UNIVERSITY OF TWENTE.

VERIFICATION OF CONCURRENT AND DISTRIBUTED SOFTWARE

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

SEVENTH FRAMEWORK PROGRAMME

erc European Research Council

Established by the European Commission

OUTLINE OF THIS LECTURE

- How to ensure software quality?
- Classical program logic
- VerCors exercise
- Separation logic: reasoning about pointers
- The next challenge: concurrent software
- Permission-based separation logic
- VerCors exercise

- Verification of GPU kernels
- Reasoning about parallel blocks
- VerCors exercise
- Advanced verification features

Code voor exercises and some examples available from https://wwwhome.ewi.utwente.nl/~marieke/VTSA

SOFTWARE QUALITY

Peter Naur
1968
Working on the *Software crisis* report

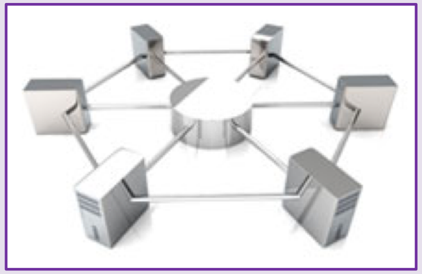
SOFTWARE IS EVERYWHERE

All software has errors!

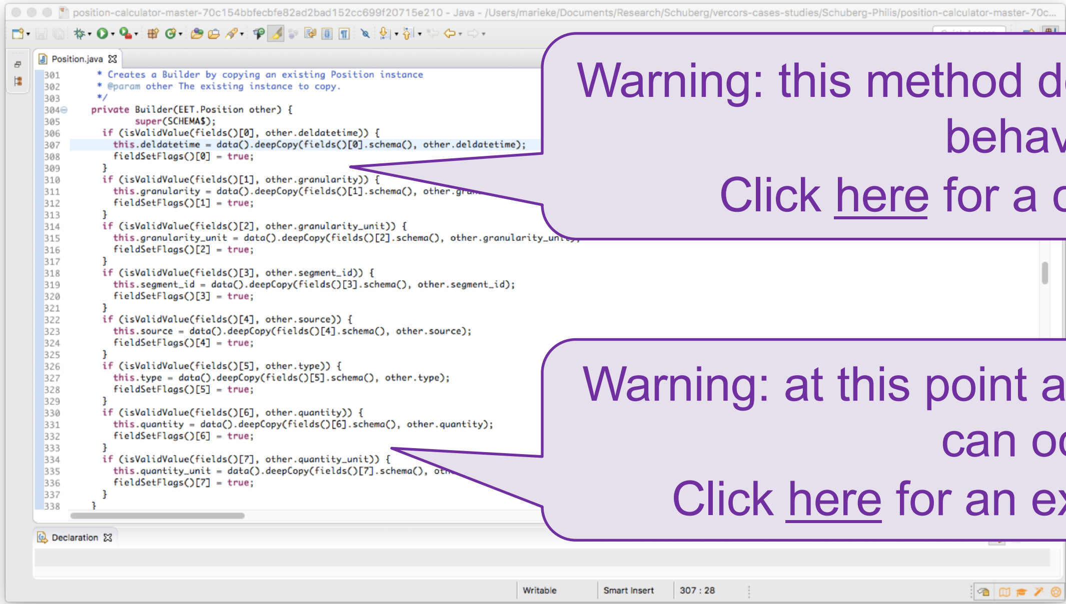
Software failures can have enormous impact

Gedateerd computersysteem veroorzaakte chaos op Britse luchthavens

An important challenge: Reliable software with parallel computations



VERIFICATION AS PART OF SOFTWARE DEVELOPMENT



Warning: this method does not have intended behaviour.
Click here for a counterexample

Warning: at this point a nullpointer exception can occur.
Click here for an example execution

Realising this dream requires substantial research

- Enlarge class of properties that can be established
- Automatic feedback

Dates back
to the 60-ies

SPECIFYING PROGRAM BEHAVIOUR

Use logic to describe behaviour of program components

- Precondition: what do you know in advance?
- Postcondition: what holds afterwards

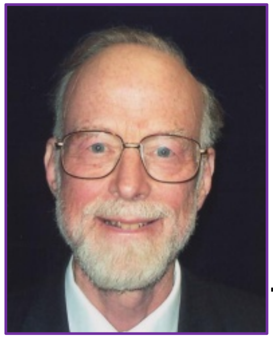
Example: increaseBy(int n)
requires n >= 0
ensures x == old(x) + n

Bob Floyd

Hoare triples + logic
Notation: {P}S{Q}
Syntactic verification of programs

precondition

postcondition



Tony Hoare

HISTORY OF PROGRAM VERIFICATION

Floyd - Hoare

Dijkstra

My thesis
(around 2000)

LOOP FOREVER

KeY

Krakatoa

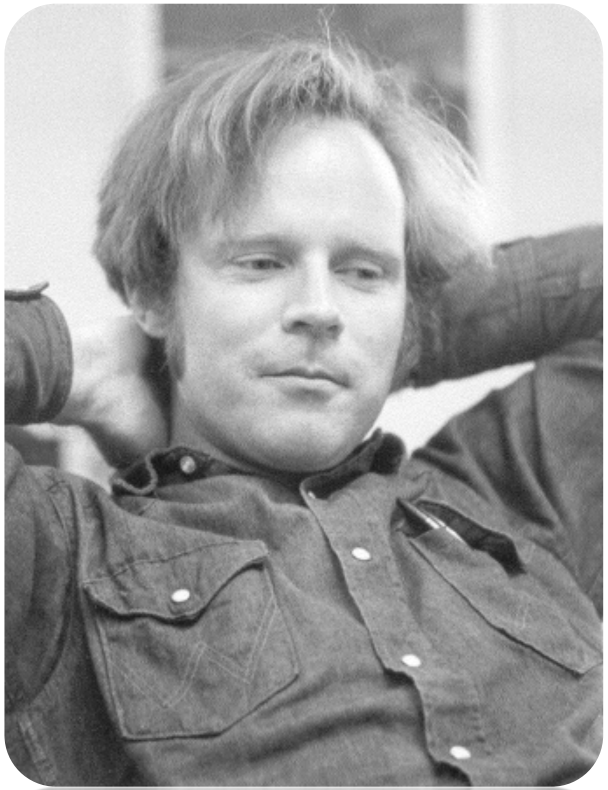
State-of-the-art

JML

Code Contracts

Dafny

PROGRAM LOGIC



Bob Floyd
1936 - 2001

PRE- AND POSTCONDITIONS

- Precondition: property that should be satisfied when method is called – otherwise correct functioning of method is not guaranteed
- Postcondition: property that method establishes – caller can assume this upon return of method

