1

2ITX0 Applied Logic, fall 2021

Topic **Program correctness**

Lecturer: Hans Zantema

`www.win.tue.nl/~hzantema/al.html`

We will present two approaches: **Bounded Model Checking** and **Invariants**

We use **Hoare logic** to express properties of programs and reason about them

We start by Bounded Model Checking, extending the 4th lecture of week 2 of the MOOC

We present a puzzle to explain the idea

2

Start by a single marble

We do steps in which either

- one marble is added, or
- the number of marbles is doubled

What is the **smallest** number of steps required to end up in **exactly 1000** marbles?

3

How to solve this by SAT/SMT?

Fix number k: try for k steps

Introduce $M[i]$ to represent the number of marbles after i steps, for $i = 0, \ldots, k$

Start by one marble:

$$M[0] = 1$$

At the end exactly 1000 marbles:

$$M[k] = 1000$$

4

Requirements for the steps:

$$(M[i] = M[i-1] + 1) \vee (M[i] = 2M[i-1])$$

for $i = 1, \ldots, k$

Resulting formula

$$M[0] = 1 \ \wedge \ M[k] = 1000 \wedge$$
$$\bigwedge_{i=1}^{k} (M[i] = M[i-1] + 1 \vee M[i] = 2M[i-1])$$

is satisfiable if and only there is a solution in k steps

5

For $k = 1, 2, \ldots, 13$ this formula is unsatisfiable

For $k = 14$ it yields the satisfying assignment $M[i] =$
1, 2, 3, 6, 7, 14, 15, 30, 31, 62, 124, 125, 250, 500, 1000
for $i = 0, \ldots, 14$

So 14 is the smallest number of steps

Find out the right number by
binary search: if for k_1 the formula is unsat and for $k_2 > k_1$ it is sat, then continue by $\lfloor \frac{k_1+k_2}{2} \rfloor$

(using `minimize` also possible after some tricks)

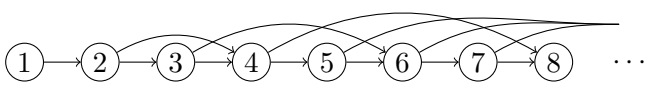
6

More general, we have

- A notion of **state**s, described by the values of variables, in our case only the number of marbles
- A notion of **step**s, describing jumps from one state to another

  The combination of states and steps is called a **transition system**

  A sequence of steps is called a **path**
- A set of **initial** states, in our case consisting of the single state of one marble
- A property to check, in our case: exactly 1000 marbles obtained in k steps



7

Model Checking = automatically prove or disprove a property expressed in some **temporal logic** on paths in a transition system

A typical temporal property is $\mathbf{G}\phi$: ϕ holds **globally** for all paths

If one wants to know whether a state s satisfying a property ϕ is reachable, then one checks $\mathbf{G}\neg\phi$

If it holds, then we know that no state s satisfying ϕ is reachable

If it does not hold, then a state s satisfying ϕ is reachable; the model checker may find a counter example of the formula being a path to s

8

Bounded Model Checking = only consider paths of length bounded by k, and solve this by SAT/SMT introducing v_i for every variable v, for i running from 0 to k, where v_i means the value of v after i steps

Express requirements on initial values on v_0

Express formula to be proven, like requirements on final values on v_k

For every $i = 1, \ldots, k$ describe how v_i is expressed in the values v_{i-1}

For instance, if in a program on variables a, b, c, d the i-th step is $a := b + c$, the corresponding formula is

$$(a_i = b_{i-1} + c_{i-1}) \wedge b_i = b_{i-1} \wedge c_i = c_{i-1} \wedge d_i = d_{i-1}$$

9

More general:

For a program
for $j := 1$ to m do $\cdots$

introduce $m + 1$ copies $a_0, \ldots, a_m$ for every variable a, where a_i means: the value of a after i steps

$a := E$ in step i is expressed as

$$(a_i = E_{i-1}) \wedge \bigwedge_c (c_i = c_{i-1})$$

where c runs over all variables $\neq a$, and E_{i-1} is the copy of expression E in which every variable a is replaced by a_{i-1}

10

Expressing if-statements

if C then S_1 else S_2
in step i is expressed as

$$(C_{i-1} \rightarrow E(S_1)) \wedge (\neg C_{i-1} \rightarrow E(S_2))$$

where

- C_i is the copy of condition C in which every variable a is replaced by a_i
- $E(S_j)$ is the encoding of statement S_j in step i, for $j = 1, 2$

11

Example

for $i := 1$ to 10 do
if $a > b$ then $b := b + i$ else $a := a + b$

is expressed by

$$\bigwedge_{i=1}^{10}((a_{i-1} > b_{i-1} \rightarrow (a_i = a_{i-1} \wedge b_i = b_{i-1} + i))$$

$$\wedge(a_{i-1} \leq b_{i-1} \rightarrow (a_i = a_{i-1} + b_{i-1} \wedge b_i = b_{i-1})))$$

12

How to express requirements for program correctness?
(independent of bounded model checking)

Typically:
Assume some initial condition P holds, called the **precondition**

Then run a program S

Then after running the program has finished, we want a final condition Q to hold, called the **postcondition**

The three ingredients P, S, Q are called a **Hoare triple**, notation

$$\{P\}S\{Q\}$$

13

So in the Hoare triple $\{P\}S\{Q\}$ the conditions P, Q **specify** what the program S is intended to do

Later we will consider **Hoare logic** to reason about Hoare triples $\{P\}S\{Q\}$ for arbitrary programs S

First we will relate Hoare triples to bounded model checking, for programs S doing a fixed number m of steps, and show how the Hoare triple $\{P\}S\{Q\}$ can be shown by SAT/SMT

14

How to express a Hoare triple $\{P\}S\{Q\}$ for S being a program doing a fixed number m of steps?

