UNIVERSITY OF TWENTE.

RELIABLE CONCURRENT SOFTWARE

MARIEKE HUISMAN
UNIVERSITY OF TWENTE, NETHERLANDS

OUTLINE OF THIS LECTURE

- How to ensure software reliability?
- Classical program logic
  - Verification at compile-time
  - Verification at run-time
- The next challenge: concurrent software
- Permission-based separation logic
  - Compile-time verification of concurrent programs
  - Run-time verification of concurrent programs

SOFTWARE IS EVERYWHERE

- Organisations spend $332 billion on software in 2016 (and this number increases every year)
- Large part of development effort goes into bug fixing, maintenance, re-understanding software
- Software is too complicated to fully understand its behaviour by manual code inspection
- Software updates might break the software in other places

THE SOFTWARE QUALITY PROBLEM IS AS OLD AS SOFTWARE ITSELF

Peter Naur
1968
Working on the *Software crisis* report

SOFTWARE QUALITY NOWADAYS

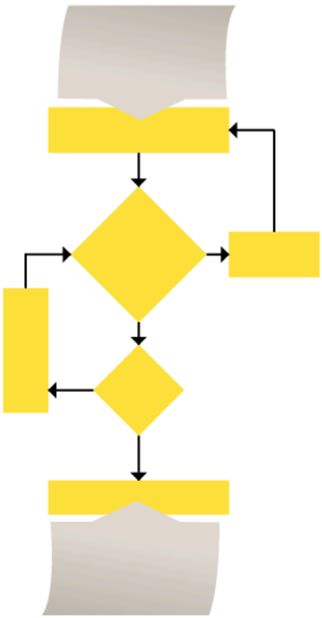
ICT problems Dutch gouvernment

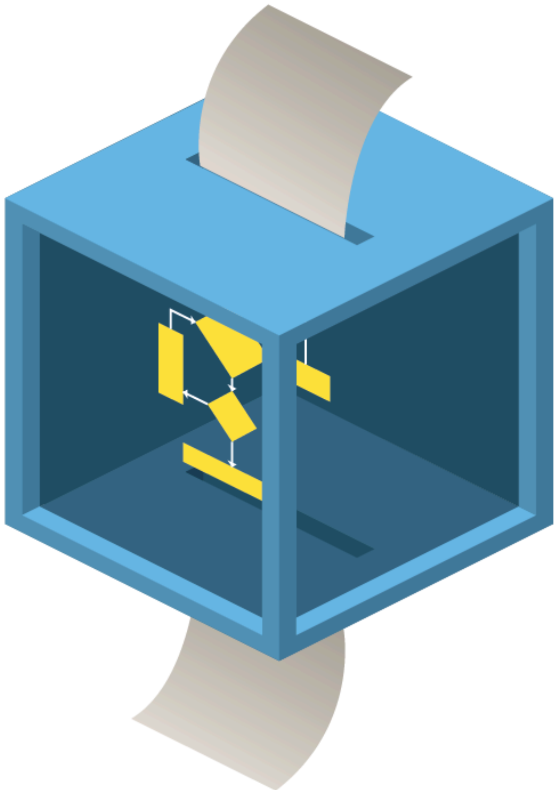
Toyata Prius: software errors due to lack of testing

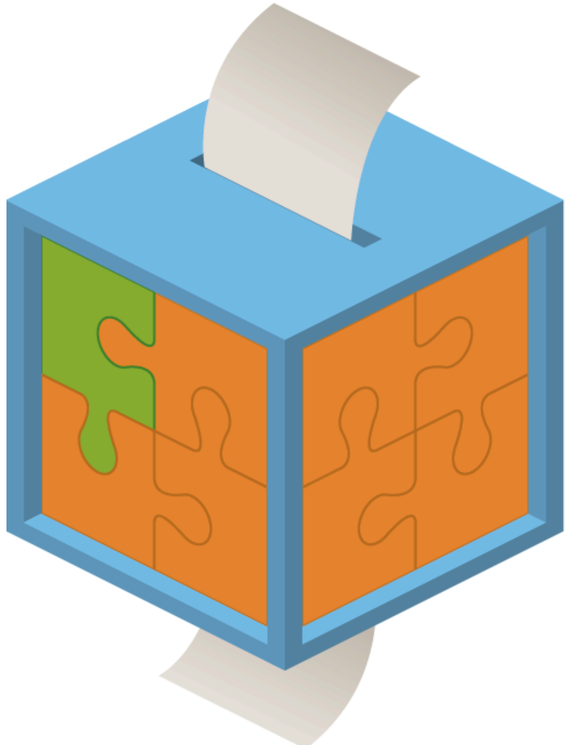
Unreachable banks because of network problems

Mars Climate Orbiter: Crash due to different units

OUR APPROACH







Software

Box it

Check the components

SPECIFYING PROGRAM BEHAVIOUR

Use logic to describe behaviour of program components

- Precondition: what do you know in advance?

  Example: increaseBy(int n)

  requires n > 0

- Postcondition: what holds afterwards

  Example: increaseBy(int n)

  x increased by n

  ensures x == old(x) + n

Bob Floyd (1936 – 2001)