- Method specification is contract between implementer and caller of method.
  - Caller promises to call method only in states in which precondition holds
  - Implementer guarantees postcondition will be established

HOARE TRIPLES

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

- Due to Tony Hoare (1969)

- Meaning: if *P* holds in initial state *s*, and execution of *S* in *s* terminates in state *s′*, then *Q* holds in *s'*
- Formally:

$$\{P\}S\{Q\} = \forall s.P(s) \wedge (S,s) \rightarrow s' \Rightarrow Q(s')$$

HOARE LOGIC

- Hoare triples: specify behaviour of methods
- How to guarantee that methods indeed respect this behaviour?

- Collection of derivation rules to reason about Hoare triples
- Rules defined by induction on the program structure
- Proven sound w.r.t. program semantics

- Here: a very simple language, but exists for more complicated languages

AXIOMS

Skip $\dfrac{}{\{P\}Skip\{P\}}$

Ass. $\dfrac{}{\{P[v := e]\}v := e\{P\}}$

STATEMENT DECOMPOSITION

$$\text{Seq}\quad \frac{\{P\}S1\{Q\} \quad \{Q\}S2\{R\}}{\{P\}S1;S2\{R\}}$$

$$\text{If}\quad \frac{\{P \wedge b\}S1\{Q\} \quad \{P \wedge \neg b\}S2\{Q\}}{\{P\}\text{if } (b)\ S1 \text{ else } S2\ \{Q\}}$$

(*): precondition strengthening

EXAMPLE

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\ [z := 1] =$
$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1$

Ass

$\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1\}\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \Rightarrow a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1$

$a \geq 0 \wedge n \geq 0 \wedge k = 0\ [k := 0] =$
$a \geq 0 \wedge n \geq 0 \wedge 0 = 0$

Ass

$\{a \geq 0 \wedge n \geq 0 \wedge 0 = 0\}\ k := 0\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0\}$

$a \geq 0 \wedge n \geq 0 \Rightarrow a \geq 0 \wedge n \geq 0 \wedge 0 = 0$

(*)

$\{a \geq 0 \wedge n \geq 0\}\ k := 0\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0\}$

(*)

$\{a \geq 0 \wedge n \geq 0 \wedge k = 0\}\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$

Seq

$\{a \geq 0 \wedge n \geq 0\}\ k := 0;\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$

$\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$ while $(k < n)$ $\{z := z * a;\ k := k + 1;\}$ $\{z = a\text{^}n\}$

Seq

$\{a \geq 0 \wedge n \geq 0\}\ k := 0;\ z := 1;$ while $(k < n)$ $\{z := z * a;\ k := k + 1;\}$ $\{z = a\text{^}n\}$

RULES OF CONSEQUENCE

$$\text{Pre. Str.}\ \frac{P \Rightarrow P' \qquad \{P'\}S\{Q\}}{\{P\}S\{Q\}}$$

$$\text{Post. Weak.}\ \frac{\{P\}S\{Q\} \qquad Q \Rightarrow Q'}{\{P\}S\{Q'\}}$$

LOOPS

$$\text{Loop}\ \frac{\{I \wedge b\}S\{I\}}{\{I\}\text{while }(b)\ S\ \{I \wedge \neg b\}}$$

- I called loop invariant
- Preserved by every iteration of the loop
- Can in general not be found automatically

EXAMPLE: METHOD POWER

$\{ a \geq 0 \wedge n \geq 0 \}$

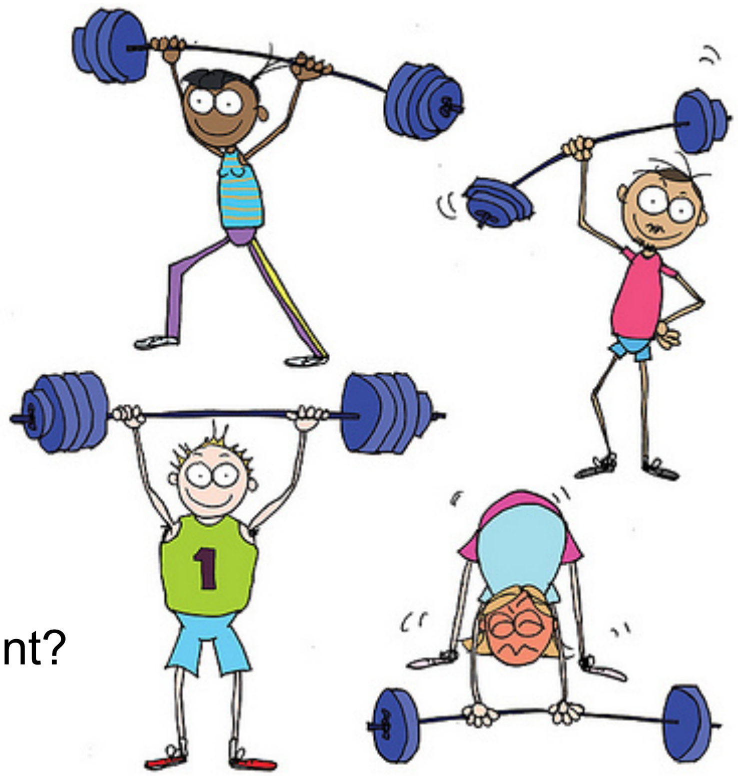
```
k := 0;
z := 1;
```

$\{ a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1 \}$

```
while (k < n)
   {  z := z * a;
      k := k + 1;
   }
```

$\{ z = a\char`^n \}$

What should be the loop invariant?



