true
# { Note that if tracing is not enabled very little is learned about the
    internals of the program. 3 raised to the 5th power is 243 as expected.
    Let's enable the branch option and try raising a negative number. }

# restart

 1  program power;

Execution mode ?
Enter (t)est or (v)erify: t
initialize debug functions
# branch on
```

go
initialize variables
$ **a=-13, b=3**

predicate on line 7 evaluates to 3 > 0
true path taken

predicate on line 7 evaluates to 2 > 0
true path taken

predicate on line 7 evaluates to 1 > 0
true path taken

predicate on line 7 evaluates to 0 > 0
false path taken

pascal program execution completed
vars
Local variables:
a = -13
b = 3
raise = -2197
temb = 0

pc: true
{ Again the correct result was obtained, but very little additional information was provided. }

restart

```
    1   program power;
```

Execution mode ?
Enter (t)est or (v)erify: t
initialize debug functions
{ To avoid having to restart each time we'll put a break at statement 4. We'll also enable full tracing by using the verbose option. Let's try a symbolic and a numeric value. }

setbr 4
verb on
go
```
    1   program power;
    2   var
    3      a,b,raise,temb: integer;
```
initialize variables
$ **a=VALa, b=3**
b = 3
a = VALa
Local variables:
a = VALa
b = 3
raise = undef3
temb = undef2

```
pc: true

breakpoint at line 4
    4   begin
# { We'll save this for a starting point. }

# save starting point
newly saved state -- 1 --  starting point
# go
    5      temb := b;
temb = 3
    6      raise := 1;
raise = 1
    7      while temb>0 do

predicate on line 7 evaluates to 3 > 0
true path taken
    8        begin
    9          raise := raise*a;
raise = VALa
   10          temb := temb-1
temb = 2
   11        end
    7      while temb>0 do

predicate on line 7 evaluates to 2 > 0
true path taken
    8        begin
    9          raise := raise*a;
raise = VALa*VALa
   10          temb := temb-1
temb = 1
   11        end
    7      while temb>0 do

predicate on line 7 evaluates to 1 > 0
true path taken
    8        begin
    9          raise := raise*a;
raise = VALa*VALa*VALa
   10          temb := temb-1
temb = 0
   11        end
    7      while temb>0 do

predicate on line 7 evaluates to 0 > 0
false path taken
   12  end.

pascal program execution completed
# { The final value of raise contains the desired result. Now let's try
    all symbolic values. First we restore the start state, then we set
    variable values using the change command. }
```

restore 1
state 1 restored -- starting point
next line to be executed -- 4

breakpoint at line 4
    4 begin
# **change a=symbolA**
a = symbolA
# **change b=symbolB**
b = symbolB
# **vars**
Local variables:
a = symbolA
b = symbolB
raise = undef3
temb = undef2

pc: true
go
    5   temb := b;
temb = symbolB
    6   raise := 1;
raise = 1
    7   while temb>0 do

predicate on line 7 evaluates to symbolB > 0

Does Path Condition --
true
---
    imply
symbolB > 0

Enter (t)rue, (f)alse or (n)either
{ We have no way of telling if symbolB > 0, so we would like to try both branches. Thus we save the current state. }

n
Neither is implied.
ASSUME (t)rue or (f)alse
save for false 1st time at loop
newly saved state -- 2 -- for false 1st time at loop
t
New pc: symbolB > 0
---
true path taken
    8   begin
    9   raise := raise*a;
raise = symbolA
   10   temb := temb-1
temb = symbolB - 1
   11   end
    7   while temb>0 do

predicate on line 7 evaluates to symbolB - 1 > 0

Does Path Condition --
symbolB > 0
---
imply
symbolB > 1

Enter (t)rue, (f)alse or (n)either
{ We're at the loop again and again do not know the answer. }

save for false 2nd time at loop
newly saved state -- 3 -- for false 2nd time at loop
n
Neither is implied.
ASSUME (t)rue or (f)alse
t
New pc: symbolB > 1
---
true path taken
8 begin
9 raise := raise*a;
raise = symbolA*symbolA
10 temb := temb-1
temb = symbolB - 2
11 end
7 while temb>0 do

predicate on line 7 evaluates to symbolB - 2 > 0

Does Path Condition --
symbolB > 1
---
imply
symbolB > 2

Enter (t)rue, (f)alse or (n)either
{ Looks like more of the same. Let's save the current state and pursue some other paths to wee what happens. }

save third time at loop
newly saved state -- 4 -- third time at loop
sates
input not understood -- sates
states
currently stored states:

state	line	in	comment
4	7	power	third time at loop
3	7	power	for false 2nd time at loop
2	7	power	for false 1st time at loop
1	4	power	starting point

restore 2
state 2 restored -- for false 1st time at loop
next line to be executed -- 7