Dates back to the 60-ies

Tony Hoare

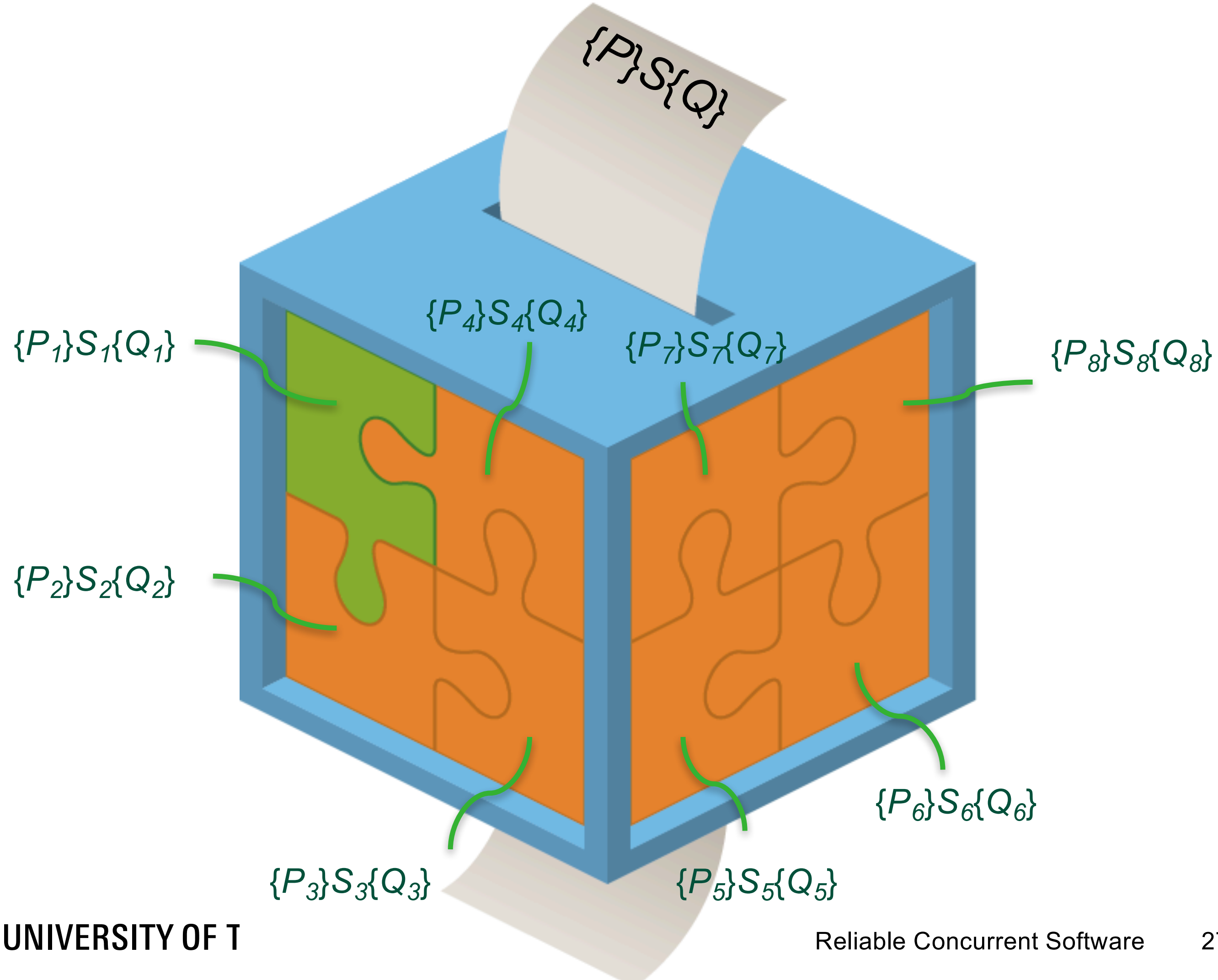
Hoare triples

Notation: {P}S{Q}

precondition

postcondition

HOARE TRIPLES FOR ALL COMPONENTS

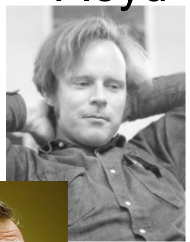


HISTORY OF PROGRAM VERIFICATION

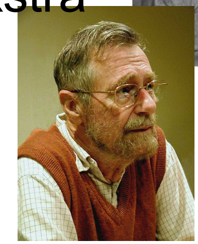
My thesis
(around 2000)

State-of-the-art

Floyd - Hoare



Dijkstra



LOOP FOREVER

KeY

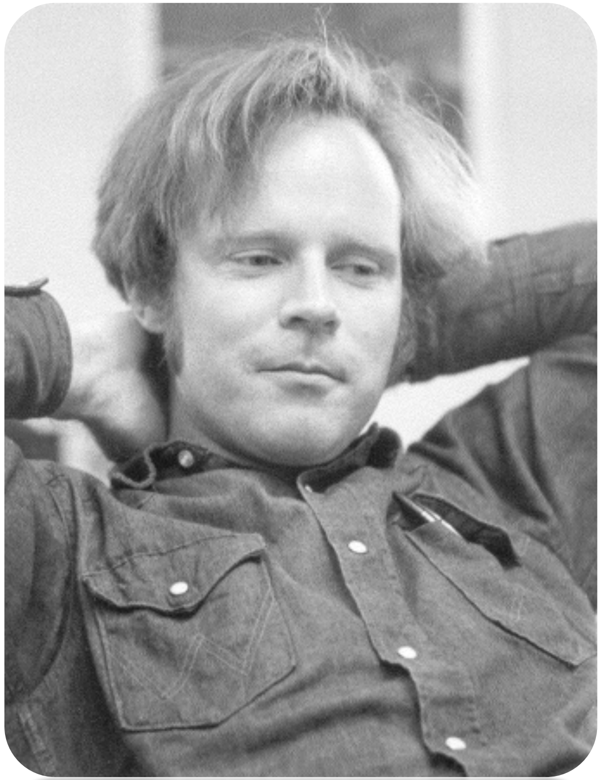
Krakatoa

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

SOME EXAMPLE PROOF RULES

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

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

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

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

- Notation in our language
  invariant I;
  while (*b*) *S*

EXAMPLE: METHOD POWER

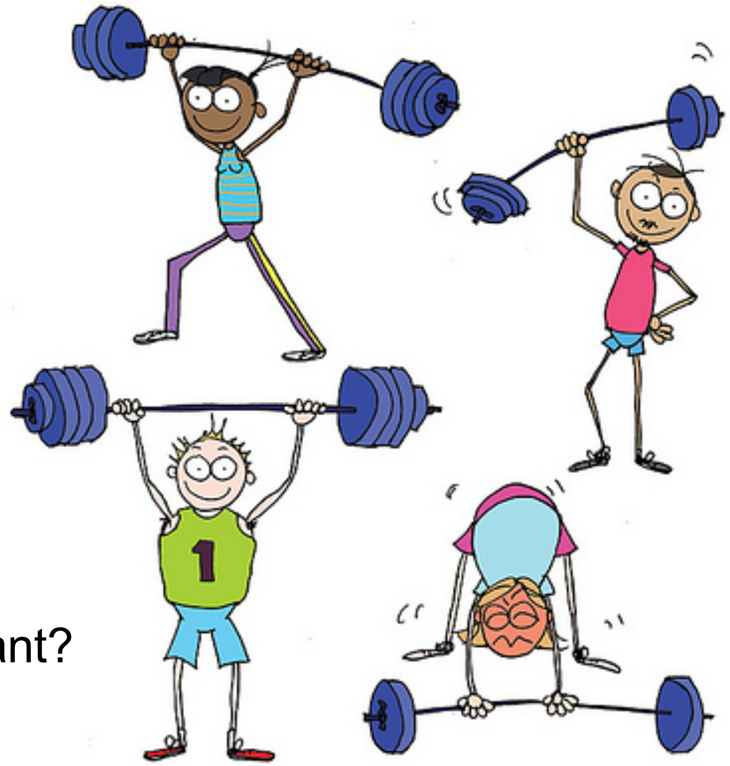
$\{ a \geq 0 \wedge n \geq 0 \}$

```
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;
  }
```

$\{ z = a\hat{\ }n \}$



What should be the loop invariant?