Take the formula being the conjunction of P_0, $\neg Q_m$ and the encoding of S as sketched before

Run an SMT solver on this formula

If it results in unsat, then the program satisfies the claim specified by $\{P\}S\{Q\}$

That is, we assume that P initially holds (stated by P_0), the intermediate values of the variables statisfy the meaning of S (stated by the encoding of S) and at the end Q does **not** hold (stated by $\neg Q_m$), and from this a contradiction is derived

---

15

Example: after running the program

$a := 0$;
for $i := 1$ to 3 do $a := a + k$

a has the value $3k$

Stated as a Hoare triple: $\{a = 0\}S\{a = 3k\}$, for S being the for-loop

Claim proved by establishing that

$$a_0 = 0 \wedge \neg(a_3 = 3k) \wedge (a_1 = a_0 + k \wedge a_2 = a_1 + k \wedge a_3 = a_2 + k)$$

is unsatisfiable

16

So the specification $\{P\}S\{Q\}$ is proved by proving that

$$P_0 \wedge E(S) \wedge \neg Q_m$$

is **unsatisfiable**, where m is the number of steps of S and $E(S)$ is the encoding of S

In case it is satisfiable, then the corresponding satisfying assignment yields a counter example: a path of values for which initially the precondition P holds, but after the m steps the postcondition Q does not hold

Sometimes this is the goal, for instance in solving the marble puzzle

---

17

When is bounded model checking useful?

If there is one initial state and the program is **deterministic**, then it is not: just run the program and see what happens

But if either

- the number of initial states is very large, or
- the program is **non-deterministic**, that is, makes unknown choices

then running the program in all possible ways is not feasible, and bounded model checking makes sense

---

18

Example of many initial states

Prove correctness of **sorting algorithm**

if $a[1] > a[2]$ then swap$(a[1], a[2])$
if $a[3] > a[4]$ then swap$(a[3], a[4])$
if $a[4] > a[5]$ then swap$(a[4], a[5])$
if $a[2] > a[3]$ then swap$(a[2], a[3])$
$\cdots$

Precondition P: true

Postcondition Q: $a[1] \leq a[2] \leq a[3] \cdots$

Correct if $P_0 \wedge E(S) \wedge \neg Q_m$ is unsatisfiable, and some yet unspecified requirements hold

---

19

For instance, let S be

for $i := 1$ to n do $a[i] := 0$

Then this satisfies
$\{\text{true}\}\ S\ \{a[1] \leq a[2] \leq a[3] \cdots\}$

Clearly S is not a correct sorting algorithm

20

This can be repaired by giving the name A to the initial value of a, require that A is not changed in S, and specify

$\{\forall i : a[i] = A[i]\}$
S
$\{a[1] \leq a[2] \leq a[3] \cdots \ \wedge \ a \text{ is a permutation of } A\}$

If the only way in which a is changed in S is by swap$(a[i], a[j])$, then indeed at every moment a is a permutation of A

So then from

$$E(S) \wedge \neg(\bigwedge_i a[i] \leq a[i+1])_m$$

correctness can be concluded

21

Example with non-determinism: our marbles problem

Program example with non-determinism:

Prove that

$$\{a = 1 \wedge b = 1\}\ S\ \{a \leq 20000\}$$

for S being

for $i := 1$ to 20 do
    if ? then $a := a + b$ else $b := a + b$

for ? being any unknown test

22

Example: foxes and rabbits

Five foxes and five rabbits have to cross a river

There is only one boat that carries at most two animals (and at least one to control the boat)

When the boat is on the river, at any side with at least one rabbit the number of foxes should be $\leq$ the number of rabbits (otherwise the rabbits will be eaten)

How to solve this?

23

Several ways to encode

We prefer to choose an encoding not explicitly distinguishing both sides, and look for an odd number k of crossings

Let f be the number of foxes at the side where the boat is, and let fb be the number of foxes that goes into the boat, and similar r and rb for the rabbits

More precisely:

- f_i is the number of foxes at the other side after crossing i
- fb_i is the number of foxes that is in the boat at crossing i

and similar for rabbits, for $i = 1, \ldots, k$

So $f_i = (5 - f_{i-1}) + fb_i$ for $i = 1, \ldots, k$, and similar for rabbits

---

24

Total formula is conjunction of

$f_0 = 5 \wedge r_0 = 5 \wedge f_k = 5 \wedge r_k = 5$ (initial/final, k odd)

and for each $i = 1, \ldots, k$:

$f_i = (5 - f_{i-1}) + fb_i \ \wedge \ r_i = (5 - r_{i-1}) + rb_i$

$f_i \geq 0 \wedge r_i \geq 0 \wedge fb_i \geq 0 \wedge rb_i \geq 0 \wedge fb_i \leq f_{i-1} \wedge rb_i \leq r_{i-1}$

$0 < fb_i + rb_i \leq 2$

$r_{i-1} = rb_i \ \vee \ (r_{i-1} - rb_i) \geq (f_{i-1} - fb_i)$

Is unsat for k odd, $k \leq 15$, gives solution for $k = 17$

25

Conclusions bounded model checking:

Model checking = prove some property of a program described by initial states end steps

Bounded model checking = encode this in SAT/SMT, over variables a_i meaning the value of a after i steps

Only applies on programs that do a fixed limited number of steps (e.g., for loops)

Several examples, both with many initial states and non-deterministic behavior

By restricting to the first k steps it can approximate results for programs with an unbounded number of steps (e.g., with while loops)

By tool NuSMV with flag `-bmc`, encoding to SAT is done automatically

---

26

Advantage of bounded model checking for limited fixed number of steps: works fully automatically

Typically works well if fixed number of steps ≤ 30, but not for, say, 1000

But what to do if the number of steps is unknown (in a while-loop) or very large?