$z = a\char`^k \wedge k \leq n \wedge a \geq 0 \wedge k \geq 0$

EXAMPLE CONTINUED

Ass

$\{z*a = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$ z := z * a $\{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$

$z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n) \Rightarrow z*a = a\hat{}(k+1) \wedge a \geq 0 \wedge k + 1 \leq n$

Pre. Str.

Ass

$\{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$ k := k + 1 $\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n)\}$ z := z * a $\{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$

Seq

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n)\}$ z := z * a; k := k + 1 $\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$

Loop

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$ while (!(k = n)) {z := z * a; k := k + 1;} $\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge k = n\}$

$z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge k = n \Rightarrow z = a\hat{}n$

Post. Weak.

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$ while (!(k = n)) {z := z * a; k := k + 1;} $\{z = a\hat{}n\}$

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1 \Rightarrow z = a\hat{}k \wedge k \leq n \wedge a \geq 0$

Pre. Str.

$\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$ while (!(k = n)) {z := z * a; k := k + 1;} $\{z = a\hat{}n\}$

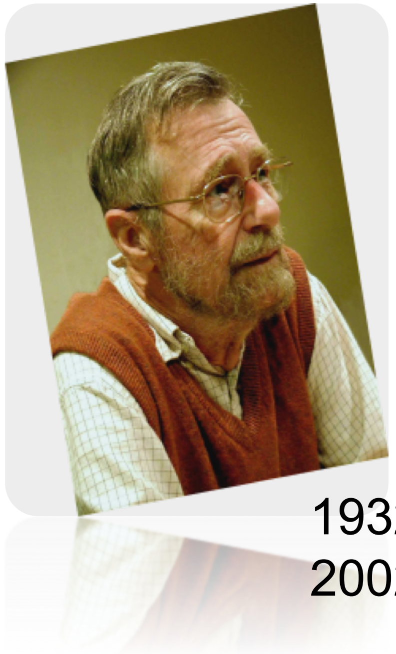
TOOL SUPPORT FOR PROGRAM VERIFICATION

Rustan Leino

A CALCULATIONAL APPROACH

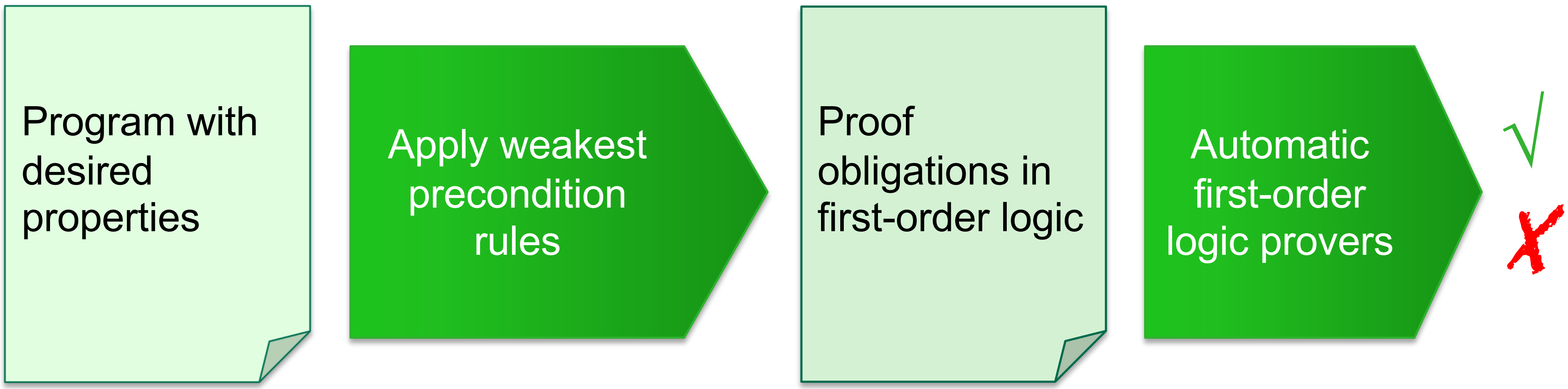
Many intermediate predicates can be computed

- Weakest liberal precondition wp(S,Q)
- The weakest predicate such that {wp(S,Q)}S{Q}
- Due to Edsger Dijkstra (1975)
- Calculus allows to compute weakest preconditions of sequential code
- Proof obligations: preconditions imply weakest liberal preconditions
- Loop invariants still given explicitly