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

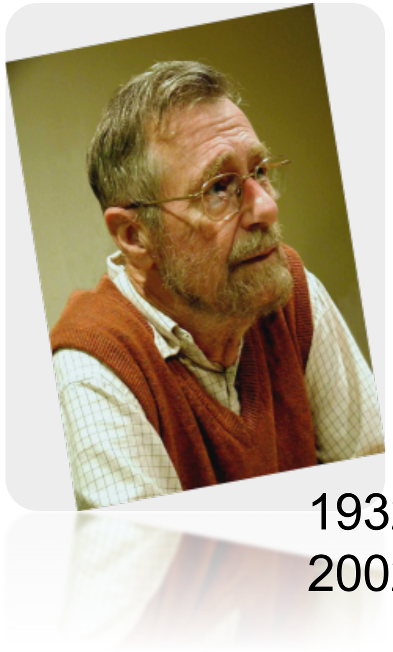
TOOL SUPPORT FOR PROGRAM VERIFICATION

Rustan Leino

A CALCULATIONAL APPROACH

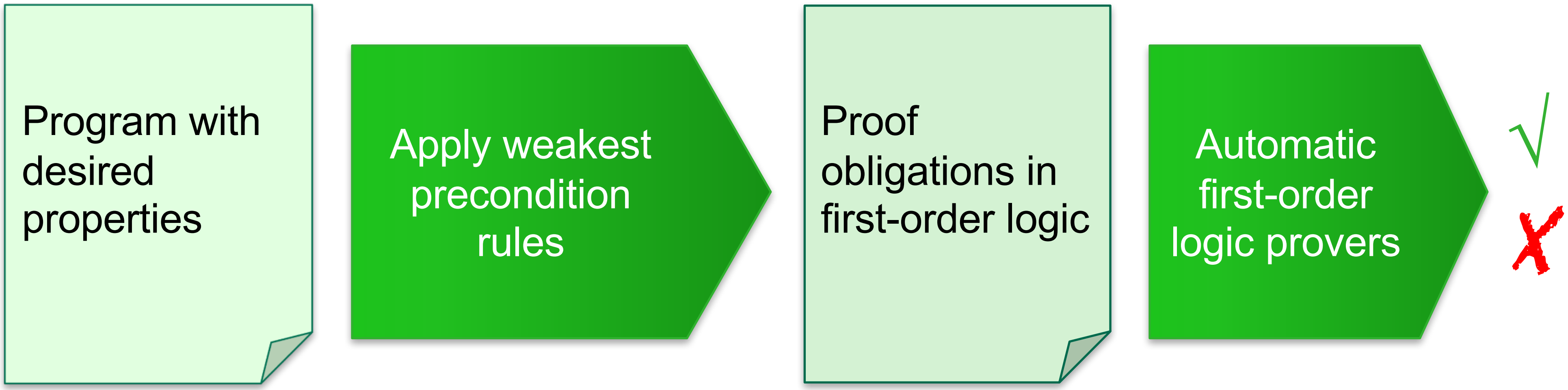
Many intermediate predicates can be computed

- 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

Alternative: perform symbolic evaluation (forward reasoning)

VALIDITY OF SPECIFICATION AT RUNTIME?

```
requires P;
ensures Q;
.... method() {
      body;
   }
```

→

```
… method() {
      assert P;
      body;
      assert  Q;
}
```

What would be the difficulties?

CHALLENGES TO DO THIS SYSTEMATICALLY

- Changes the program source
- Methods with multiple exit points
- Exceptional postconditions
- Specification-only expressions can not be used in Java assert (as they are not in Java)
- Executability of specifications
- Class-level specifications

A lot of engineering…
and some research

IMPLEMENTATION

CHEON & LEAVENS

- Method bodies wrapped in specification checks
- Method body wrapped in try-catch-finally to check exceptional postconditions

Yoonsik Cheon
JML2

Challenges addressed

- Undefinedness (0/x)
- Executability of specifications
- Quantified expressions
- \old-expressions

David Cok
OpenJML

REQUIREMENTS ON RUN-TIME ASSERTION CHECKER

- Transparency:

  If there are no annotation violations detected, then

  behaviour with and without run-time checker should be equivalent
- Isolation:

  Annotation violation reported when it occurs
- Thrustworthy:

  Do not report false annotation violations

JML RUN-TIME ASSERTION CHECKER

- Special compilation option
- Inserts tests at appropriate points
- Pre-deployment usage
  - Execution with run-time checks enabled during debugging phase
  - Final version: without run-time checks
- Post-deployment usage
  - Monitoring for unwanted situations
  - Reducing overhead is crucial

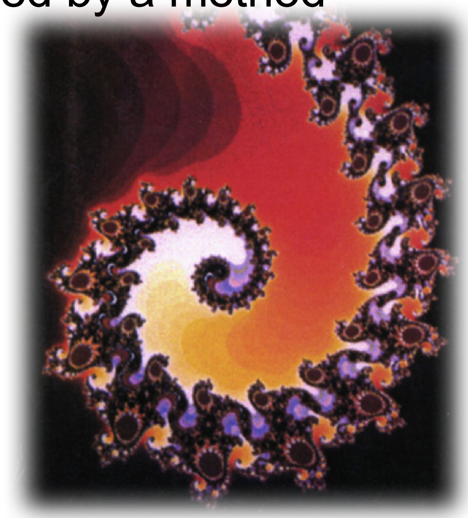
RUN-TIME VS. STATIC CHECKING

properties	run-time	static
data	run-time assertion checking	deductive verification
traces	runtime verification	model checking

Challenge: how to combine reasoning about data and traces?

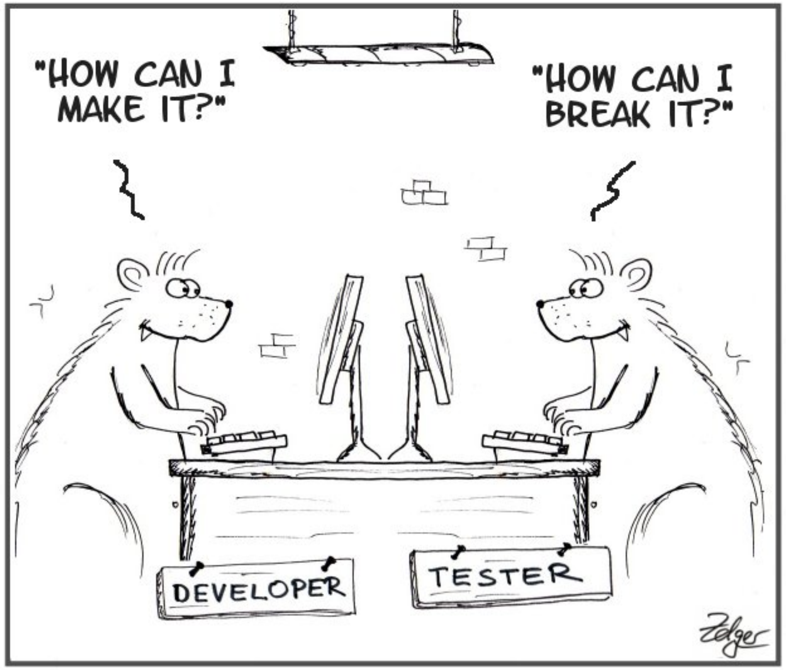
LIMITATIONS OF RUN-TIME CHECKING

- Only checks concrete executions
- Only executable specifications can be checked
- Problematic: unbounded quantifications over all objects
- Assignable clauses: which variables are modified by a method



RUN-TIME ASSERTION CHECKING = EXTENDED TESTING

- Test plan describes what aspects of program will be tested
- Specifications give idea about interesting corner cases
- Test coverage should also consider specifications



They weren't so much different,
but they had different goals

JMLUnit(NG)

UNIT TESTING CHALLENGES

- Write the test
  - Code to check the outcome – test oracle
  - Choose input data
- Test coverage
  - Are all execution paths exercised?
  - Are there any inputs that can cause abnormal behaviour?
- Time consuming
  - Testing tends to take more time than coding

JML specifications

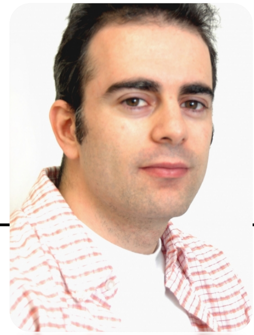
- Machine readable description of intended method behaviour
- With execution mechanism (RAC)