```
    7      while temb>0 do

predicate on line 7 evaluates to symbolB > 0
----------------------
Does Path Condition --
true
--------
    imply
symbolB > 0
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# f
New pc: symbolB <= 0
----------------------
false path taken
   12  end.

pascal program execution completed
# vars
Local variables:
a = symbolA
b = symbolB
raise = 1
temb = symbolB

pc: symbolB <= 0
# { By looking at the pc we see that this path is followed if the
   input value for b is less than or equal to zero. Thus, the result
   is 1 whenever the input for b is non-positive. This may or may not
   be what is desired. }

# restore 3
 state 3 restored --  for false 2nd time at loop
next line to be executed -- 7
    7      while temb>0 do

predicate on line 7 evaluates to symbolB - 1 > 0
----------------------
Does Path Condition --
symbolB > 0
--------
    imply
symbolB > 1
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# f
New pc: symbolB = 1
----------------------
```

```
false path taken
   12  end.

pascal program execution completed
# vars
Local variables:
a = symbolA
b = symbolB
raise = symbolA
temb = symbolB - 1

pc: symbolB = 1
# { This looks good. Let's try one more iteration of the loop. }

# restore 4
 state 4 restored --  third time at loop
next line to be executed -- 7
    7      while temb>0 do

predicate on line 7 evaluates to symbolB - 2 > 0
------------------------
Does Path Condition --
symbolB > 1
--------
    imply
symbolB > 2
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# f
New pc: symbolB = 2
------------------------
false path taken
   12  end.

pascal program execution completed
# vars
Local variables:
a = symbolA
b = symbolB
raise = symbolA*symbolA
temb = symbolB - 2

pc: symbolB = 2
# { It appears that the program is working given that we want numbers
   raised to a negative power to yield the result 1. }

# go
return to unix?
Enter (y)es or (n)o: y

%
```

The second example demonstrates the use of the UNISEX system to determine the restrictions that must be placed on symbolic values to cause a particular path to be traversed.

```
% unisex mpy.p
compiling mpy.p
loading unisex.....

 1  program multiply;
 2  var
 3    a,b,prod,tema,temb,s: integer;
 4  begin
 5    tema := a;
 6    prod := 0;
 7    while tema<>0 do
 8      begin
 9        if tema>0 then
10          s := 1
11        else
12          s := -1;
13        temb := b;
14        while temb<>0 do
15          if temb>0 then
16            begin
17              prod := prod+s;
18              temb := temb-1
19            end
20          else
21            begin
22              prod := prod-s;
23              temb := temb+1
24            end;
25        tema := tema-s
26      end
27  end.

Execution mode ?
Enter (t)est or (v)erify: t
initialize debug functions
# verb on
# go
 1  program multiply;
 2  var
 3    a,b,prod,tema,temb,s: integer;
initialize variables
$ a=A,b=B
b = B
a = A
Local variables:
a = A
b = B
prod = undef5
tema = undef4
temb = undef3
```

```
s = undef2

pc: true
    4   begin
    5      tema := a;
tema = A
    6      prod := 0;
prod = 0
    7      while tema<>0 do

predicate on line 7 evaluates to A <> 0
-----------------------
Does Path Condition --
true
--------
    imply
A <> 0
--------
Enter (t)rue, (f)alse or (n)either
# { Respond true for one iteration of outer loop. Must always respond
   with neither first so that assumption is conjoined to pc. }

# n
Neither is implied.
ASSUME (t)rue or (f)alse
# t
New pc: A <> 0
-----------------------
true path taken
    8         begin
    9            if tema>0 then

predicate on line 9 evaluates to A > 0
------------------------
Does Path Condition --
A <> 0
--------
    imply
A > 0
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# f
New pc: A < 0
-----------------------
false path taken
   11            else
   12               s := -1;
s = -1
   13            temb := b;
temb = B
   14            while temb<>0 do
```