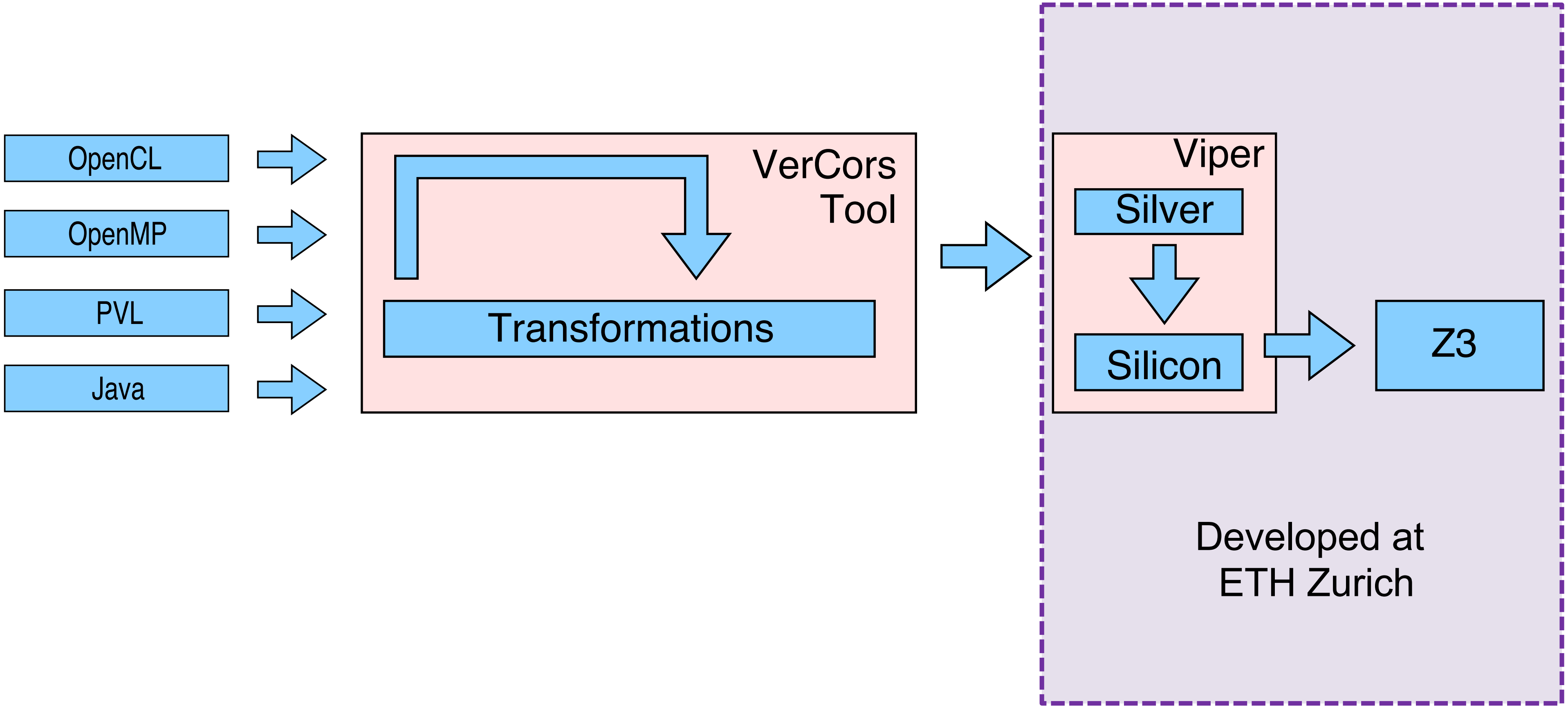
1932 - 2002

AUTOMATION



Preferably also counter example: why does program not have desired behaviour

VERCORS TOOL ARCHITECTURE



See iFM 2017

PROGRAM CORRECTNESS IN VERCORS

- PVL syntax: https://github.com/utwente-fmt/vercors/wiki/PVL-Syntax
- Two kinds of verification:
  - Memory safety (postcondition true), method will terminate without exceptions
  - Functional correctness: postcondition expresses something about poststate of the method

- Two useful abbreviations
  - Context: pre- and postcondition
  - Invariant: pre- and postcondition, and at loop entry and exit

EXERCISES

Code voor exercises and some examples available from
https://wwwhome.ewi.utwente.nl/~marieke/VTSA

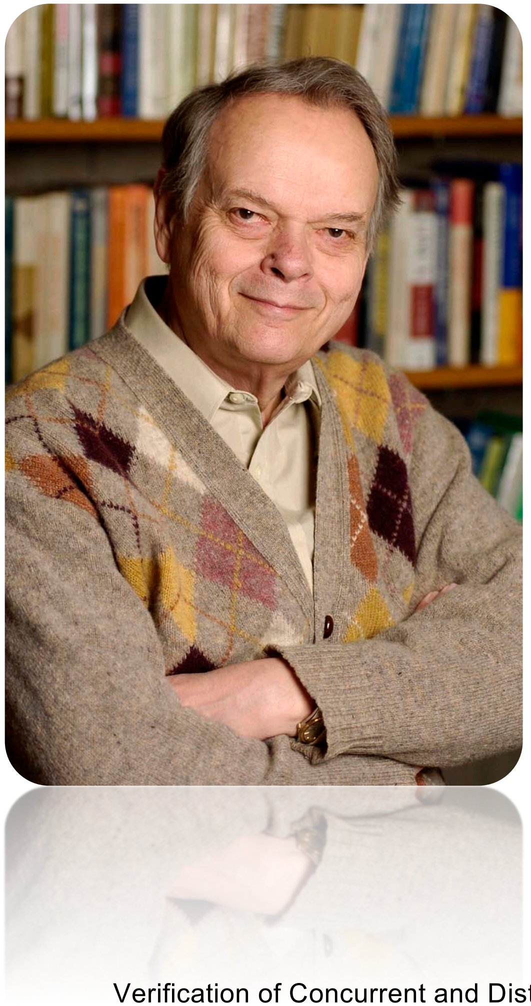
LIMITATIONS OF CLASSICAL PROGRAM LOGIC

- Idealised language
- No side-effects in conditions
- No pointers
- No multi-threading

Separation logic

- Reasoning about pointers
- Natural extension to multi-threading

SEPARATION LOGIC



John Reynolds
1935 - 2013

THE CHALLENGE OF POINTER PROGRAMS

```
class C {

  D f;
  D g;
}

class D {
  int x;
  D() {
    x  = 0;
 }
```

```
ensures c.g.x == 0;
void m(C c) {
  d = new D;
  c.f = d;
  c.g = d;
  update_x(c.f, 3);
}

ensures d.x == v;
void update_x(D d, int v) {
  d.x = v;
}
}
```

This should not be verified!