BASIC IDEA

- Use JML Specs as Tests/Test Oracles
- Take the input test data, evaluate precondition
  - If true: run the method with input data
  - If false: skip – meaningless test
- After execution of the method evaluate the postcondition
  - If true: test passed
  - If false: test fails, quote the values of the input data
- JMLUnitNG: Make this process automatic

In essence:
Promoting RAC to unit testing

Daniel Zimmerman Rinkesh Nagmoti

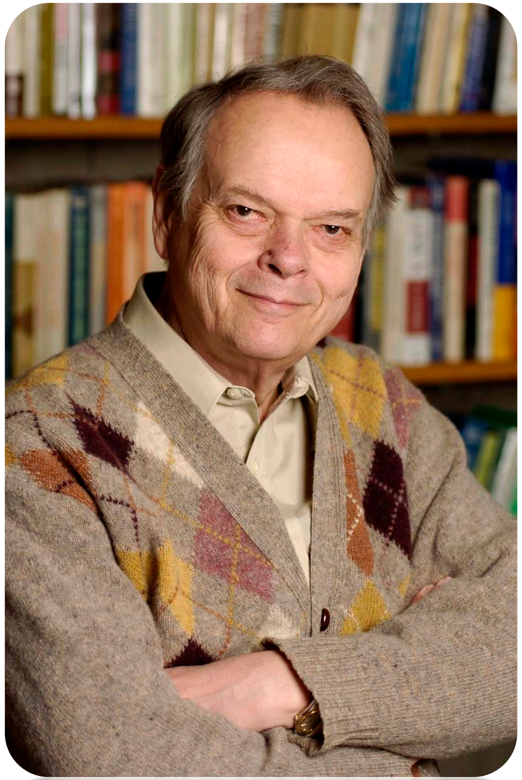


JMLUNIT NEW GENERATION

- Comprehensive JML based testing framework
- Core test generator
  - Collect classes and methods with JML specifications
  - Data generators with templates for manual input
  - Create testing structure for everything
- Runtime Assertion Checker (RAC) compiler
  - Embed JML checks into compiled Java code
  - Report results of evaluating JML expressions to the testing framework
- Result: a standalone test suite based on the TestNG engine

Efficient with good coverage

SEPARATION LOGIC



John Reynolds
1935 - 2013

THE CHALLENGE OF POINTER PROGRAMS

```
class C {

  D f;
  D g;
}

class D {
  int x := 0;
}
```

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

ensures d.x = v;
method update_x(d, 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

UPDATES AND LOOKUP OF THE HEAP

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

$$\frac{}{\{X = e \wedge X.f \rightarrow Y\} v := e.f\ \{X.f \rightarrow Y \wedge v = Y\}}$$

where X and Y are logical variables

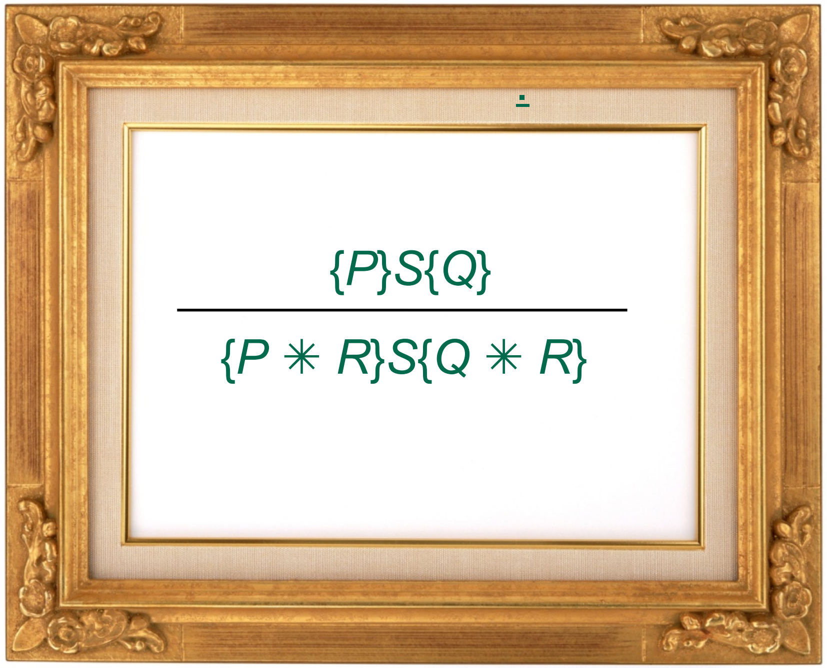
- Two interpretations $e.f \rightarrow v$
  - Field $e.f$ contains value v
  - Permission to access field $e.f$

    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

FRAME RULE

Local reasoning
only reason about heap that is actually accessed by code fragment
rest of heap is implicitly unaffected



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

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 := 0;
}
```

```
method m() {
  c := new C;
  d := new D;
  c.f := d;
  c.g := d;
  update_x(c.f, 3);
}

