AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

SOFTWARE ENGINEERING PRINCIPLES
INTRODUCTION

STEFAN HALLESTEDE
PETER GORM LARSEN
CARL SCHULTZ

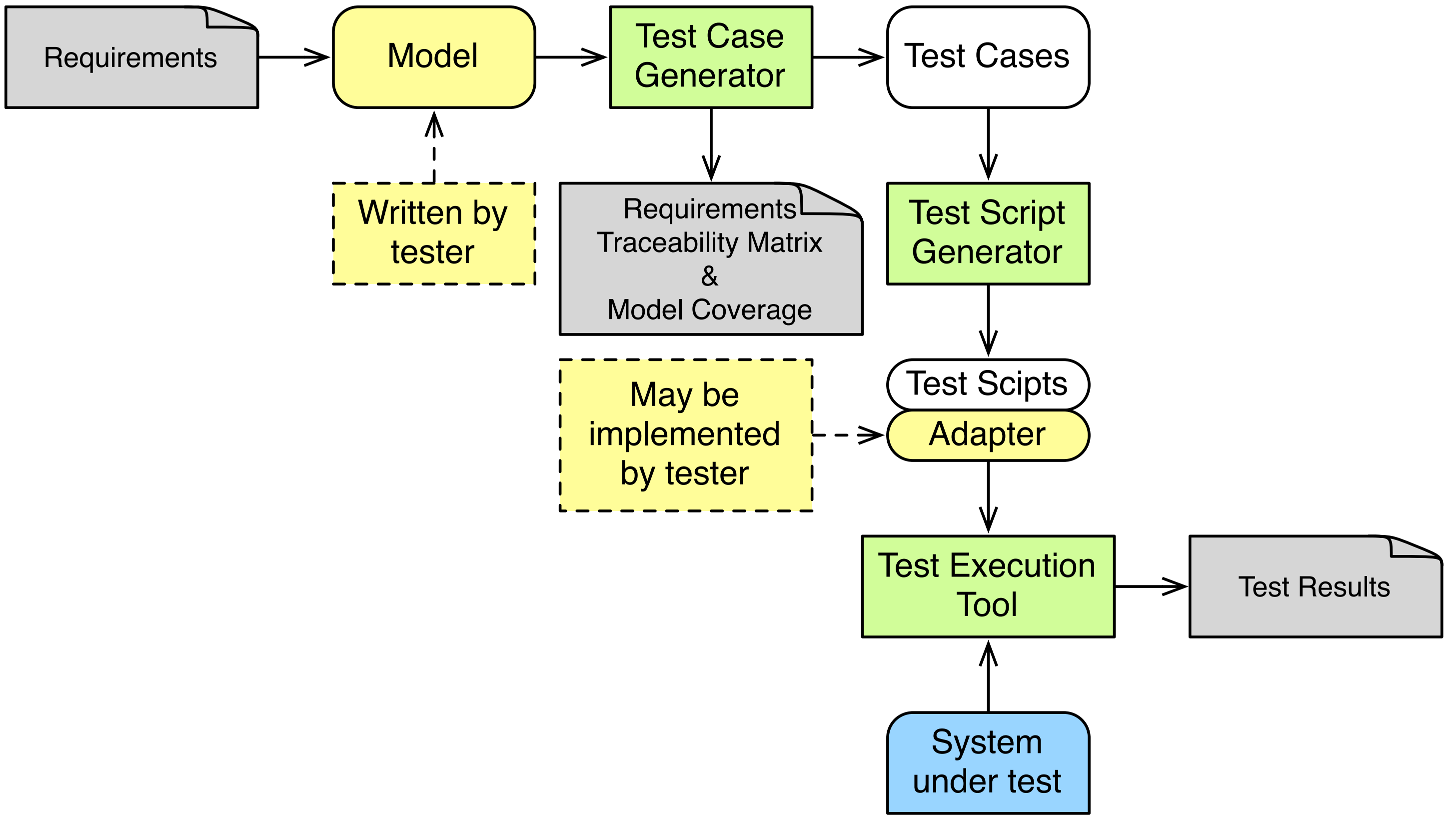
WHY FORMAL METHODS?