Then often **invariants** are helpful

Before introducing invariants, first we will present **Hoare logic**: general proof rules to reason about Hoare triples, without indexing all variables by number of step

---

27

Hoare logic

First we give and motivate some general rules, then we will give rules for building blocks of a simple programming language

Note that in contrast to bounded model checking, the variables are **not** labeled any more by the number of the steps

We give the rules in the same notation as the resolution rule:

- a horizontal line,
- on top of it the conditions of the rule
- below the line the conclusion that is drawn

28

Basic rules

Note that if $P \rightarrow Q$ holds, then P is called **stronger** than Q, and Q is called **weaker** then P

Strengthening precondition

$$\frac{\begin{array}{c} M \rightarrow P \\ \{P\}S\{Q\} \end{array}}{\{M\}S\{Q\}}$$

Weakening postcondition

$$\frac{\begin{array}{c} Q \rightarrow R \\ \{P\}S\{Q\} \end{array}}{\{P\}S\{R\}}$$

29

Basic rules

Disjunction precondition

$$\frac{\begin{array}{c} \{M\}S\{Q\} \\ \{P\}S\{Q\} \end{array}}{\{M \vee P\}S\{Q\}}$$

Conjunction postcondition

$$\frac{\begin{array}{c} \{P\}S\{Q\} \\ \{P\}S\{R\} \end{array}}{\{P\}S\{Q \wedge R\}}$$

30

For a given program S, there may be several P, Q such that $\{P\}S\{Q\}$ holds

What are the most interesting / powerful?

For a given postcondition Q, typically describing the goal of the program, a corresponding precondition P may always be strengthened, while for the strongest property false the Hoare triple $\{\text{false}\}S\{Q\}$ does not give any information

So the weaker the precondition P is, the more information is given by the Hoare triple $\{P\}S\{Q\}$

In many cases the weakest possible precondition for S and Q exists, denoted by $wp(S, Q)$, and satisfies

$$\{P\}S\{Q\} \;\equiv\; P \rightarrow wp(S, Q)$$

31

Due to

$$\{P\}S\{Q\} \;\equiv\; P \rightarrow wp(S, Q)$$

for concluding a Hoare triple we have

WP rule

$$\frac{P \rightarrow wp(S, Q)}{\{P\}S\{Q\}}$$

We will see that often first computing $wp(S, Q)$ and then apply this WP rule is more convenient then using the proof rules for the building blocks of S

32

By computing **weakest precondition** and not **strongest postcondition** we start by reasoning from the end situation described by the postcondition, and do a **backward** reasoning from there

It looks more intuitive to start from the beginning and then reason forward, but doing the backward reasoning guided by the weakest precondition computation has a number of advantages:

- in a specification by a Hoare triple $\{P\}S\{Q\}$ the key information describing the goal of the program is typically given in the postcondition Q
- the rules for computing weakest precondition are simple and straightforward

---

33

For

- assignment :=
- composition ;, and
- if-then-else

we will give both rules for computing $wp(S, Q)$ and proof rules for Hoare triples

We will use $\langle \cdots \rangle$ to group parts of programs together

---

34

Sometimes the use of $\langle \cdots \rangle$ does not influence the meaning of the program, and may be omitted

For instance, both
$\langle S1; S2\rangle; S3$ and $S1; \langle S2; S3\rangle$

mean: first do $S1$, then $S2$, then $S3$, so it is just written as $S1; S2; S3$

But
$\langle$ if C then $S1\rangle; S2$ and if C then $\langle S1; S2\rangle$

describe distinct programs, so here the use of $\langle \cdots \rangle$ is crucial

---

35

36

The assignment :=

Notation:
For a variable x, an expression E and a condition P we write

$$P^x_E$$

for a copy of P in which every occurrence of x is replaced by E

Assignment rule

$$\overline{\{Q^x_E\}x := E\{Q\}}$$

37

Note that

$$\overline{\{Q^x_E\}x := E\{Q\}}$$

is a proof rule without a condition

Q^x_E is not just a precondition; it is the weakest precondition:

WP for assignment

$$wp(x := E, Q) \equiv Q^x_E$$

38

Example:

What is the weakest precondition P such that

$$\{P\}\ x := 83\ \{x + y = 100\}$$

Indeed, P should state $83 + y = 100$

$$83 + y = 100 \equiv (x + y = 100)^x_{83}$$

so

$$wp(x := 83, x + y = 100) \equiv (x + y = 100)^x_{83}$$

39

Also correct if E contains the variable x

Example:

What is the weakest precondition P such that

$$\{P\}\ x := x + 3\ \{x = 100\}$$

Indeed, P should state $x + 3 = 100$

$$x + 3 = 100 \equiv (x = 100)^x_{x+3}$$

so

$$wp(x := x + 3, x = 100) \equiv (x = 100)^x_{x+3}$$

40

Composition ;

We have $\{P\}S;T\{Q\}$ if and only if a condition M exists such that

$$\{P\}S\{M\} \quad \text{and} \quad \{M\}T\{Q\}$$

So:
Composition rule

$$\frac{\{P\}S\{M\} \quad \{M\}T\{Q\}}{\{P\}S;T\{Q\}}$$

41

Choose $M \equiv wp(T, Q)$, then we obtain
WP of composition

$$wp(S;T,Q) \;\equiv\; wp(S, wp(T,Q))$$

Indeed,

$$\begin{array}{rcl} wp(x := x+1; x := x+2, x = 100) & \equiv & wp(x := x+1, wp(x := x+2, x = 100) \\ & \equiv & wp(x := x+1, x+2 = 100) \\ & \equiv & (x+1)+2 = 100 \\ & \equiv & x = 97 \end{array}$$

42

We consider two programs to swap two variables a, b, that is, for A, B not in the program S it should hold

$$\{a = A \wedge b = B\}S\{a = B \wedge b = A\}$$

The first one uses an extra variable c:

$$c := a;\; a := b;\; b := c$$

Indeed,

$$\begin{array}{rcl} & & wp(c := a;\; a := b;\; b := c,\; a = B \wedge b = A) \\ & \equiv & wp(c := a;\; a := b,\; wp(b := c,\; a = B \wedge b = A)) \\ & \equiv & wp(c := a;\; a := b,\; a = B \wedge c = A) \\ & \equiv & wp(c := a,\; wp(a := b,\; a = B \wedge c = A)) \\ & \equiv & wp(c := a,\; b = B \wedge c = A) \\ & \equiv & b = B \wedge a = A \end{array}$$