ensures d.x = v;
method update_x(d, v) {
  d.x := v;
}
```

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

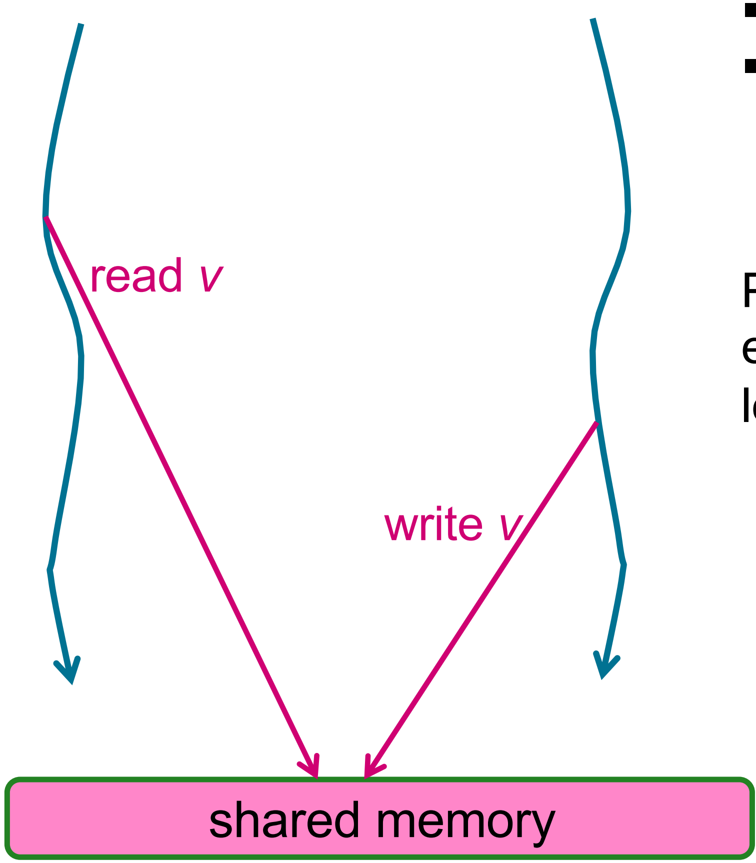
Empty frame

Thus: c.f.x == 0 cannot be verified

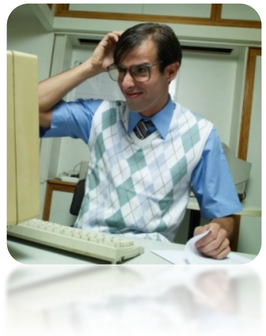
CONCURRENCY: THE NEXT CHALLENGE

Doug Lea

MULTIPLE THREADS CAUSE PROBLEMS

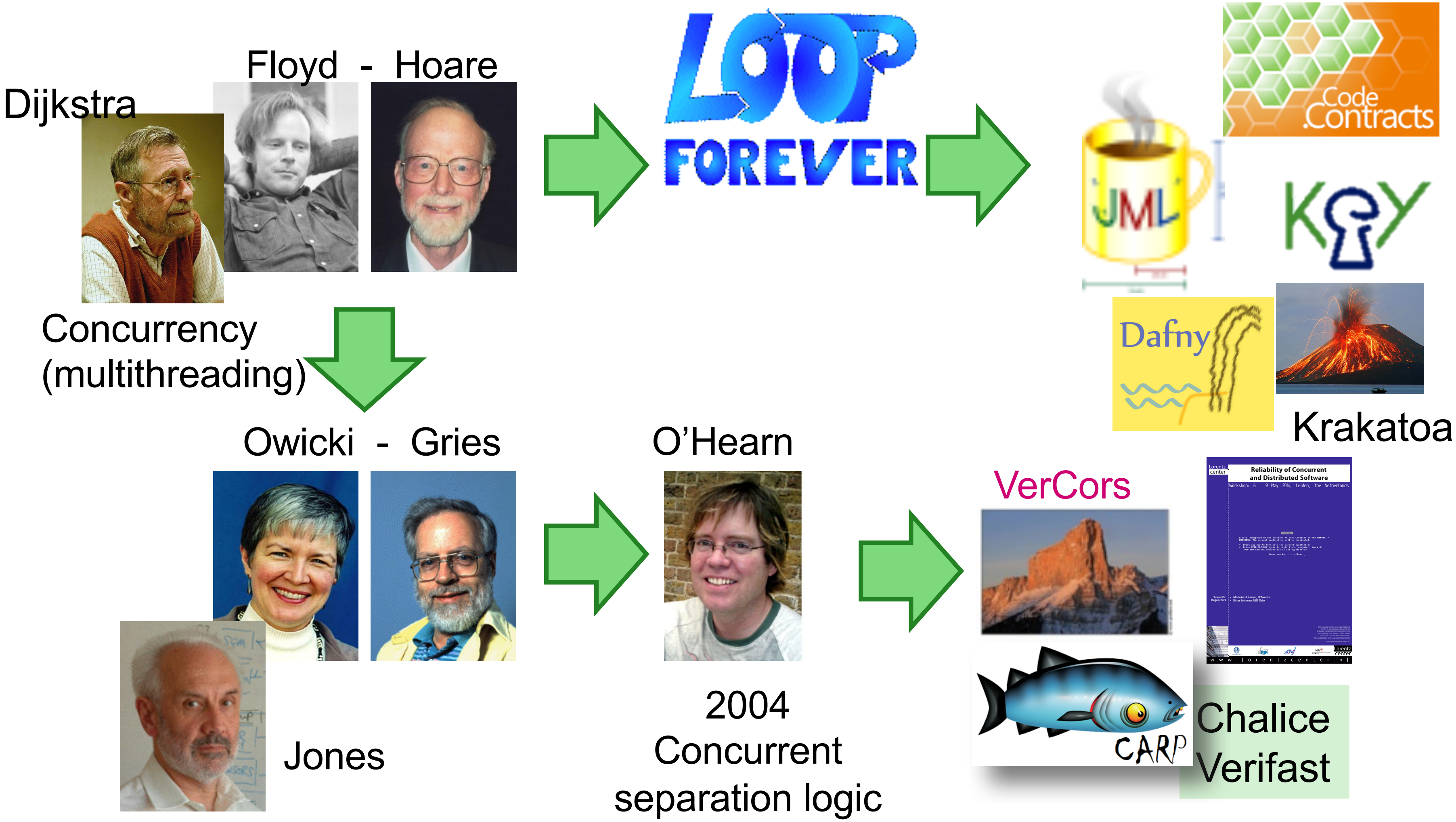


- Order?
- More threads?

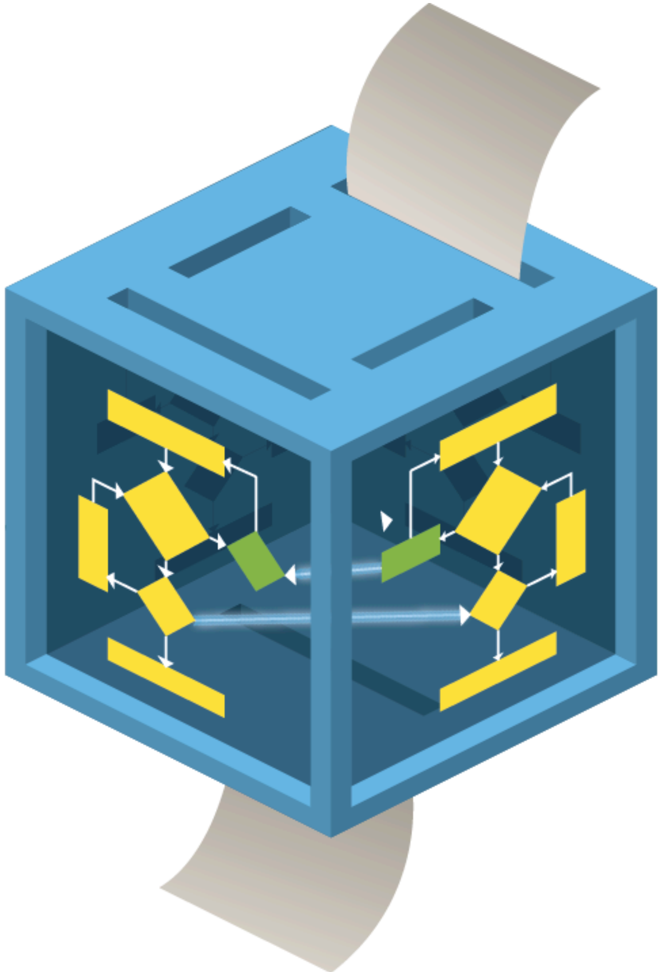


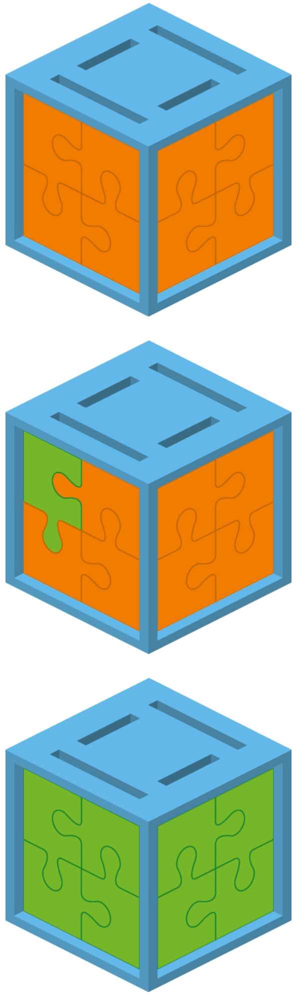
Possible consequences:
errors such as data races caused lethal bugs as in Therac-25

VERIFICATION OF MULTITHREADED PROGRAMS



OUR APPROACH



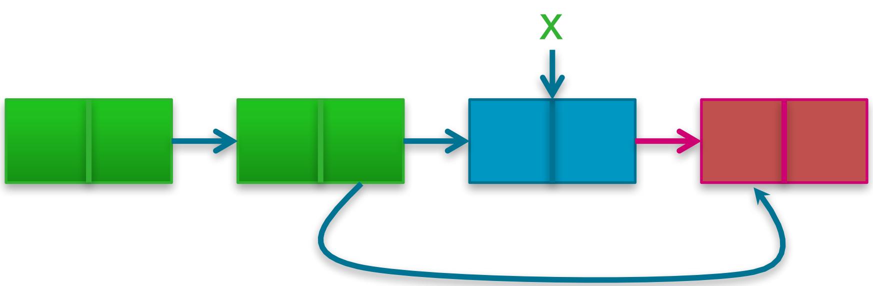


SPECIFICATIONS IN A CONCURRENT SETTING

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

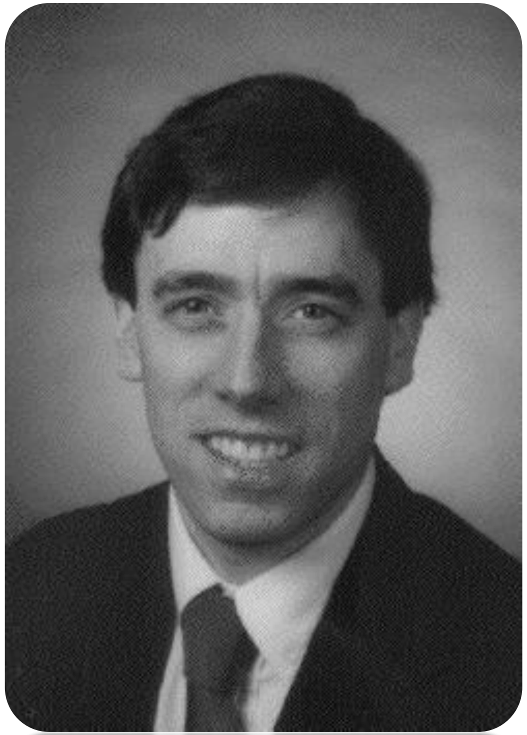
Any other thread might invalidate this!

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



Except when no other thread can update the list

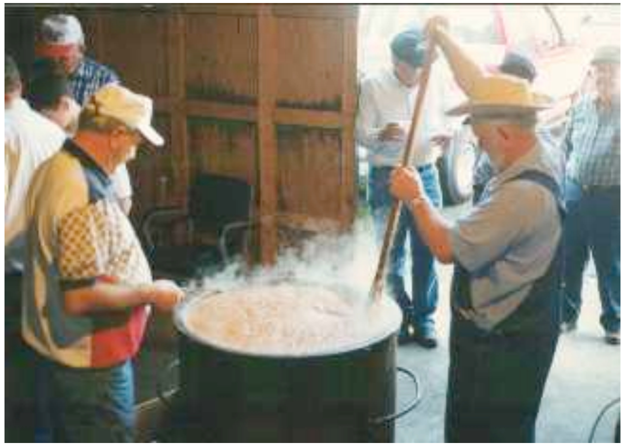
AVOIDING DATA RACES



John Boyland

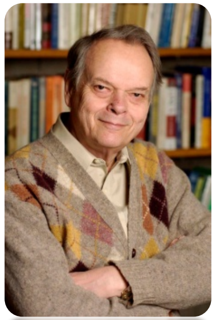
RECIPE FOR REASONING ABOUT JAVA

- Separation logic for sequential Java (Parkinson)
- Concurrent Separation Logic (O'Hearn)
- Permissions (Boyland)



Permission-based Separation Logic for Java

JOHN REYNOLDS'S 70TH BIRTHDAY PRESENT



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

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

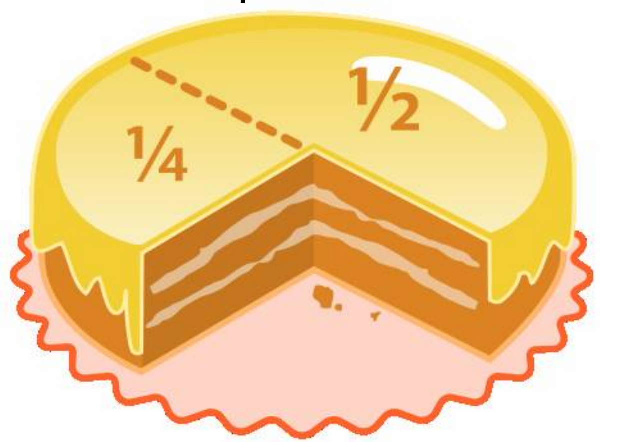