- Testing is good for finding defects
- Review is good for judging code quality
- Formal methods are good for guaranteeing correctness

EXPLOIT COMPLEMENTARY STRENGTHS

- Use formal methods with testing
- We have seen test cases derived from specifications before
- So the idea seems not really new
- But now we require formal specifications
- Useful for test automation (Model-based testing)

THE MODEL-BASED TESTING PROCESS



USE OF FORMAL METHODS

- **Requirements analysis**
  - proof of consistency
- **Validation**
  - model animation
- **Verification**
  - proof of correctness of final program
- **Correct by construction**
  - prove correctness during program construction

EXAMPLE: FIND NUMBER IN ARRAY

Assume max : nat is larger than 0

```
state FindAlgo of
  n : nat                    -- limit of array to search
  x : int                    -- value to locate
  a : map nat to int         -- array to search
  r : bool                   -- true if x is found
inv mk_FindAlgo(-,-,a,-) == (dom a = {0,...,max-1})
end
```

SPECIFICATION OF FIND

- **pre** 0 <= n and n <= max
- **post** (r <=> x in set rng (0,...,n-1) <: a) and a = a~

- **pre** ... is called the pre-condition
- **post** ... is called the post-condition

SPECIFICATION OF FIND

- **pre** 0 <= n and n <= max
- **post** (r <=> x in set rng (0,...,n-1) <: a) and a = a~

- In the solutions that we consider a is not modified, so we drop the formula a = a~ to simplify matters for us, leaving us with

- **pre** 0 <= n and n <= max
- **post** r <=> x in set rng (0,...,n-1) <: a

IMPLEMENTING FIND

```
find() == (
   dcl i : nat := 0;
   r := false;
   while true do (
      if a(i) = x then (
         r := true;
         break
      );
      if i >= n-1 then
break;
      i := i+1
   )
)
```

Is this program correct?
(Does it satisfy the specification?)

No,
if n = 0 and a(0) = x,
then r = true.

Of course, VDM does **not** have a break statement!

Let's try again.

IMPLEMENTING FIND AGAIN

find() == (
  dcl i : nat := 0;
  r := false;
  while not r and i<n **do** (
    if a(i) = x **then** r := true;
    i := i+1
)
)

Let's simplify matters a little.

We make i a global variable.

We use that
  if b **then** S
is the same as
  if b **then** S **else** skip

SIMPLIFYING FIND

```
find() == (
   i := 0;
   r := false;
   while not r and i<n do (
      if a(i) = x then
         r := true
      else
         skip;
      i := i+1
   )
)
pre 0 <= n and n <= max
post r <=> x in set rng (0,...,n-1) <: a
```

REASONING ABOUT ”IF”

We write assertions that we expect to be true at certain locations in a program { P }, where P is a predicate.

{ P }
if b **then**
  { P and b }
  S
else
  { P and not b }
  T

REASONING ABOUT ”WHILE”

{ P }
while b **do**
  { P and b }
  S
{ P and not b }

- when the loop starts b is true
- when the loop has terminated b is false
- P is called the invariant of the loop

PROOF RULES

- The intuitive reasoning actually correspond to proof rules that can be used to verify a program

- Given a program annotated with (intermediate) assertions and invariants we can verify that a program satisfies its specification by systematically applying a set of proof rules

- *We verify that the intermediate assertions hold in the corresponding program states*

PROOF RULE FOR ":="

Axiom (BEC):

- { P[x\E] } x := E { P }

Example:

- { x = 0 [x\0] } x := 0 { x = 0 }, that is
- { 0 = 0 } x := 0 { x = 0 }, so we have
- { true } x := 0 { x = 0 }, meaning that x := 0 leads to a state where x = 0 !

PROOF RULE FOR "SKIP"

Axiom (SK):

- { P } skip { P }

PROOF RULE FOR ";"

Rule (SEQ):

- { P } S { Q } and { Q } T { R }
  implies
  { P } S ; T { R }

We often keep Q in the annotated program, writing

- { P } S { Q }; T { R }

- And similarly for the other constructs to come.