SEPARATION LOGIC

- State distinguishes heap and store
- Heap contains dynamically allocated data that exists during run-time of program

  (Object-oriented program: the objects are stored on the heap)
- Store (or call stack) contains data related to method call (parameters, local variables)
- Heap accessed by pointers
- Locations on heap can be aliased
- Main idea: assertions about state can be decomposed into assertions about disjoint substates

INTUITIONISTIC SEPARATION LOGIC

Syntax extension of predicate logic:

$\varphi ::= e.f \rightarrow e' \mid \varphi \ast \varphi \mid \varphi -\!\ast\ \varphi \mid \ldots$

where *e* is an expression, and *f* a field

Meaning:

- $e.f \rightarrow e'$ – heap contains location pointed to by *e.f,* containing the value given by the meaning *e'*
- $\varphi 1 \ast \varphi 2$ – heap can be split in disjoint parts, satisfying φ1 and φ2, respectively
- $\varphi 1 -\!\ast\ \varphi 2$ – if heap extended with part that satisfies φ1, composition satisfies φ2

Monotone w.r.t. extensions of the heap

Magic wand not frequently used

ADVANTAGES OF SEPARATION LOGIC

- Reasoning about programs with pointers
- Two interpretations $e.f \rightarrow v$
  - Field $e.f$ contains value v
  - Permission to access field $e.f$

    A field can only be accessed or written if $e.f \rightarrow$ _ holds!
- Implicit disjointness of parts of the heap allows reasoning about (absence) of aliasing

  $x.f \rightarrow$ _ $\ast$ $y.f \rightarrow$ _ implicitly says that x and y are not aliases
- Local reasoning
  - only reason about heap that is actually accessed by code fragment
  - rest of heap is implicitly unaffected: frame rule

PROOF RULE FOR UPDATES OF THE HEAP

$$\frac{}{\{e.f \rightarrow \_\}\ e.f = v\ \{e.f \rightarrow v\}}$$

- For simplicity *v* is typically assumed to be a simple (unqualified) expression
- Any assignment *e.f = e'.g* can be split up in *x = e'.g; e.f = x*

EXAMPLE: CLASS BOX

class Box {

int cnts;

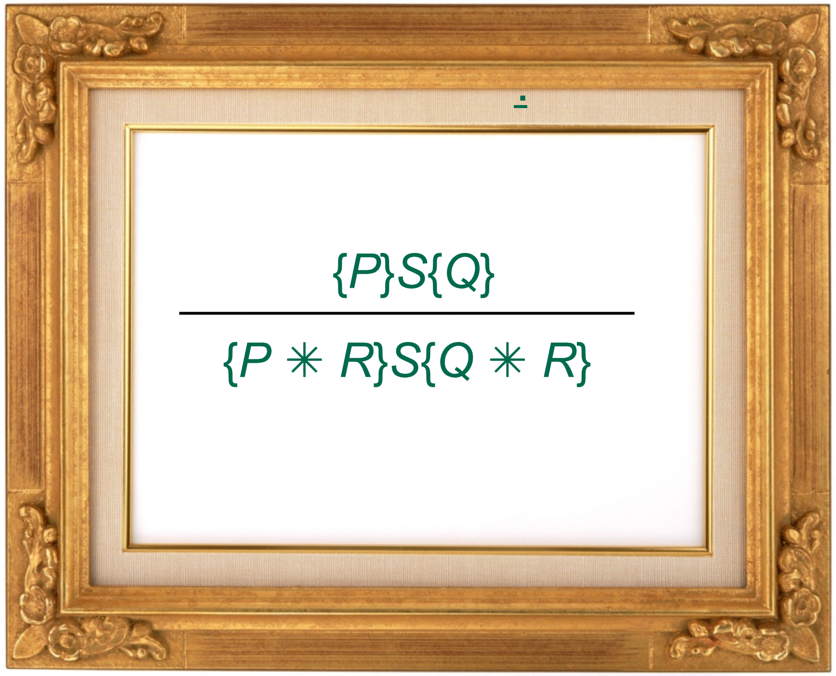
requires this.cnts $\rightarrow$ _;
ensures this.cnts $\rightarrow$ o;
void set (int o) {
    this.cnts = o;
    return null;
}

requires this.cnts $\rightarrow$ X;
ensures this.cnts $\rightarrow$ X $\wedge$ result = X;
int get() {
    return this.cnts;
}

}

Compare with specifications in classical Hoare logic
requires true;
ensures this.cnts == o;

FRAME RULE



$$\frac{\{P\}S\{Q\}}{\{P \ast R\}S\{Q \ast R\}}$$

where R does not contain any variable that is modified by S.

THE CHALLENGE OF POINTER PROGRAMS

```
class C {
  D f;
  D g;
}

class D {
  int x;

  D() {
    x = 0;
  }
```

```
ensures c.g.x == 0;
void m(C c) {
  d = new D;
  c.f = d;
  c.g = d;
  update_x(c.f, 3);
}

ensures d.x == v;
void update_x(D d, int v) {
  d.x = v;
}
```

$c.f \rightarrow \_ \ast c.g \rightarrow \_$ does not hold

Empty frame

SEPARATION LOGIC VS IMPLICIT DYNAMIC FRAMES

- Classical separation logic: this.cnts $\rightarrow$ X
- Implicit dynamic frames: Perm(this.cnts) $\ast$ this.cnts == X
- VerCors: Perm(this.cnts, write) ** this.cnts == X

BOX IN VERCORS

```
class Box {

  int cnts;

  requires Perm(this.cnts, write);
  ensures Perm(this.cnts, write);
  void setCnts (int o) {
    this.cnts = o;
  }

given int x;
requires Perm(this.cnts, write) **
              this.cnts == x;
ensures Perm(this.cnts, write) **
              \result == x;
  int getCnts () {
    return this.cnts;
  }
}
```