EXAMPLE

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

No interference between the threads

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

⋮ ⋮

$\{\text{PointsTo}(x,1,0) \ast \text{Perm}(n, ½)\}$ $\{\text{PointsTo}(y,1,0) \ast \text{Perm}(n, ½)\}$

x := x + n; x := x + n   y := y + n; y := y + n

$\{\text{PointsTo}(x,1,2*n) \ast \text{Perm}(n, ½)\}$ $\{\text{PointsTo}(y,1,2*n) \ast \text{Perm}(n, ½)\}$

$\{\text{PointsTo}(x,1,0) \ast \text{PointsTo}(y,1,0) \ast \text{Perm}(n,1)\}$

x := x + n; x := x + n || y := y + n; y := y + n

$\{\text{PointsTo}(x,1,2*n) \ast \text{PointsTo}(y,1,2*n) \ast \text{Perm}(n,1)\}\}$

Perm(x,1) = Perm(x, ½) ✳ Perm(x, ½)

Shared variable is only read
No interference between the threads

WHAT MORE IS NEEDED

- Synchronisation between threads:
  - Exclusive access allows writing
  - Shared access only reading allowed
- Reasoning about dynamic thread creation
- Reasoning about thread termination

RULES FOR FORK AND JOIN

- Precondition fork = precondition run
  - Which permissions are transferred from creating to the newly created thread
- Postcondition run = postcondition join
  - Which permissions are released by the terminating thread, and can be reclaimed by another thread
  - Join only terminates when run has terminated
- Specification for run final, it can only be changed by extending definition of predicates preFork and postJoin

EXAMPLE: CLASS FIB

