UNIVERSITY OF TWENTE.

VERIFICATION OF CONCURRENT AND DISTRIBUTED SOFTWARE

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

SEVENTH FRAMEWORK PROGRAMME

erc

European Research Council

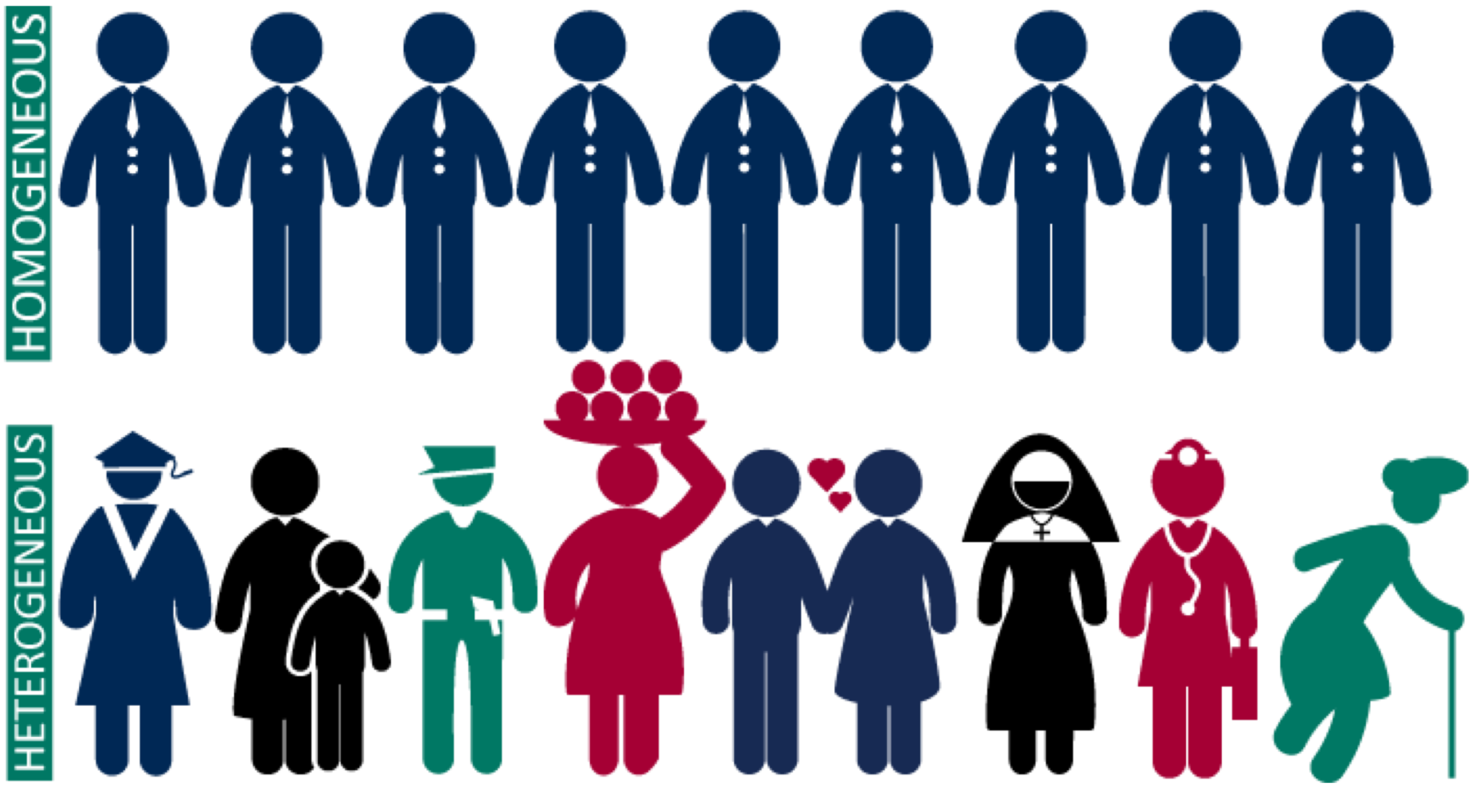
Established by the European Commission

OUTLINE OF THIS LECTURE

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

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

HETEROGENEOUS VS HOMOGENEOUS THREADING



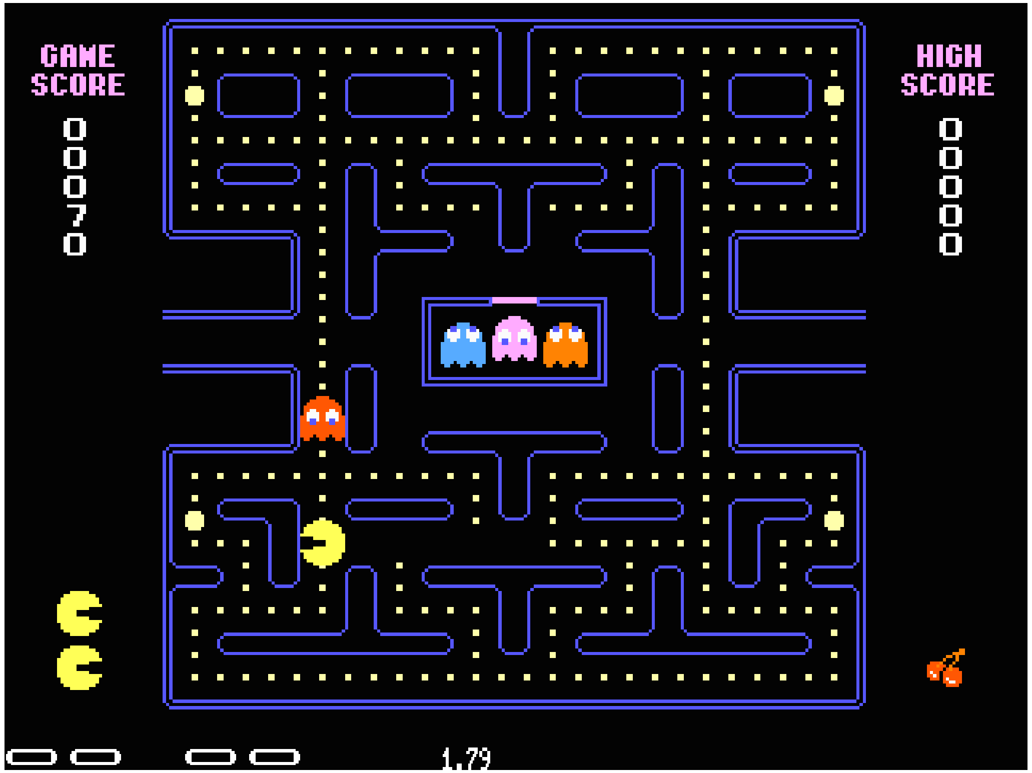
HETEROGENEOUS THREADING

- Every thread executes its own program
- Threads share data
- Efficiency achieved by smart division of tasks

HOMOGENEOUS THREADING

- Each thread executes same sequence of instructions
- Threads share data
- Typically, each thread accesses it own part of the data (otherwise: data race)
- Efficiency achieved by parallel execution of same job

GPU PROGRAMMING



GPU ARCHITECTURE

- Wikipedia:

  A graphics processing unit (GPU),

  also occasionally called visual processing unit (VPU),

  is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display.

- SIMD architecture: built-in support for homogeneous threading
- Also useful for general purpose applications

GPU PROGRAMMING MODELS

- Cuda
  - NVIDIA-only
  - First
  - Widely-used
- OpenCL
  - Platform-independent
  - Can even run on CPU
  - Gaining interest

Essentially: an extended subset of C

EXAMPLE: ADDITION OF TWO VECTORS

Sequential program:

```
void vector_add(int size, float* a, float* b, float* c {
    for(int index = 0; index < size; index++) {
        c[index] = a[index] + b[index];
    }
}
```