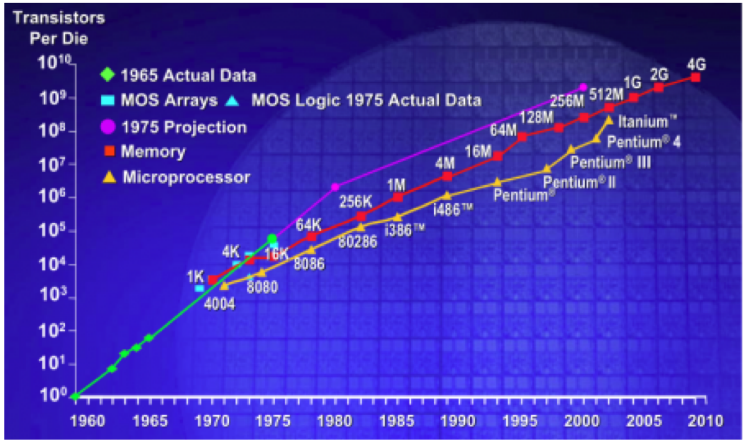
Given: ghost parameter

CONCURRENCY: THE NEXT CHALLENGE

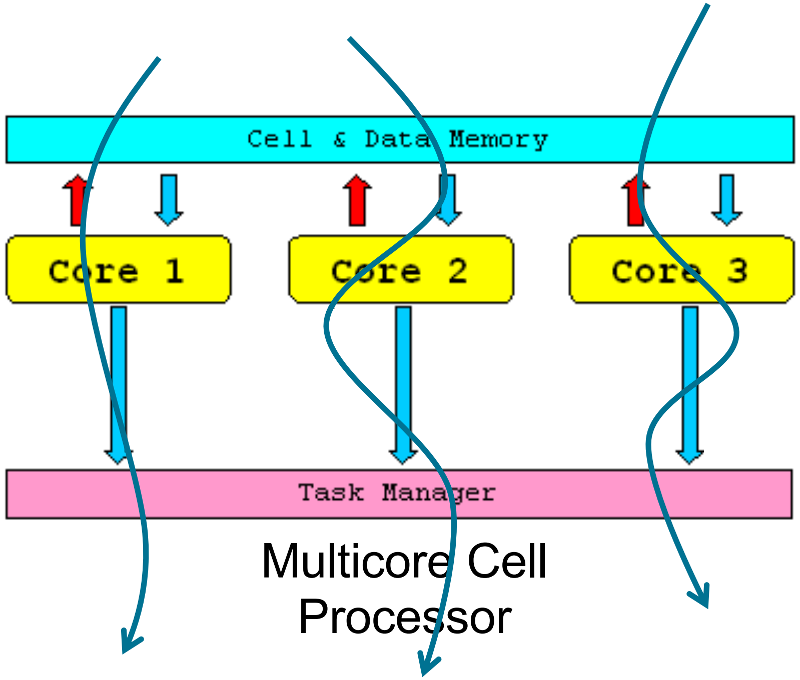
Doug Lea

THE FUTURE OF COMPUTING IS MULTICORE

Single core processors:
The end of Moore's law



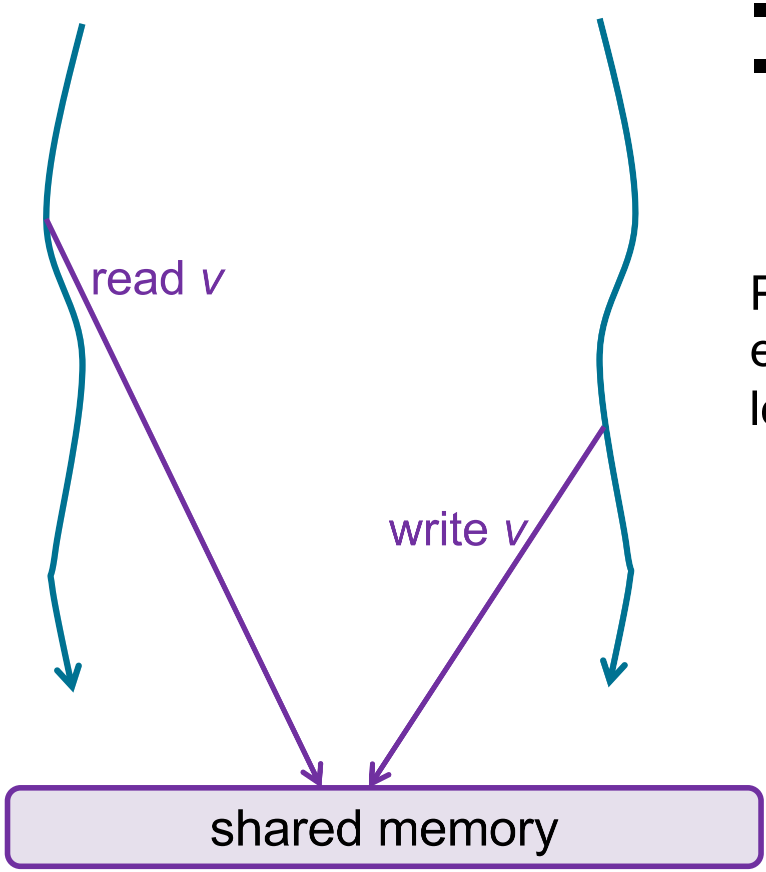
Solution:
Multi-core processors



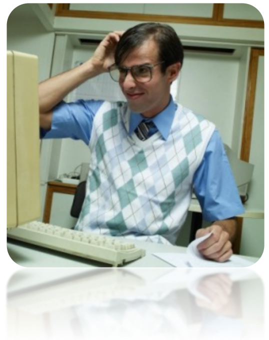
Multiple threads of execution

Coordination problem shifts from hardware to software

MULTIPLE THREADS CAUSE PROBLEMS



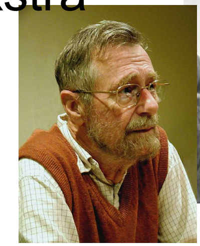
- Order?
- More threads?