```
predicate on line 14 evaluates to B <> 0
-----------------------
Does Path Condition --
A < 0
--------
    imply
B <> 0
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# t
New pc: A < 0  and  B <> 0
-----------------------
true path taken
   15            if temb>0 then

predicate on line 15 evaluates to B > 0
-----------------------
Does Path Condition --
A < 0  and  B <> 0
--------
    imply
B > 0
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
# t
New pc: B > 0  and  A < 0
-----------------------
true path taken
   16              begin
   17                prod := prod+s;
prod = -1
   18                temb := temb-1
temb = B - 1
   19              end
   14          while temb<>0 do

predicate on line 14 evaluates to B - 1 <> 0
-----------------------
Does Path Condition --
B > 0  and  A < 0
--------
    imply
B > 1
--------
Enter (t)rue, (f)alse or (n)either
# n
Neither is implied.
ASSUME (t)rue or (f)alse
```

t
New pc: B > 1 and A < 0
---
true path taken
   15   if temb>0 then

predicate on line 15 evaluates to B - 1 > 0
true path taken
   16   begin
   17   prod := prod+s;
prod = -2
   18   temb := temb-1
temb = B - 2
   19   end
   14   while temb<>0 do

predicate on line 14 evaluates to B - 2 <> 0

Does Path Condition --
B > 1 and A < 0
---
    imply
B > 2

Enter (t)rue, (f)alse or (n)either
{ Respond false to exit after second iteration. }

n
Neither is implied.
ASSUME (t)rue or (f)alse
f
New pc: B = 2 and A < 0
---
false path taken
   25   tema := tema-s
tema = A + 1
   26   end
    7   while tema<>0 do

predicate on line 7 evaluates to A + 1 <> 0

Does Path Condition --
B = 2 and A < 0
---
    imply
A < -1

Enter (t)rue, (f)alse or (n)either
{ Respond false to exit outer loop after one iteration. }

n
Neither is implied.
ASSUME (t)rue or (f)alse
f

```
New pc: A = -1  and  B = 2
----------------------
false path taken
   27  end.

pascal program execution completed
# { Current value of pc gives restrictions on input values needed to force
   this path to be followed. In this case the values are restricted to a
   single possible value for A, -1, and for B, 2. The current path can be
   seen b using the paths command. }

# paths
current path: 1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19,14,15,16,17,18,19,14,25,26,7,27
# restart

    1   program multiply;

Execution mode ?
Enter (t)est or (v)erify: t
initialize debug functions
# { We will now try input values -1 and 2. }

# go
initialize variables
$ a=-1, b=2

pascal program execution completed
# vars
Local variables:
a = -1
b = 2
prod = -2
tema = 0
temb = 0
s = -1

pc: true
# paths
current path: 1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19,14,15,16,17,18,19,14,25,26,7,27
# q
return to unix?
Enter (y)es or (n)o: y

%
```

The final example demonstrates how to generate the necessary verification conditions to show that a program is consistent with its specifications.

```
% unisex divide.p
compiling divide.p
loading unisex.....

 1  program divide(input,output);
 2  var
 3    x,y,quot,rem: integer;
 4  {: entry ( (x>=0),(y>=0) ) :}
 5  {: exit ( (x'=quot*y'+rem),(rem>=0),(rem<y') ) :}
 6  begin
 7    quot := 0;
 8    rem := x;
 9    {: assert ((x=quot*y+rem),(rem>=0),(x=x'),(y=y')) :}
10    while rem>=y do
11      begin
12        quot := quot+1;
13        rem := rem-y
14      end
15  end.

Execution mode ?
Enter (t)est or (v)erify:
v
initialize debug functions
# { This program illustrates the use of an assert-while loop, as well as
    entry and exit specifications. }

# verb on
# go
 1  program divide(input,output);
 2  var
 3    x,y,quot,rem: integer;
initializing variables:
 4  {: entry ( (x>=0),(y>=0) ) :}
 5  {: exit ( (x'=quot*y'+rem),(rem>=0),(rem<y') ) :}
New pc: $x >= 0  and  $y >= 0
 6  begin
 7    quot := 0;
quot = 0
 8    rem := x;
rem = $x
 9    {: assert ((x=quot*y+rem),(rem>=0),(x=x'),(y=y')) :}

predicate on line 9 evaluates to:
  $y = $y  and  $x = $x  and  $x >= 0  and  $x = 0*$y + $x
-----------------------
 ** Prove Condition **
-----------------------
Does Path Condition --
$x >= 0  and  $y >= 0
```

```
--------
    imply
true
--------
     Path verified
----------------------
initializing variables:
New pc: @quot*@y + @rem = @x  and  $x = @x  and  $y = @y  and  @rem >= 0
   10     while rem>=y do

predicate on line 10 evaluates to @rem >= @y
----------------------
Does Path Condition --
@quot*@y + @rem = @x  and  $x = @x  and  $y = @y  and  @rem >= 0
--------
    imply
@rem >= @y
--------
Enter (t)rue, (f)alse or (n)either
# { After the Verification Condition (VC) was taken care of by the simplifier,
   UNISEX reinitialized all of the variables. The current path is not related
   to the one just completed. Here we have an unresolvable decision. Unisex
   prompts the user (as theorem prover) to make a decision. In automatic mode
   a decision of "neither" would result in the false path being stacked
   and the true path being continued. }