```
class Fib {
int number;

void init(n) {
    this.number := n;
    }

void run() {
    ..
    }
}
```

Leonardo di Pisa/
Fibonacci

FIB'S RUN METHOD

pred preFork = number $\xrightarrow{1}$ _;

group postJoin<perm p> = number $\xrightarrow{p}$ _;

```
requires preFork;
ensures postJoin<1>;
void run() {
    if (! (this.number < 2))
    {   f1 = new Fib; f1.init(number -1);
        f2 = new Fib; f2.init(number - 2);
        fork f1; fork f2; join f1; join f2;
        this.number := f1.number + f2.number }
    else this.number := 1;
}
```

PROOF OUTLINE

pred preFork = number $\xrightarrow{1}$ _;
group postJoin<perm p> = number $\xrightarrow{p}$ _;

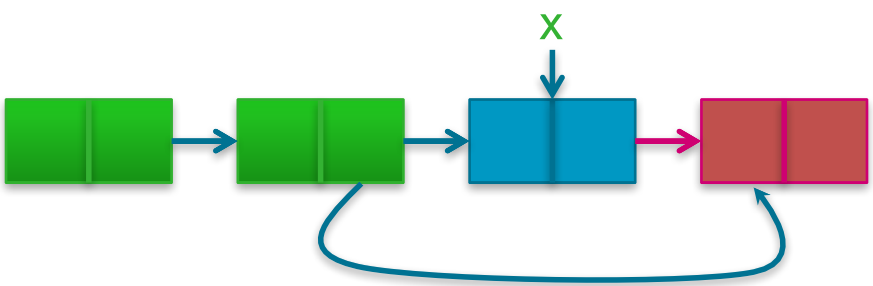
```
requires preFork;
void run() {
    if (! (this.number < 2))
    {   f1 = new Fib; f1.init(number -1); f2 = new Fib; f2.init(number - 2);
        {Perm(f1.number, 1) ∗ Perm(f2.number, 1) ∗ Perm(number, 1)}
        [fold preFork (2x)]
        {f1.preFork ∗ f2.preFork ∗ Perm(number, 1)}
        fork f1;
        {join(f1, 1) ∗ f2.preFork ∗ Perm(number, 1)}
        fork f2;
        {join(f1, 1) ∗ join(f2, 1) ∗ Perm(number, 1)}
        join f1; join f2;
        {f1.postJoin ∗ f2.postJoin ∗ Perm(number, 1)}
        [unfold postJoin (2x)]
        {Perm(f1.number, 1) ∗ Perm(f2.number, 1) ∗ Perm(number, 1)}
        this.number := f1.number + f2.number
        [close postJoin]
        {this.PostJoin}}
    else this.number := 1;
}
ensures postJoin(1);
```

WHAT MORE WOULD WE LIKE TO VERIFY?

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

FUNCTIONAL VERIFICATION OF CONCURRENT PROGRAMS

WORK IN PROGRESS

Marina Zaharieva – Stojanovski

Wytse Oortwijn

EXAMPLE: PARALLEL INCREASE

How to prove:

Ghost code solution:

```
                {x = a + b & a == 0 & b == 0}
{x == a + b & a == 0}      || {x == a + b & b == 0}
<x := x + 1;>              || <x := x + 1;>
<a := 1;> // ghost         || <b :=1;> //ghost
{x == a + b & a == 1}      || {x == a + b & b == 1}
               {x == a + b & a == 1 & b == 1}
                          {x == 2}
```

Problem:

{x == 0}

< x := x + 1;>

{x == 1}

unstable: assertions can be made invalid by other threads

Our approach:
Maintain abstract history of updates

A JAVA-LIKE PROGRAM

```
class Counter{
   int data;
   Lock l;
   resource_inv = exists v. PointsTo(data, 1, v);

requires true;
ensures  true;
void increase(int ){
   l.lock();         // obtain PointsTo(data, 1, v);
      data = data + n;
   l.unlock();       // loose PointsTo(data, 1, v + n);
   // now we don't know anything about data anymore
  }
}
```

Client:

```
   c = new Counter(0);
   fork t1;   //t1 calls c.increase(4);
   fork t2;   //t2 calls c.multiply(4);
   join t1;
   join t2;

   // What is  c.data?
```

Permission to read and update data

Needed:
A specification of increase that records the update

COUNTER SPECIFICATION

```
class Counter{
   int data;
   Lock l;
    //resource_inv = Perm(data, 1);

    //action add(int n) = \old(x) + n;

    requires H;
    ensures H.add(n);
    void increase(int n){
        l.lock();  /* start a */ data = data + n; /* record a */ l.unlock();
    }
}
```

Record LOCAL changes in the history

Similar spec for multiply

COMPUTING THE FINAL VALUE

Global behaviour:

add(4).mul(4) + mul(4).add(4)

Action specifications:

//action **add**(int n) = \old(x) + n;

//action **mul**(int n) = \old(x) * n;

c.data == 4 || c.data == 16

Extensions

- Non-terminating programs
- Predicting behaviour
- Abstracting with larger granularity
- Reasoning about sequences of method calls

Client:

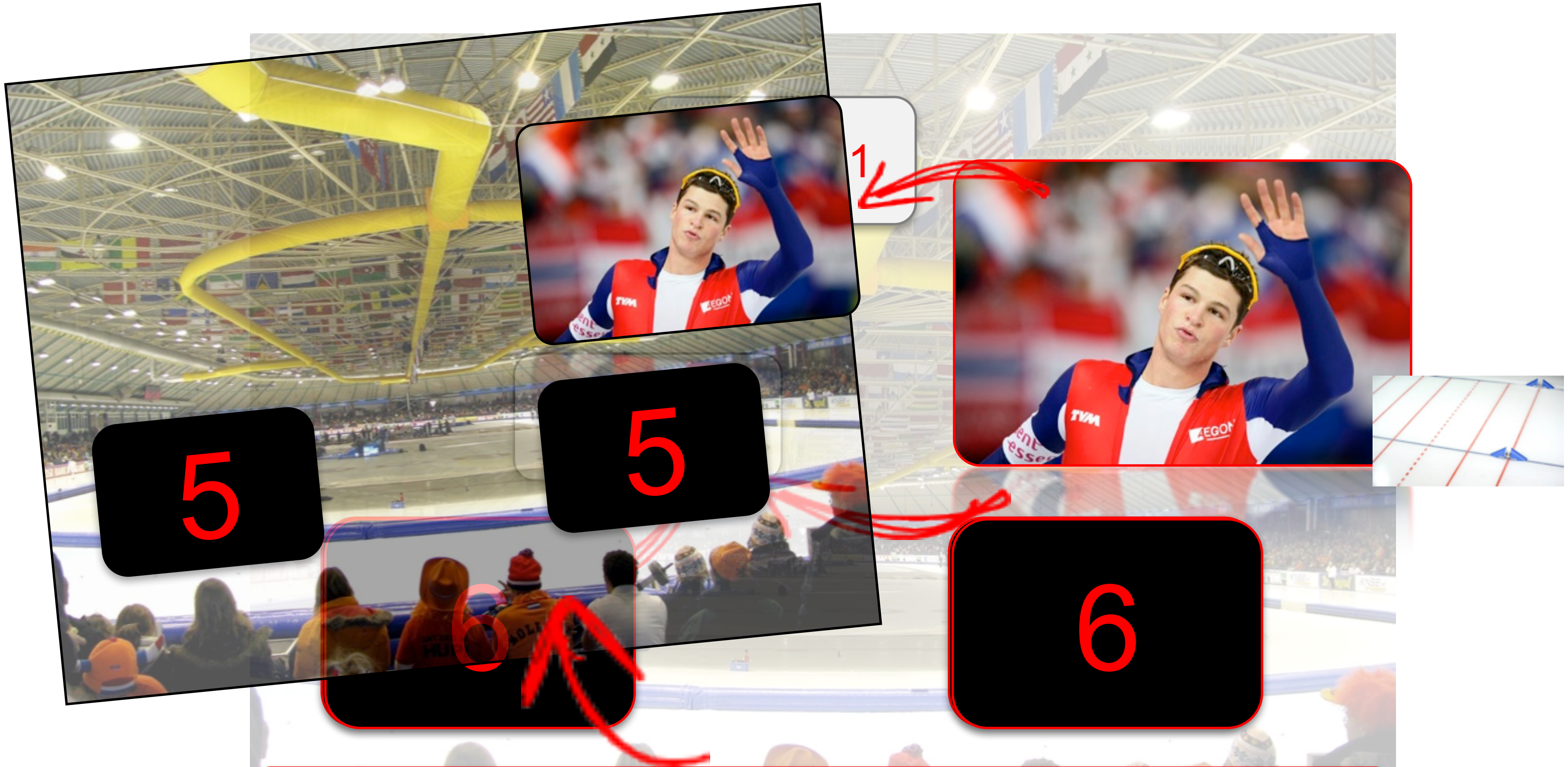
```
c = new Counter(0);
fork t1;   //t1: c.increase(4);
fork t2;   //t2: c.multiply(4);
join t1;
join t2;