Possible consequences:
errors such as data races caused
lethal bugs as in Therac-25

VERIFICATION OF MULTITHREADED PROGRAMS

Dijkstra



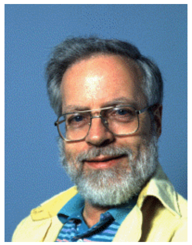
Floyd - Hoare

Concurrency
(multithreading)

LOOP FOREVER

Krakatoa

Owicki - Gries



Jones

O'Hearn

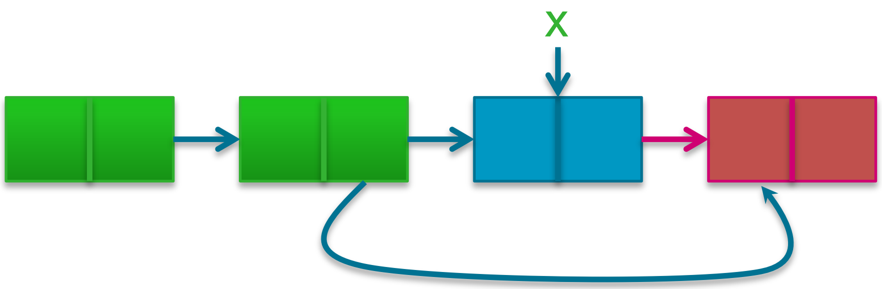
2004
separation logic

VerCors

Viper
Verifast
Iris
TaDa
Flow
...

SPECIFICATIONS IN A CONCURRENT SETTING

```
requires true
ensures x is the last element in the list
void addToList(Elem x) {
    // code
}
```



Any other thread might invalidate this!

'x is in the list' cannot even be guaranteed!

Except when no other thread can update the list

SOME HISTORY: REASONING ABOUT THREADS

Susan Owicki

OWICKI-GRIES METHOD (1975)

- For each thread: give a complete proof outline
- Verify each thread w.r.t. the proof outline
- For each annotation in the proof outline, show that it cannot be invalidated by any other thread: interference freedom

David Gries

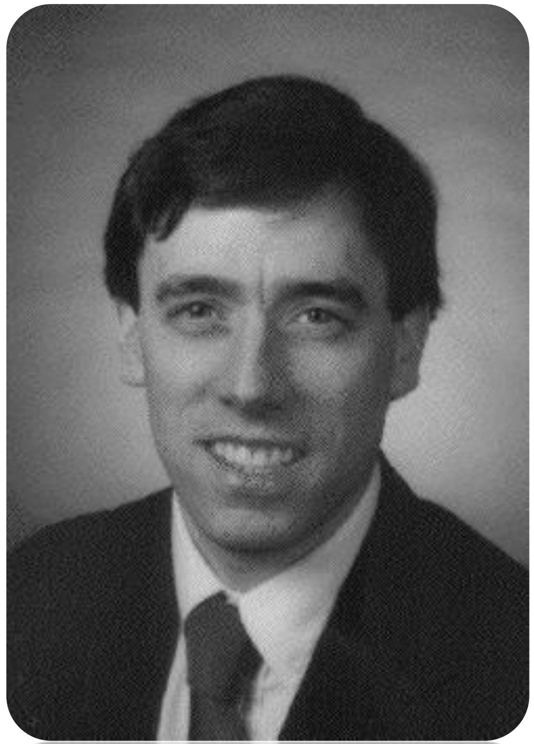
RELY-GUARANTEE METHOD

- Jones (1980)
- Compositional
- For each thread, specify
  - what it assumes from other threads
  - what it guarantees to other threads

Rely: what transitions may other threads make
Guarantee: what transitions may current thread make

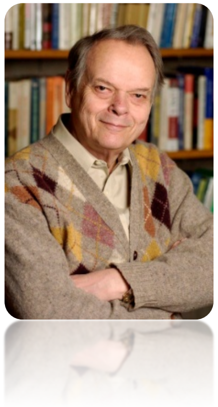
$$\frac{\begin{array}{c} rely \vee guar1 \Rightarrow rely2 \\ rely \vee guar2 \Rightarrow rely1 \\ guar1 \vee guar2 \Rightarrow guar \\ \langle relyi, guari \rangle : \{Pi\}\ Si\ \{Qi\},\ i = 1,2 \end{array}}{\langle rely, guar \rangle : \{P\}\ S1\ ||\ S2\ \{Q\}}$$

CONCURRENT SEPARATION LOGIC



John Boyland

JOHN REYNOLDS'S 70TH BIRTHDAY PRESENT



$$\frac{\{P1\}S1\{Q1\} \quad \ldots\ldots\ldots\ldots \{Pn\}Sn\{Qn\}}{\{P1 \ast \ldots \ast Pn\}\ S1 \parallel \ldots \parallel Sn\ \{Q1 \ast \ldots \ast Qn\}}$$

where no variable free in *Pi* or *Qi* is changed in *Sj* (if $i \neq j$)

EXAMPLE

$$\dfrac{\dfrac{\vdots}{\{x = 0\}x := x + 1; x := x + 1\{x = 2\}} \qquad \dfrac{\vdots}{\{y = 0\}\ y := y + 1; y := y + 1\ \{y = 2\}}}{\{x = 0 \ast y = 0\}x := x + 1; x := x + 1 \mid\mid y := y + 1; y := y + 1\ \{x = 2 \ast y = 2\}}$$

No interference between the threads

WHY IS THIS NOT SUFFICIENT?