# n
Neither is implied.
ASSUME (t)rue or (f)alse
# { Since there are two paths starting at this point we save the current state. }

# save entry to exit (for false branch)
newly saved state -- 1 --  entry to exit (for false branch)
# t
New pc:    @quot*@y + @rem = @x
       and  $x = @x
       and  $y = @y
       and  @rem >= 0
       and  @rem >= @y
----------------------
true path taken
   11       begin
   12          quot := quot+1;
quot = @quot + 1
   13          rem := rem-y
rem = @rem - @y
   14       end
    9     {: assert ((x=quot*y+rem),(rem>=0),(x=x'),(y=y')) :}

predicate on line 9 evaluates to:
  @y = $y  and  @x = $x  and  @rem - @y >= 0  and  @x = (@quot + 1)*@y + @rem - @y
----------------------
  ** Prove Condition **
----------------------
```

Does Path Condition --
@quot*@y + @rem = @x and $x = @x and $y = @y and @rem >= 0 and @rem >= @y
----
    imply
true

     Path verified
----
initializing variables:
New pc: @quot*@y + @rem = @x and $x = @x and $y = @y and @rem >= 0
   10   while rem>=y do

predicate on line 10 evaluates to @rem >= @y

Does Path Condition --
@quot*@y + @rem = @x and $x = @x and $y = @y and @rem >= 0
----
    imply
@rem >= @y

Enter (t)rue, (f)alse or (n)either
paths
current path: 9,10

Paths verified:
9,10,11,12,13,14,9
3,4,5,6,7,8,9

No unverifiable paths.

{ At this point 2 of the 3 paths in the program have been verified. The
decision we are being asked to make now is exactly the same as the last one. }

states
currently stored states:
state line   in   comment
  1   10   divide   entry to exit (for false branch)
restore 1
 state 1 restored -- entry to exit (for false branch)
next line to be executed -- 10
   10   while rem>=y do

predicate on line 10 evaluates to @rem >= @y

Does Path Condition --
@quot*@y + @rem = @x and $x = @x and $y = @y and @rem >= 0
----
    imply
@rem >= @y

Enter (t)rue, (f)alse or (n)either
n
Neither is implied.
ASSUME (t)rue or (f)alse

{ Note that UNISEX does not remember that we told it this decision was unresolvable before we saved the state. We have to tell it again. }

f

```
New pc:    @quot*@y + @rem = @x
       and  $x = @x
       and  $y = @y
       and  @rem >= 0
       and  @rem < @y
-----------------------
false path taken
   15  end.

predicate on line 15 evaluates to:
  @rem < $y  and  @rem >= 0  and  $x = @quot*$y + @rem
-----------------------
  ** Prove Condition **
-----------------------
Does Path Condition --
@quot*@y + @rem = @x  and  $x = @x  and  $y = @y  and  @rem >= 0  and  @rem < @y
--------
    imply
$x - $y*@quot = @rem  and  $y > @rem
--------
Enter (t)rue, (f)alse or (n)either
```

{ This VC shows one limitation of the simplifier; it does not substitute for equal values. We can replace $x with @x and $y with @y and see that the VC is indeed true. }

t

```
     Path verified
-----------------------

pascal program execution completed
```

paths

```
current path: 9,10,15

Paths verified:
9,10,15
9,10,11,12,13,14,9
3,4,5,6,7,8,9

No unverifiable paths.
```

{ The verification of program divide is complete (assuming that the user-theorem prover is sound!). Here are some other commands: }

help

```
      l l l.
      go     quit   restart
      list   list n  list n1,n2
      vars   var varname        change var=value
      states pc     types
```

```
    save name   restore number      rmstate number
    curbr  setbr n1,...   rmbr n1,...
    brk on|off    verb on|off  step on|off
    branch on|off        paths  addpred (expr)
    simplify on|off      axioms       verify routine
    autoverify routine          autoverify program
```

```
# list 9
    9     {: assert ((x=quot*y+rem),(rem>=0),(x=x'),(y=y')) :}
# list 2,8
    2   var
    3     x,y,quot,rem: integer;
    4   {: entry ( (x>=0),(y>=0) ) :}
    5   {: exit ( (x'=quot*y'+rem),(rem>=0),(rem<y') ) :}
    6   begin
    7     quot := 0;
    8     rem := x;
# curbr
No breakpoints set.
# setbr 12
# curbr
Current breakpoints:
12
# q
return to unix?
Enter (y)es or (n)o: y

%
```