43

Same example

$$\{a = A \wedge b = B\}\; c := a;\; a := b;\; b := c\; \{a = B \wedge b = A\}$$

argued by forward reasoning starting from the beginning:

$$\begin{array}{rcl} \{a = A \wedge b = B\} & c := a; & \{a = A \wedge b = B \wedge c = A\} \\ \{a = A \wedge b = B \wedge c = A\} & a := b; & \{a = B \wedge b = B \wedge c = A\} \\ \{a = B \wedge b = B \wedge c = A\} & b := c & \{a = B \wedge b = A \wedge c = A\} \end{array}$$

so indeed $\{a = A \wedge b = B\}\ c := a;\ a := b;\ b := c\ \{a = B \wedge b = A\}$ by weakening postcondition

For this example still feasible, but needs reasoning over double set of variables a, b, c, A, B, C instead of only a, b, c, and becomes more involved for more complicated examples

So we will focus on using backward reasoning using the wp rules

---

44

The second program also satisfies

$$\{a = A \wedge b = B\} S \{a = B \wedge b = A\}$$

but instead of using an extra variable c it exploits $+, -$:

$$a := a + b;\ b := a - b;\ a := a - b$$

We obtain

$$\begin{array}{ll} & wp(a := a + b;\ b := a - b;\ a := a - b,\ a = B \wedge b = A) \\ \equiv & wp(a := a + b;\ b := a - b,\ wp(a := a - b,\ a = B \wedge b = A)) \\ \equiv & wp(a := a + b;\ b := a - b,\ a - b = B \wedge b = A) \\ \equiv & wp(a := a + b,\ wp(b := a - b, a - b = B \wedge b = A)) \\ \equiv & wp(a := a + b,\ a - (a - b) = B \wedge a - b = A) \\ \equiv & wp(a := a + b,\ b = B \wedge a - b = A) \\ \equiv & b = B \wedge (a + b) - b = A \\ \equiv & b = B \wedge a = A \end{array}$$

45

If-then-else

If-then-else rule

$$\frac{\{P \wedge C\} S \{Q\} \quad \{P \wedge \neg C\} T \{Q\}}{\{P\} \text{ if } C \text{ then } S \text{ else } T \{Q\}}$$

WP of if-then-else

$$wp(\text{ if } C \text{ then } S \text{ else } T, Q) \ \equiv\ (C \rightarrow wp(S, Q)) \wedge (\neg C \rightarrow wp(T, Q))$$

46

Variant if there is no else-branch:

If-then rule

$$\frac{\{P \wedge C\}S\{Q\} \quad (P \wedge \neg C) \to Q}{\{P\} \text{ if } C \text{ then } S\{Q\}}$$

WP of if-then

$$wp(\text{ if } C \text{ then } S, Q) \;\equiv\; (C \to wp(S, Q)) \wedge (\neg C \to Q)$$

47

Example:

We want to prove that after executing

$$\text{if } x > y \text{ then } z := x \text{ else } z := y$$

the value of z is the maximum of x and y, that is: $z \geq x \wedge z \geq y \wedge (x = z \vee y = z)$

$wp(\text{ if } x > y \text{ then } z := x \text{ else } z := y, z \geq x \wedge z \geq y \wedge (x = z \vee y = z))$

$\equiv\; (x > y \to wp(z := x, z \geq x \wedge z \geq y \wedge (x = z \vee y = z))) \wedge$
$\qquad (\neg(x > y) \to wp(z := y, z \geq x \wedge z \geq y \wedge (x = z \vee y = z)))$

$\equiv\; (x > y \to (x \geq x \wedge x \geq y \wedge (x = x \vee y = x))) \wedge$
$\qquad (\neg(x > y) \to (y \geq x \wedge y \geq y \wedge (x = y \vee y = y)))$

$\equiv$ true

48

If expressions like the last claim $P \equiv$ true become more complicated, you may prove that $\neg P$ is unsatisfiable by an SMT solver

If we see programs like

$$\text{for } i := 1 \text{ to } 10 \text{ do } S$$

as an abbreviation of

$$i := 1; S; i := 2; S; i := 3; S; \cdots; i := 10; S$$

then for all program doing a fixed number of steps we developed rules to compute its weakest precondition

---

49

By

$$\{P\}S\{Q\} \;\equiv\; P \rightarrow wp(S,Q)$$

we may prove or disprove Hoare triples for all such programs, in a way where we do not need to label the variables by the number of the step as we did in bounded model checking

In contrast to bounded model checking, we can also develop rules for while-programs doing an unbounded number of steps

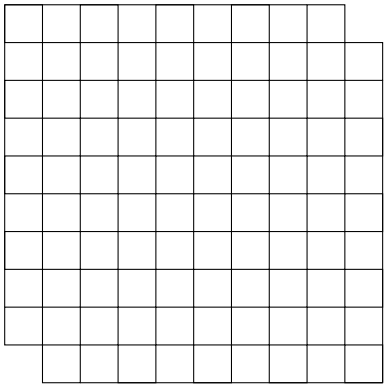
Exactly computing the weakest precondition of a while-program will fail, but we can deal with Hoare triples

The key notion for doing so is **invariant**, that we now introduce by some puzzles

---

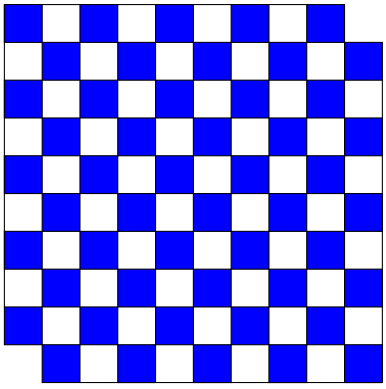
50

Can you fill a 10×10 square with two opposite corner cells removed



by only copies of the domino  ?

51



NO!

nr of blue cells $= 50 \;\neq\; 48 =$ nr of white cells

Every domino covers one blue and one white cell, so in every covering by dominoes the number of blue cells covered = number of white cells covered

---

52

The key idea in the argument is finding the **invariant**, that is, a property

- that holds initially
- if it holds, then after doing a step then it holds again
- so it holds at the end

In the tiling puzzle the invariant is
the number of covered blue cells = the number of covered white cells

---

53

Peg solitaire, see wikipedia

54

The MU puzzle (Hofstadter: Gödel, Escher, Bach)

Start with string MI

Allow following steps:

- Behind a string ending in I you may put symbol U
- String Mx may be replaced by Mxx
- Every pattern III may be replaced by U
- Every pattern UU may be removed

MI $\to$ MII $\to$ MIIII $\to$ MIIIIU $\to$ MUIU $\to$ MUIUUIU $\to$ MUIIU $\to$ $\cdots$

Can you make MU?