PROOF RULE FOR ";" (EXAMPLE)

- { true } x := 0 { x = 0 }

- { x = 1 [x\x+1] } x := x+1 { x = 1 }, hence
- { x + 1 = 1 } x := x+1 { x = 1 }, hence
- { x = 0 } x := x+1 { x = 1 }

- thus:
- { true } x := 0; x := x+1 { x = 1 }

PROOF RULE FOR ”IF”

Rule (IF):

- { P and b } S { Q } and { P and not b } T { Q }
  implies
  { P } **if** b **then** S **else** T { Q }

PROOF RULE FOR "IF" (EXAMPLE)

- { true and x <= y } z := y { x <= z and y <= z }

- { true and not x <= y } z := x { x <= z and y <= z }

- { true }
  if x <= y **then** z := y **else** z := x
  { x <= z and y <= z }

PROOF RULE FOR ”WHILE”

Rule (WH):

- { P and b } S { P }
  implies
  { P } **while** b **do** S { P and not b }

PROOF RULE FOR "WHILE" (EXAMPLE)

- { x <= 10 and x < 10 } x := x+1 { x <= 10 }

- { x <= 10 }
  while x < 10 **do** x := x+1
  { x <= 10 and not x < 10 }

CONSEQUENCE PROOF RULE

Rule (CON):

- P => P' and { P' } S { Q' } and Q' => Q
  implies
  { P } S { Q }

- We also write { P }{ Q } when P => Q when appealing to CON

- We can **weaken** the pre-condition
- And **strengthen** the post-condition

CONSEQUENCE RULE EXAMPLE

- x = 0 => x <= 10
- { x <= 10 }
  while x < 10 **do** x := x+1
  { x <= 10 and not x < 10 }
- x <= 10 and not x < 10 => x = 10

- { x = 0 }
  while x < 10 **do** x := x+1
  { x = 10 }

HOARE PROOF SYSTEM

- { P } S { Q } is called a Hoare triple
- The axioms and rules BEC, SK, SEQ, IF, WH, CON are a complete Hoare proof system for while-programs
- After C. A. R. Hoare (computer scientist)

SIMPLIFIED FIND

```
find() == (
  i := 0;
  r := false;
  while not r and i<n do (
    if a(i) = x then
      r := true
    else
      skip;
    i := i+1
  )
)
pre 0 <= n and n <= max
post r <=> x in set rng (0,...,n-1) <: a
```

PROVING FIND CORRECT

```
{ 0 <= n and n <= max }
i := 0;
r := false;
while not r and i<n do (
   if a(i) = x then
      r := true
   else
      skip;
   i := i+1
)
{ r <=> x in set rng (0,...,n-1) <: a }
```

PROVING FIND CORRECT

```
{ 0  <=  n  and  n  <=  max }
i  :=  0;
{ 0 <= i <=  n  and  i  <=  0 }
r  :=  false;
{ 0 <= i  <=  n and not r }
{ inv: 0  <=  i <= n  and
(r <=> x in set rng (0,...,i-1) <: a) }
while  not  r  and  i<n  do  (
  { inv and not  r  and  i<n  }
    if  a(i)  =  x  then
    { inv and not  r  and  i<n  and
      a(i) =x }
        r  :=  true
    else
     { inv and not  r  and  i<n  and
       a(i) <> x }
        skip;
  { 0 <= i < n and
  r <=> x in set rng (0,...,i) <: a }
    i := i+1
  { inv }
)
{ not (not r and i<n) and
 0  <=  i <= n  and
(r <=> x in set rng (0,...,i-1) <: a) }
{r <=> x in set rng (0,...,n-1) <: a}
```

PROOF (INVARIANT ESTABLISHMENT)

0 <= n and i <= 0 and not r

=>

0 <= i <= n and (r <=> x in set rng (0,...,i-1) <: a)

PROOF (INITIALISATION)

```
{ 0 <= n and n <= max }
i := 0;
{ 0 <= n and i <= 0 }
r := false;
{ 0 <= n and i <= 0 and not r }
```

PROOF (WHILE)

```
not (not r and i<n) and 0  <=  i <= n  and
  (r <=> x in set rng (0,...,i-1) <: a
<=>
(r or i>=n) and 0  <=  i <= n  and
  (r <=> x in set rng (0,...,i-1) <: a
<=>
(r and 0  <=  i <= n  and
  (r <=> x in set rng (0,...,i-1) <: a) or
(i>=n and 0  <=  i <= n  and
  (r <=> x in set rng (0,...,i-1) <: a)
=> r  <=>  x  in  set  rng  (0,...,n-1) <: a
```

PROOF (IF)

(0 <= i < n and r <=> x in set rng (0,...,i-1) <: a)[r\true]

<=>

(0 <= i < n and true <=> x in set rng (0,...,i-1) <: a)

<=

inv and not r and i<n and a(i) =x

PROOF (ELSE)

(0 <= i < n and r <=> x in set rng (0,...,i) <: a)

<=

inv and not r and i<n and a(i) <> x