VECTOR ADDITION AS OPENCL KERNEL

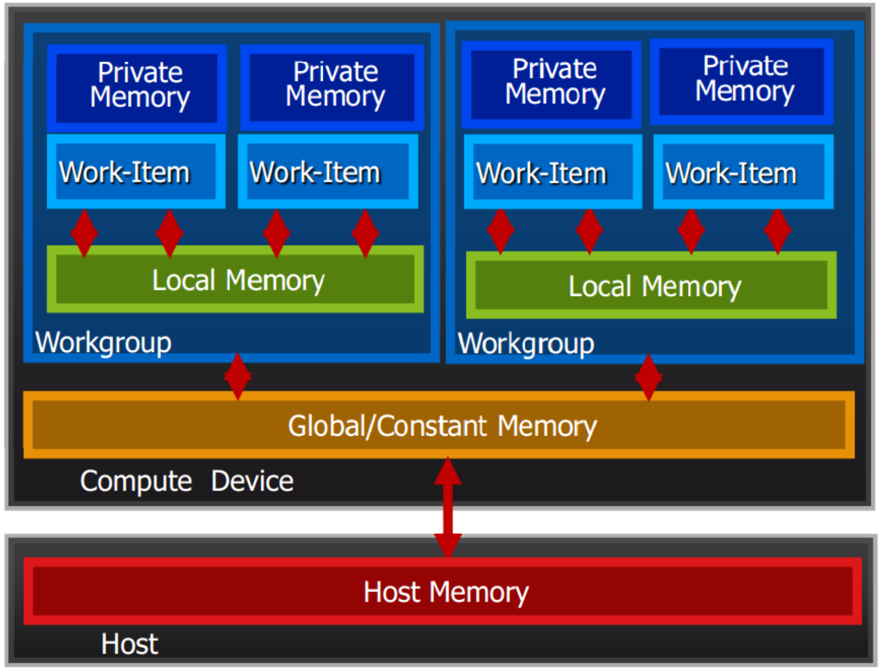
```
__kernel void vectorAdd(__global float* a,
                        __global float* b,
                        __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```

__global
Where are the arrays stored

MULTIPLE MEMORY SCOPES

Beware for terminology:
- Local memory is shared by multiple work items (threads)
- Private memory is private to a single thread

- Per-thread private memory
- Per-workgroup shared memory
  - Low latency
- Global device memory (and constant memory)
  - Slower access
  - Can be accessed by any thread in any workgroup



Explicit copying between local and global memory

HIERARCHY OF CONCURRENT THREADS

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the kernel
- Threads (work items) are grouped into thread blocks (working group)
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

Thread identifiers typically determine which data is accessed

- Each thread has:
  - Local identifier: thread number in thread block
  - Global identifier: thread number in kernel

    Derived from local identifier and working group identifier

REASONING ABOUT KERNEL CODE

```
__kernel bla (__global float* a) {
    int tid = get_global_id(0);
    if (tid > 0) {
        a[tid] = a[tid] + a[tid – 1];
    }
}
```

What will happen here?

Data races should be avoided
Synchronisation needed
Solution: insert a barrier between the two assignments

REASONING ABOUT KERNEL CODE

```
__kernel bla (__global float* a) {
    int tid = get_global_id(0);
    int tmp = a[tid];
    if (tid > 0) {
        tmp = a[tid] + a[tid – 1];
    }
    BARRIER(CLK_GLOBAL_MEM_FENCE);
    a[tid]  = tmp;
}
```

SYNCHRONISATION WITHIN A KERNEL

- Barrier: all threads within a work group block until all threads have reached (the same) barrier
- This is the only moment where you can make an assumption about the state of another thread
- Barriers can be flagged with empty, local, global or local & global
  - Flag indicates which memory is synchronised when all threads reach the barrier

REDUCTION PATTERS AS KERNELS

Could we turn this into a kernel?

```
for (i=0; i<n; i++)
    sum += a[i];
```

We need a way to ensure that each thread does an atomic update

```
__kernel(__global float* a, __global int sum) {
    int tid = get_global_id(0);
    atomic_add(sum,a[tid]);
}
```

A LOGIC FOR GPU KERNELS

- Kernel specification
  - All permissions that a kernel needs for its execution
  - Separated in permissions for
    - Global Memory – given up by host code
    - Shared Memory – local to the GPU
- Thread specification
  - Permissions needed by single thread
  - Should be a subset of kernel permissions
- Barrier specification
  - Each barrier allows redistribution of permissions

Actually:
Group specification in between kernel and thread specification

Plus: functional specifications (pre- and postconditions)

EXAMPLE SPECIFICATION