55

NO

What happens with numbers of I and U?

step type	I	U
1	$+0$	$+1$
2	$\times 2$	$\times 2$
3	-3	$+1$
4	$+0$	-2

We see the invariant

Number of I is **NOT** a multiple of 3

So MU can **NOT** be obtained, since it contains 0 I's, and 0 is a multiple of 3

56

For program correctness invariants are helpful, even if the number of steps in the program is unknown or very large

Example (informal): after running the program

$a := 0$;
for $i := 1$ to 1000 do $a := a + k$

a has the value $1000k$

Stated as a Hoare triple: $\{a = 0\}S\{a = 1000k\}$, for S being the for-loop

Invariant: after i steps the value of a is ik

Indeed, initially it holds, and if $a = ik$ then after $a := a + k$ we obtain $a = (i + 1)k$, so after 1000 steps we have $a = 1000k$

57

Partial correctness

By introducing while loops we also introduce programs that do not **terminate**, that is, they run forever

Now $\{P\}S\{Q\}$ can be interpreted in two ways:

- **Total correctness**: if P holds, then by running S the program S will terminate, and after termination Q holds
- **Partial correctness**: if P holds, then the program S will either run forever, or after termination Q holds

We will consider $\{P\}S\{Q\}$ as **partial correctness**

Then for proving total correctness the extra requirement that it terminates has to be proven separately

---

58

Invariants for while loops

For this notion of partial correctness we can prove

$$\{P\} \text{ while } C \text{ do } S\{Q\}$$

by finding an invariant I such that

- $P \to I$ (**Initial condition**: I holds initially)
- $\{I \wedge C\}S\{I\}$ (**Invariance**: if I holds, then after loop step again I holds)
- $(I \wedge \neg C) \to Q$ (**End condition**: at the end Q holds)

In fact, invariance corresponds to the induction step when proving by induction on n that the invariant holds after n steps

---

59

Stated as a proof rule:

While rule

$$\frac{\begin{array}{c} P \to I \\ \{I \wedge C\}S\{I\} \\ (I \wedge \neg C) \to Q \end{array}}{\{P\} \text{ while } C \text{ do } S\{Q\}}$$

In proving partial correctness (specified by a Hoare triple) of a while loop, you **always** have to

- choose a suitable invariant I, and
- check exactly these three conditions for this invariant I

60

If S is composed from assignments, composition and/or if-then-else then

$$\{I \wedge C\}S\{I\}$$

can be proved by computing $wp(S, I)$, and checking that

$$(I \wedge C) \rightarrow wp(S, I)$$

holds

61

Example
$\{a > 0 \wedge a = A \wedge b = B\}$
while $a \neq 0$ do $\langle a := a - 1; b := b + 1\rangle$
$\{b = A + B\}$

Choose $I \equiv a + b = A + B$

Now we check the three conditions:

- $P \rightarrow I$: $(a > 0 \wedge a = A \wedge b = B) \rightarrow (a + b = A + B)$ holds
- $\{I \wedge C\}S\{I\}$: first compute $wp(S, I)$

62

$$\begin{aligned} wp(S, I) &\equiv wp(a := a - 1; b := b + 1, a + b = A + B) \\ &\equiv wp(a := a - 1, wp(b := b + 1, a + b = A + B)) \\ &\equiv wp(a := a - 1, a + b + 1 = A + B) \\ &\equiv a - 1 + b + 1 = A + B \\ &\equiv a + b = A + B \\ &\equiv I \end{aligned}$$

Now $(I \wedge C) \rightarrow wp(S, I)$ holds, so indeed $\{I \wedge C\}S\{I\}$

- $(I \wedge \neg C) \rightarrow Q$: $(a + b = A + B \wedge \neg(a \neq 0)) \rightarrow b = A + B$ holds

So indeed the three conditions of the while rule hold, and we have proved partial correctness stated by the given Hoare triple

---

63

Typically, for a given program finding a suitable invariant I satisfying these properties may be hard and is not done automatically

Checking the three requirements $P \rightarrow I$, $\{I \wedge C\}S\{I\}$ and $(I \wedge \neg C) \rightarrow Q$ may be done either by hand or by support of SMT

$\{I \wedge C\}S\{I\}$ can be checked either by wp calculus, or by bounded model checking

By bounded model checking: prove that

- $\neg(P \to I)$
- $(I \wedge C)_0 \wedge E(S) \wedge \neg I_m$ (S does m steps and is encoded by $E(S)$)
- $\neg((I \wedge \neg C) \to Q)$

are all three unsatisfiable

Note that in first and last requirement the variables are not numbered

---

64

Often a while statement is preceded by an initialization S_1

Recall the While rule

$$\frac{\begin{array}{c} P \to I \\ \{I \wedge C\} S \{I\} \\ (I \wedge \neg C) \to Q \end{array}}{\{P\} \text{ while } C \text{ do } S \{Q\}}$$

Since $I \to I$ trivially holds, from assuming $\{I \wedge C\} S \{I\}$ and $(I \wedge \neg C) \to Q$ we may conclude

$$\{I\} \text{ while } C \text{ do } S \{Q\}$$

If moreover $\{P\}\ S_1\ \{I\}$ holds, then by the composition rule we obtain

$$\{P\}\ S_1;\ \text{while } C \text{ do } S\ \{Q\}$$

65

Summarizing, we have

While with initialization rule

$$\frac{\begin{array}{c} \{P\}\ S_1\ \{I\} \\ \{I \wedge C\}\ S\ \{I\} \\ (I \wedge \neg C) \to Q \end{array}}{\{P\}\ S_1;\ \text{while } C \text{ do } S\ \{Q\}}$$