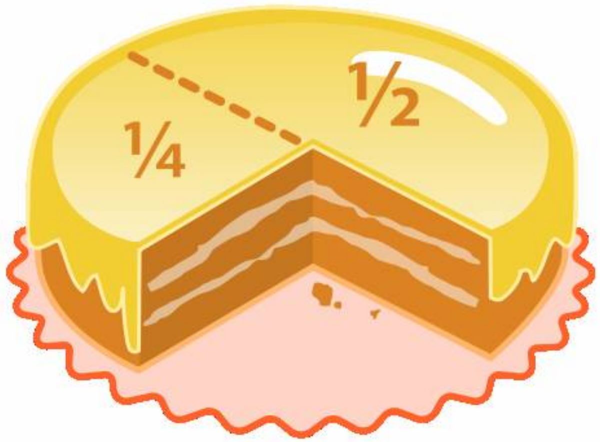
- Simultaneous reads not allowed

  1. Distinguish between read and write accesses

- Number of parallel threads is fixed

PERMISSIONS

- Permission to access a variable
- Value between 0 and 1
- Full permission 1 allows to change the variable
- Fractional permission in (0, 1) allows to inspect a variable
- Points-to predicate decorated with a permission
- Global invariant: for each variable, the sum of all the permissions in the system is never more than 1
- Permissions can be split and combined



EXAMPLE

Permissions on n equally distributed over threads

$$\frac{\dfrac{\vdots}{\{PointsTo(x,1,0) \ast Perm(n, ½)\}\ x := x + n;\ x := x + n\ \{PointsTo(x,1,2*n) \ast Perm(n, ½)\}} \quad \dfrac{\vdots}{\{PointsTo(y,1,0) \ast Perm(n, ½)\}\ y := y + n;\ y := y + n\ \{PointsTo(y,1,2*n) \ast Perm(n, ½)\}}}{\{PointsTo(x,1,0) \ast PointsTo(y,1,0) \ast Perm(n,1)\}\ x := x + n;\ x := x + n \mid\mid y := y + n;\ y := y + n\ \{PointsTo(x,1,2*n) \ast PointsTo(y,1,2*n) \ast Perm(n,1)\}\}}$$

Perm(x,1) = Perm(x, ½) ✳ Perm(x, ½)

Shared variable is only read
No interference between the threads

WHY IS THIS NOT SUFFICIENT?

- Simultaneous reads not allowed

1. Distinguish between read and write accesses

- Number of parallel threads is fixed

2. Dynamic thread creation

Thread specifications indicate how permissions should be distributed

EXAMPLE

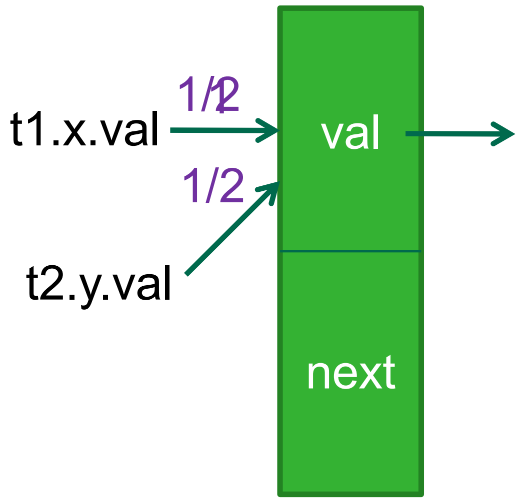
```
class List {
      int val; List next;
       ...
}

class T {
      List y;
      void run() { ... }
}
```

t1

```
x := new List;
x.val := ...;
t2 := new T;
t2.y := x;
fork t2;
read x.val;
...


join t2;
x.val := ...;
```

t2

```
run(){
...
read y.val
...
}
```



SPECIFICATION FOR RUN METHOD IN T2

requires Perm (*y*.val, ½);
ensures Perm(*y*.val, ½);
void run() {....}

- Forking thread has to give up required permissions
- Joining thread gains back ensured permissions

What happens if run is specified as follows:

requires Perm(*y*.val, 1);
ensures Perm(*y*.val, 1);;
void run() {....}

EXAMPLE

```
class List {
    int val; List next;
    ...
}

class T {
    List y;
    void run() { ... }
}
```

t1

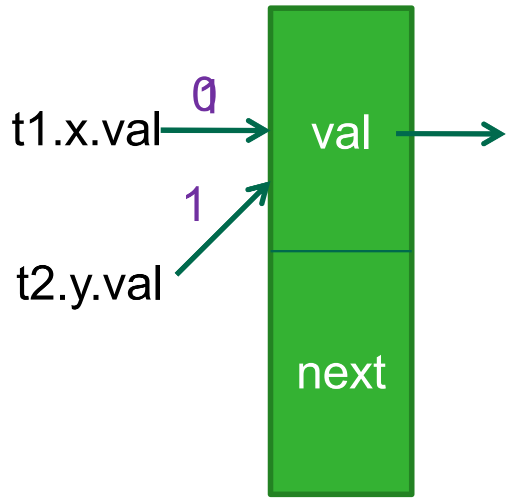
```
x := new List;
x.val := ...;
t2 := new T;
t2.y := x;
fork t2();
read x.val;
...

join t2;
read x.val;
x.val := ...;
```

NOT ALLOWED!

Now the permissions are back

t2

```
run(){
...
read y.val
...
}
```



RESOURCE INVARIANT – CLASSICAL APPROACH

- Lock *x* acquired and released with lock *x* and unlock *x*
- Each lock has associated resource invariant
- Lock acquired ⟹ resource invariant lend to thread
- Lock released ⟹ resource invariant taken back from thread
- Class Object contains predicate

  resource lock_invariant() = true;
- In rules: if *I* is resource invariant of *x*

  {true} lock *x* {*I*}

  {*I*}unlock *x*{true}
- This is sound only for single-entrant locks

{true}
lock *x*;
{*I*}
lock *x*;
{*I* ∗ *I*}
...

Resource *I* has been duplicated!

LOCKS IN PVL

```
class locktest {

resource lock_invariant() =
          Perm(x, 1/2);

  int x;

  locktest(int y) {
    x = y;
  }
```

```
  void m() {
    lock this;
    int v = x;
    unlock this;
  }
}
```

Perm(x, write) = Perm(x, 1)
Perm(x, read) = Perm(x, v) for some v

EXERCISES

Code voor exercises and some examples available from
https://wwwhome.ewi.utwente.nl/~marieke/VTSA