```
_kernel void bla( _global float* input,
                  _global float* output) {
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
  barrier(CLK_GLOBAL_MEM_FENCE);
  output[(i+1)%wg_size]=
        output[(i+1)%wg_size] * input[i];
}
```

Provided by host

Kernel Specification:

Global Memory Resources:

(forall∗ int i; 0 <= i < output.length; Perm(output[i], 1)

(forall∗ int i; 0 <= i < input.length; Perm(input[i], ½)

Shared Memory Resources: -

Thread Specification:

Precondition:

Perm(output[tid], 1) ∗
Perm(input[tid], ½)

Postcondition:

Perm(output[(tid + 1) % wg_size], 1) ∗
Perm(input[tid], ¼) ∗ Perm(input[(tid + 1) % wg_size], ¼) ∗
output[(tid + 1) % wg_size] = input[tid] ∗ input[(tid + 1) % wg_size]^2

Global proof obligation:
All threads together use no more resources than available in the kernel

EXAMPLE
BARRIER SPECIFICATION

```
_kernel void bla( _global float* input,
                              _global float* output) {
   int i = get_global_id(0);
   output[i] = input[i] * input[i];
   barrier(CLK_GLOBAL_MEM_FENCE);
   output[(i+1)%wg_size]=
          output[(i+1)%wg_size] * input[i];
}
```

Barrier Specification:

Precondition:

Perm(output[tid], 1) ✳ Perm(input[tid], ½) ✳
output[tid] = input[tid] ✳ input[tid]

Postcondition:

Perm(output[(tid + 1) % wg_size], 1) ✳
Perm(input[tid], ¼) ✳ Perm(input[(tid + 1) % wg_size], ¼) ✳
output[(tid + 1)% wg_size] = input[(tid + 1)% wg_size]^2

Global proof obligation:
All permissions available in kernel

Global proof obligation:
Barriers correctly transfer knowledge about state

Extra layer:
workinggroup specifications

PROOF OBLIGATIONS

- Threads respect their thread specification
- Kernel resources are sufficient to provide each thread necessary global resources
- Local resources are properly distributed over threads
- Kernel precondition implies universal quantification of thread precondition
- Barriers only redistribute permissions that are in the kernel
- Universal quantification of barrier precondition implies universal quantification of barrier postcondition
- Universal quantification of thread postcondition implies kernel postcondition

PARALLEL BLOCK: VERCORS ENCODING

- OpenCL kernels encoded as sequences of parallel blocks

```
par (int tid = 0 .. A.length)
{
    A[tid] = 0;
}
```

- Each parallel block ends with an implicit barrier
- For more complicated patterns, also explicit barrier statement
- Each iteration in parallel block should be specified by pre- and postcondition
- Atomic operations:

```
atomic(inv) { critical section code }
```

where inv is the resource invariant that gives access to this code

EXERCISES

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

TOWARDS THE FULL POWER OF VERIFICATION

- Abstract predicates: encapsulate state
- Ghost variables: verification-only state
- Abstract models to reason about functional behavior of concurrent programs

ABSTRACT PREDICATES

Matthew Parkinson

SPECIFYING DATA STRUCTURES

- Abstract predicates represent and encapsulate state, with appropriate operations
- Abstract predicates are scoped
  - Code verified in scope can use name and body
  - Code verified out of scope can only use name
- Explicit open/close axiom to open definition of abstract predicate, provided it is in scope

$$\alpha(x1, .., xn) = P \text{ in scope } \vdash \alpha(e1, .., en) \Leftrightarrow P[x1 := e1, .. xn := en]$$

ABSTRACT PREDICATES ON LIST

```
class Node {
        int val;
        Node next;
}
```

- Predicate list
  - pred list (i)= (i = null) $\vee$ $\exists$ Node j, int a. i.val $\rightarrow$ a $\ast$ i.next $\rightarrow$ j $\ast$ list j

  recognises list structure
- Predicate list:
  - pred list (ϵ, i) = (i = null)
  - pred list ((a.α), i) = $\exists$Node j. i.val $\rightarrow$ a $\ast$ i.next $\rightarrow$ j $\ast$ list α j

  relates list content with abstract list value
- Operations like append and reverse in specifications can be defined on abstract type

LIST PREDICATE IN VERCORS

```
class Node {

  int value;
  Node next;

  resource list(frac p) =
     p != none ** Perm(value, p) ** Perm(next, p) **
     (next != null ==> next.list(p));

}
```

GHOST VARIABLES

- Sometimes verification requires to maintain extra state: ghost state
- Examples:
  - Keep track of original variables
  - Keep track how variables evolve
  - Compute additional properties over state

- VerCors approach:
  - given T v : pass extra ghost parameter v of type T to method
  - yields T v: method returns an extra ghost return value
  - m() with {givenvar = E} then {z = yieldsvar}

EXAMPLE GHOST TRACE