So by replacing the initial condition by $\{P\}\ S_1\ \{I\}$, we may conclude

$$\{P\}\ S_1;\ \text{while } C \text{ do } S\ \{Q\}$$

a pattern that is very common

66

Often the program is not yet given, but only the specification consisting of precondition P and postcondition Q

Then a good policy is **first** to choose the invariant I, and then design the program in such a way that the requirements hold

We illustrate this by a more complicated program to compute $n * k$, being much more efficient than n times adding the value k as we saw before

---

67

As the invariant I we choose $nk = ab + c$

This can be initialized by $a := n; b := k; c := 0$, indeed by $nk = nk + 0$ we obtain

$$\{\text{true}\}\ a := n; b := k; c := 0\ \{I\}$$

If at the end we have $a = 0$, then from $I \equiv (nk = ab + c)$ we can conclude $c = nk$, so then the desired value nk has been computed and is stored in the variable c

So choose $Q \equiv c = nk$, and $C \equiv (a \neq 0)$, then $(I \wedge \neg C) \rightarrow Q$

It remains to choose the body S of the while loop such that $\{I \wedge C\}S\{I\}$ holds

---

68

Policy: at the end we want $a = 0$, so we want to **decrease** a

Correct body that always works: $a := a - 1; c := c + b$

Indeed, for the invariant $I \equiv (nk = ab + c)$ we have

$$(nk = ab + c) \rightarrow (nk = (a - 1)b + (c + b))$$

intuitively stating that after the step a and c are replaced by their new values $a - 1$ and $b + c$, for which the invariant holds again

On the next slide we elaborate this in detail, exploiting our wp rules

---

69

$$\begin{array}{rcl} wp(S, I) & \equiv & wp(a := a - 1; c := c + b, nk = ab + c) \\ & \equiv & wp(a := a - 1, wp(c := c + b, nk = ab + c)) \\ & \equiv & wp(a := a - 1, nk = ab + (c + b)) \\ & \equiv & (a - 1)b + (c + b) \\ & \equiv & ab + c \equiv I \end{array}$$

so indeed $\{I \wedge C\}S\{I\}$ since $(I \wedge C) \rightarrow wp(S, I)$ holds

So
$a := n; b := k; c := 0;$
while $a \neq 0$ do $\langle a := a - 1; c := c + b\rangle$

is a correct program T to compute $c = nk$, that is, satisfies $\{\text{true}\}T\{c = nk\}$

However, it is not yet efficient and still does n steps

70

Key observation

if a even then also $a := \frac{a}{2}; b := 2b$ maintains the invariant $I \equiv (nk = ab + c)$, since

$$(nk = ab + c) \rightarrow (nk = \frac{a}{2}(2b) + c)$$

So the following program also satisfies the requirements:

$a := n; b := k; c := 0;$
while $a \neq 0$ do
    if a even then $\langle a := \frac{a}{2}; b := 2b \rangle$
    else $\langle a := a - 1; c := c + b \rangle$

71

More precisely:

we have to prove

$$(I \wedge a \neq 0) \rightarrow wp(S, I)$$

for S being
    if a even then $\langle a := \frac{a}{2}; b := 2b \rangle$
    else $\langle a := a - 1; c := c + b \rangle$

Using the wp rule for if-then-else we have to prove that $I \wedge a \neq 0$ implies both

a even $\rightarrow wp(a := \frac{a}{2}; b := 2b, I)$

and
$\neg(a \text{ even}) \rightarrow wp(a := a - 1; c := c + b, I)$

72

The second claim we already proved, the first follows from

$$\begin{array}{rcl} wp(a := \frac{a}{2}; b := 2b, I) & \equiv & wp(a := \frac{a}{2}, wp(b := 2b, nk = ab + c)) \\ & \equiv & wp(a := \frac{a}{2}, nk = a(2b) + c) \\ & \equiv & nk = \frac{a}{2}(2b) + c \\ & \equiv & nk = ab + c \equiv I \end{array}$$

Termination can also be proved, but is omitted here

73

This program is very efficient: one can show that it only takes $O(\log n)$ steps

In a similar way by choosing the invariant

$$I \equiv \ k^n = b^a * c$$

one derives the program

$a := n; b := k; c := 1;$
while $a \neq 0$ do
    if a even then $\langle a := \frac{a}{2}; b := b^2 \rangle$
    else $\langle a := a - 1; c := c * b \rangle$

that efficiently computes $c = k^n$

74

More precisely, again the number of steps is logarithmic in n, so this is feasible for very large n

This not only works if k is a number, but also if k is a square matrix, or a number modulo some number m and all computations are done modulo m

By the latter k^n mod m can be computed in a fraction of a second if $k, n, m \approx 10^{1000}$

This is extensively used in practice for encoding and decoding in cryptography, in particular in RSA

More precisely: k is the message, only known by the sender, and n_1, n_2 satisfy $k^{n_1 n_2} \equiv 1$ mod m

The sender encrypts by sending k^{n_1} mod m; the receiver decrypts it by taking it to the power n_2 mod m, where n_2 is not known by others

---

75

How to choose the invariant?

For a given precondition P and postcondition Q we try to find an invariant I, and a program S_1; while C do S for which we can prove the three conditions

- $\{P\}\ S_1\ \{I\}$
- $\{I \wedge C\}\ S\ \{I\}$
- $(I \wedge \neg C) \rightarrow Q$

in order to conclude correctness of the Hoare triple

$$\{P\}\ S_1;\ \text{while } C \text{ do } S\ \{Q\}$$

How to choose I and the rest?