// What is  c.data?
```

RUNTIME ASSERTION CHECKING AND CONCURRENCY

ASSERTION INTERFERENCE

ASSERTION INTERFERENCE




```
assert display1.getRounds() == display2.getRounds();
```

THE STROBE FRAMEWORK

- **Speed up assertions**
- Evaluate assertions on separate *checker threads*
- Program continues execution
- Program can change during checks
- Take **snapshot** of the memory
- Evaluate against **snapshot**

Snapshot evaluation:
no assertion interference

Edward E. Aftandilian

ASYNCHRONOUS ASSERTIONS

Implementation

- Independent tasks
- Defined as **futures**
- Will never change the behaviour of the program

SNAPSHOT INTERFACE

Create snapshot

```
int preconditionId = Snapshot.initiateProbe();
```

Execute following statements on snapshot projection

```
currentThread.snapshotId = preconditionId;
```

Execute following statements on live state

```
currentThread.snapshotId = -1;
```

Destroy snapshot

```
Snapshot.completeProbe(preconditionId);
```

USING THE SNAPSHOT INTERFACE

```
public void addNode(Node node) {
    int preconditionId = Snapshot.initiateProbe();
    RVMThread currentThread = RVMThread.getCurrentThread();
    currentThread.snapshotId = preconditionId;
    assert !this.contains(node);
    currentThread.snapshotId = -1;
    Snapshot.completeProbe(preconditionId);
    node.next = this.next;
    this.next = node;
    assert this.contains(node);
}
```

Assertion evaluated over snapshot

No assertion interference

AUTOMATED TRANSLATION WITH SNAPSHOTS

```
/* @ requires !contains(node);
   @ ensures contains(node); @*/
public void addNode(Node node) {
    node.next = this.next;
    this.next = node;
    RVMThread currentThread = RVMThread.getCurrentThread();
    int preId = Snapshot.initiateProbe();
    currentThread.snapshotId = preId;
    assert !contains(node);
    currentThread.snapshotId = -1;
    Snapshot.completeProbe(preId);


    int postId = Snapshot.initiateProbe();
    currentThread.snapshotId = postId;
    assert contains(node);
    currentThread.snapshotId = -1;
    Snapshot.completeProbe(postId);
```

Assertion evaluated in snapshot state

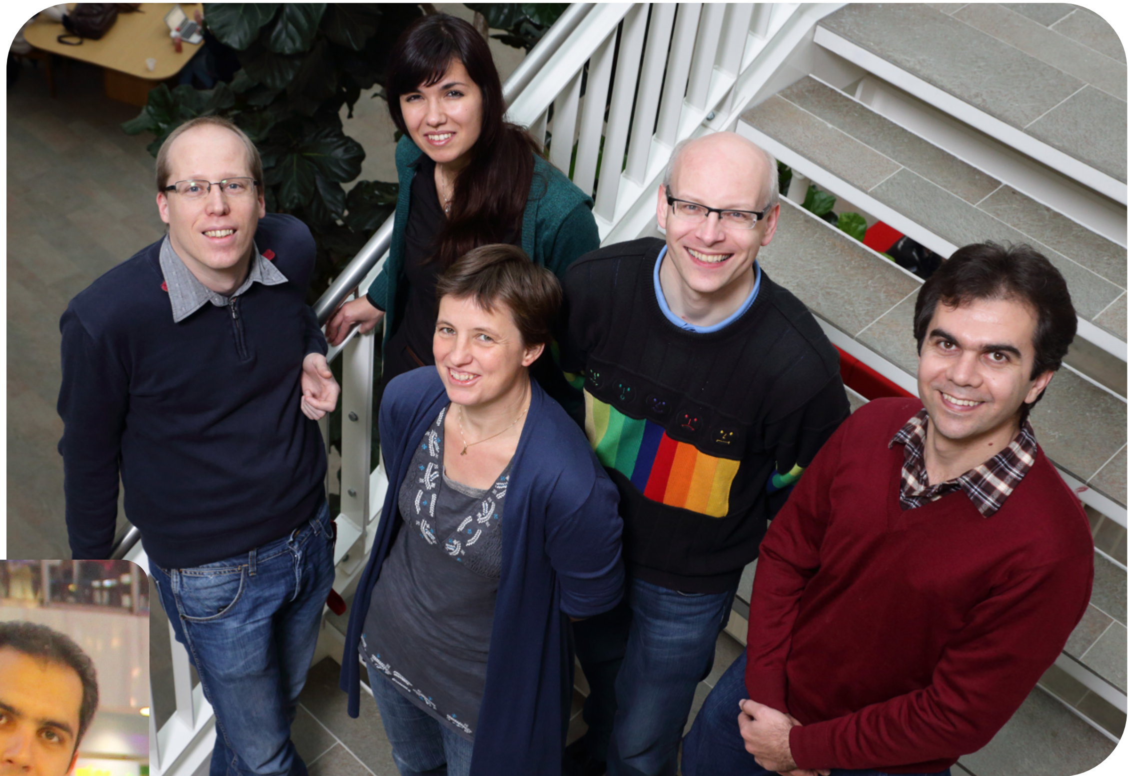
FUTURE WORK

- Static verification
  - Annotation generation
  - Generalise abstraction idea (mixing concrete and abstract specifications)
- Dynamic verification

  After deployment
  - Memory model aware runtime checking
  - Data race detection and fixing

  Before deployment
  - Exercising different executions

ACKNOWLEDGEMENTS



Saeed Darabi, Wojciech Mostowski,
Marina Zaharieva-Stojanovski,
Stefan Blom, Afshin Amighi, Wytse Oortwijn

SUMMARY

- Software quality remains a challenge
- Classical Hoare logic-based techniques are becoming more and more powerful
- Run-time assertion checking powerful extension of standard testing
- Next challenge: verification of concurrent software
  - Separation logic and permissions
  - Verification of functional properties
- Also run-time assertion checking has extra challenges when software is concurrent

> More information? Try Dafny this afternoon!
> Want to try more
> Go to: http://www.utwente.nl/vercors