```
function up-sweep(int H, int N, seq<int[]> Tree, seq<int> Input, int tid) {
    int lvl=1;
    while (lvl ≤ H) {
        if (lvl < N/2) {
            a[tid] = a[2 * tid] + a[2 * tid + 1];
            Tree[lvl][tid] = Tree[lvl - 1][2 * tid] + Tree[lvl - 1][2 * tid + 1];
            barrier(tid);
            lvl = lvl + 1;
        }
    }
}
```

FUNCTIONAL VERIFICATION OF CONCURRENT PROGRAMS

Wytse Oortwijn

Marina Zaharieva – Stojanovski

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}
```

Our approach:
Maintain abstract model of updates

unstable: assertions can be made invalid by other threads

AS A JAVA-LIKE PROGRAM

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

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

Client:

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

// Is  c.x  == 2 ?
```

Permission to read and update x

Needed:
A specification of increase that indicates what behavior of increase is

A HISTORY OF ACTIONS

Abstract model is process algebra term composed of user-defined actions (use ACP)

Examples

```
action a<int x>(int k) = \old(x) + k;

action b<list l>(int e) = cons(\old(l), e);

action c<int k>(int w) = w;
```

COUNTER SPECIFICATION

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

    //action a<int x> () = \old(x) + 1;

requires Model(x, p, M.a);
ensures Model(x, p, M);
void increase(){
    l.lock();  /* start a */ x++; /* record a */ l.unlock();
    }
}
```

Record LOCAL changes in the history

EXAMPLE: OWICKI GRIES

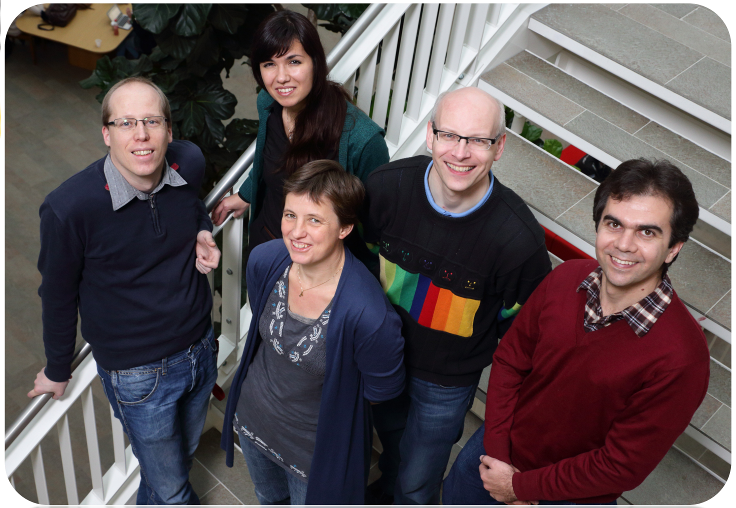
```
class Future {
    int x;

    modifies x;
    ensures x == \old(x) + 2;
    process incr();

    modifies x;
    ensures x == \old(x) + 4;
    process OG() = incr() || incr();
}
```

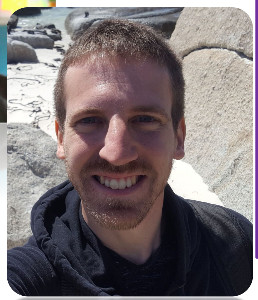
OWICKI GRIES PROGRAM

```
ensures \result == x + 4;
int main(int x) {
    model.x = x;
    invariant inv(HPerm(model.x, 1)) //;
    {
        par Thread1()
        { atomic (inv) { model.x = model.x + 2; } }
        and Thread2()
        { atomic (inv) { model.x = model.x + 2; } }
    return model.x;
}
```

For the annotations, we go to my editor

ACKNOWLEDGEMENTS





Saeed Darabi, Wojciech Mostowski,
Marina Zaharieva-Stojanovski,
Stefan Blom, Afshin Amighi, Wytse Oortwijn,
Sebastiaan Joosten, Mohsen Safari, Fauzia Ehsan,
Raul Monti, Henk Mulder, Pieter Bos, Jelte Zeilstra

EXERCISES

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