76

Main approach: **weaken the postcondition**, that is, choose I being similar to Q such that

- $Q \rightarrow I$ holds, and

- a simple expression C can be found such that $(I \wedge \neg C) \leftrightarrow Q$ holds, or at least $(I \wedge \neg C) \rightarrow Q$, and
- I is easy to initialize

So if Q can be written as (or follows from) $Q_1 \wedge Q_2$, then choose $I \equiv Q_1$ and $C \equiv \neg Q_2$

A fruitful trick may be to replace a constant k in Q by a variable b, and choose $I \equiv Q[k := b]$ and $C \equiv b \neq k$

77

Example

A simple program to compute $Q \equiv a = n * k$ for given constants k, n, $k \geq 0$

Choose $I \equiv a = n * b$ and $C \equiv b \neq k$

I is easy to initialize by $a := 0; b := 0$

Choose S to be $a := a + n; b := b + 1$, then we obtain $\{I \wedge C\}S\{I\}$, since $a = n * b \equiv a + n = n * (b + 1)$

This proves

$$\{k \geq 0\}\ a := 0; b := 0;\ \text{while } b \neq k \text{ do } \langle a := a + n; b := b + 1\rangle\ \{a = n * k\}$$

78

Binary search

Given a sorted array, that is, $a_1 \leq a_2 \leq \cdots \leq a_n$, and a value k, find p such that $a_p \leq k < a_{p+1}$

We assume $a_1 \leq k < a_n$, otherwise add $-\infty$ at begin and $+\infty$ at end of array

So we are looking for a program T, not changing the values of the array a such that

$\{a_1 \leq k < a_n\}\ T\ \{a_p \leq k < a_{p+1}\}$

79

$\{a_1 \leq k < a_n\}\ T\ \{a_p \leq k < a_{p+1}\}$

We choose the invariant $I:\ a_p \leq k < a_q$, this is easily initialized by $p := 1; q := n$

Goal of the program: decrease $q - p$ until it is 1, maintaining the invariant

For efficiency we want p, q to make big steps rather than only changing by 1

Key idea: consider the new value $\lfloor \frac{p+q}{2} \rfloor$ which is in the middle of p and q as good as possible

---

80

For $r = \lfloor \frac{p+q}{2} \rfloor$ we obtain:

if $a_p \leq k < a_q$ and $k < a_r$ then $a_p \leq k < a_r$

if $a_p \leq k < a_q$ and $\neg(k < a_r)$ then $a_r \leq k < a_q$

For our invariant $a_p \leq k < a_q$ this yields invariance in the following program T:

$p := 1; q := n;$
while $q \neq p + 1$ do
$\langle r := \lfloor \frac{p+q}{2} \rfloor;$
if $k < a_r$ then $q := r$ else $p := r\rangle$

for which we now will prove $\{a_1 \leq k < a_n\}\ T\ \{a_p \leq k < a_{p+1}\}$ in detail

---

81

We choose the invariant $I:\ a_p \leq k < a_q$

Then we have

$$\{a_1 \leq k < a_n\}\ p := 1; q := n\ \{I\}$$

since $wp(p := 1; q := n, I) \equiv a_1 \leq k < a_n$

Next we have to prove $\{I \wedge q \neq p + 1\}\ S\ \{I\}$ for S:
$r := \lfloor \frac{p+q}{2} \rfloor;$
if $k < a_r$ then $q := r$ else $p := r$

$wp(r := \lfloor \frac{p+q}{2} \rfloor;\ \text{if } k < a_r \text{ then } q := r \text{ else } p := r, I)\ \equiv$
$wp(r := \lfloor \frac{p+q}{2} \rfloor, (k < a_r \rightarrow a_p \leq k < a_r) \wedge (\neg(k < a_r) \rightarrow a_r \leq k < a_q))\ \equiv$
$(k < a_r \rightarrow a_p \leq k < a_r) \wedge (\neg(k < a_r) \rightarrow a_r \leq k < a_q)$ for $r = \lfloor \frac{p+q}{2} \rfloor$
which indeed can be concluded from $I \wedge q \neq p + 1$

82

Finally we observe $I \wedge \neg(q \neq p + 1)\ \rightarrow\ a_p \leq k < a_{p+1}$

So we have proved

$$\{a_1 \leq k < a_n\}\ T\ \{a_p \leq k < a_{p+1}\}$$

for our program T:
$p := 1; q := n;$
while $q \neq p + 1$ do
$\langle r := \lfloor \frac{p+q}{2} \rfloor;$
if $k < a_r$ then $q := r$ else $p := r\rangle$
by using the proof rule for while with initialization

This shows partial correctness; the termination argument is not given here

Surprisingly, we did not use that the array is sorted, so the claim is also correct for unsorted array (but then it cannot be used to check whether k occurs in the array)

83

Graph algorithms

A (directed) graph (V, E) consists of a set V of **vertices** or **nodes**, and a set $E \subseteq V \times V$ of **edges**

Numbering the nodes from 1 to n, we can define the graph by a boolean array $E[i, j]$ for $i, j = 1, \ldots, n$, where $E[i, j]$ is true if and only if $(i, j) \in E$, that is, there is an edge from i to j

We will see two graph algorithms of which the design is strongly guided by the choice of an invariant: detecting celebrity and shortest path algorithm

---

84

Detecting celebrity

In a group of people a person C is called a **celebrity** if

- Every person knows C, and
- C knows no other people than only C itself

From this definition it is immediate that at most one celebrity may exist

The group of people can be described as a graph, in which $E[i, j]$ if and only if i knows j, where the people ar numbered from 1 to n

It is given that the group of people contains a celebrity c, can you detect it in at most $n - 1$ inspections of E?

85

Key observation:

- if $E[i, j]$ for $i \neq j$, then $i \neq c$
- if $\neg E[i, j]$ for $i \neq j$, then $j \neq c$

So for all $i \neq j$ we always may rule out i or j for being a candidate for c

Invariant:

$$I : \ i \leq c \leq j$$

It was given that celebrity c exists among $1, \ldots, n$, so for $i = 1$, $j = n$ the invariant I holds

---

86

If $E[i, j]$ then $i \neq c$, else $j \neq c$

Invariant I: $i \leq c \leq j$

Program T:

$i := 1; j := n;$
while $i \neq j$ do
    if $E[i, j]$ then $i := i + 1$ else $j := j - 1$

Now we will prove that the program T indeed satisfies $\{1 \leq c \leq n\}T\{i = c\}$, hence detects c

---

87

$i := 1; j := n;$
while $i \neq j$ do
    if $E[i, j]$ then $i := i + 1$ else $j := j - 1$

Invariant I: $i \leq c \leq j$

Initial condition:

$$\{1 \leq c \leq n\}i := 1; j := n; \{I\}$$

follows from $wp(i := 1; j := n, I) \equiv 1 \leq c \leq n$

88

Invariance:

We have to prove

$$\{I \wedge i \neq j\} \text{ if } E[i, j] \text{ then } i := i + 1 \text{ else } j := j - 1 \{I\}$$

that is

$$(I \wedge i \neq j) \rightarrow wp(\text{ if } E[i, j] \text{ then } i := i + 1 \text{ else } j := j - 1, I)$$

Note that $wp(\text{ if } E[i, j] \text{ then } i := i + 1 \text{ else } j := j - 1, I) \equiv$

$$(E(i, j) \rightarrow i + 1 \leq c \leq j) \wedge (\neg E(i, j) \rightarrow i \leq c \leq j - 1)$$

which follows from I since

- if $E(i, j)$ and $i \leq c \leq j$, then from $i \neq c$ we obtain $i + 1 \leq c \leq j$
- if $\neg E(i, j)$ and $i \leq c \leq j$, then from $j \neq c$ we obtain $i \leq c \leq j - 1$

89

Finally, we have to prove the end condition, that is

$$I \wedge \neg(i \neq j) \rightarrow i = c$$

which is easy since if $i \leq c \leq j$ and $i = j$, then $i = c$

So we have proved that our program T indeed satisfies $\{1 \leq c \leq n\}T\{i = c\}$, hence detects celebrity c

If the problem was not to find the celebrity that is given to exist, but to detect whether a celebrity exists, we can run the same program and afterwards check whether c satisfies the definition of celebrity

---

90

Shortest path algorithm

This celebrity problem is quite artificial, more practical (used billions of times in navigation systems) is the **shortest path algorithm**

A **path** from node i to node j is a sequence $i = i_0, i_1, \ldots, i_m = j$, such that $E[i_k, i_{k+1}]$ for all $k = 0, \ldots, m-1$

Assume every edge in a graph has a given length, then the length of a path is the sum of the lengths of the edges of the path

Given two nodes i, j what is the smallest length of a path from i to j?

We focus on the case where all edges have length 1; the general case can be obtained by a modification of the algorithm we give

---

91

Number the nodes from 1 to n starting in $i = 1$

We will not only compute the shortest length of a path from 1 to j, but from 1 to **any** node

More precisely, as the postcondition we choose

$$\forall k = 1, \ldots, n : D[k] = \text{ length of shortest path from 1 to } k$$

where this length is ∞ if no path from 1 to k exists

As the invariant we choose

$\forall k = 1, \ldots, n : D[k] =$ length of shortest path from 1 to k if it is $\leq m$, otherwise ∞

Easy to initialize for $m = 0$: $D[1] = 0$, $D[k] = \infty$ for $k \neq 0$

92

Invariant:

$\forall k = 1, \ldots, n : D[k] =$ length of shortest path from 1 to k if it is $\leq m$, otherwise ∞

Exploiting the fact that every path of length $m + 1$ is obtained by extending a path of length m by one extra edge, we obtain

$m := 0; D[1] := 0;$
for $k := 2$ to n do $D[k] := \infty;$
while $m \neq n - 1$ do
$\langle$ for $k := 1$ to n do
if $D[k] = m$ then
for $j := 1$ to n do
if $E[k, j] \wedge D[j] = \infty$ then $D[j] := m + 1;$
$m := m + 1; \rangle$

In this algorithm we write 'for $j := 1$ to n do if $E[k, j] \cdots$' to run over the outgoing edges of k, by representing the edges otherwise this can be much more efficient

By storing all nodes k with $D[k] = m$ more clever, further optimization is possible, and the algorithm will be **linear** in the number of edges

How about correctness?

Initialization and invariance we already considered, so it remains to show that $I \wedge \neg C \rightarrow Q$

$\neg C \equiv m = n - 1$, so by replacing m by $n - 1$ in the invariant, from

$\forall k = 1, \ldots, n : D[k] =$ length of shortest path from 1 to k if it is $\leq n - 1$, otherwise ∞

we should conclude that the shortest path to any node has been computed

Indeed this holds, since a shortest path from 1 to k is always $\leq n - 1$, since a path of length $\geq n$ involves at least n nodes, so at least one should occur at least twice, yielding a shorter path

Many details are omitted here, but this invariance argument is crucial for many variants, including the extension to arbitrary edge lengths that is extensively used in, e.g., navigation systems

---

Conclusions of this part on program correctness

Specification of a program S is given by a Hoare triple $\{P\}\ S\ \{Q\}$

For programs doing a fixed limited number of steps, **bounded model checking** is preferred, since it is fully automatic

A formula Φ is created describing the m steps, in which for every variable a a variable a_i is introduced representing the value of a after i steps

SAT/SMT is applied to $P_0 \wedge \Phi \wedge \neg Q_m$

If this is unsat, then the Hoare triple $\{P\}\ S\ \{Q\}$ holds

For large or unknown number of steps it does not work

96

We presented **Hoare logic**: rules to conclude $\{P\}\ S\ \{Q\}$ for building blocks S of programs

For assignment, composition and if-then-else we gave rules for computing **weakest precondition** $wp(S, Q)$

By the rule $\{P\}\ S\ \{Q\}\ \equiv\ (P \to wp(S, Q))$ then the Hoare triple $\{P\}\ S\ \{Q\}$ can be concluded

For a while loop S we have no rule for weakest precondition, but the Hoare triple $\{P\}\ S\ \{Q\}$ can be concluded from a proof rule exploiting an **invariant**

Unfortunately, choosing the right invariant is not suitable for automation

We focussed on **partial correctness**, that is, we prove $\{P\}S\{Q\}$ assuming that S is terminating

---

97

For the full correctness (called **total correctness**), also a proof of termination has to be given

Policy: first choose the invariant, then design the program accordingly

For small examples the three requirements for invariants can be checked without tool support, as we saw

For larger examples checking the requirements can be done by SMT solving

In the tool **Dafny** one specifies programs with several assertions, including invariants, and the corresponding requirements are automatically transformed to SMT and solved by Z3

About this machinery for verifying programs there will be an elective course **Provable Programming 2ITB0** in the second / third year