UNIVERSITY OF TWENTE.

VERIFICATION OF CONCURRENT DATA STRUCTURES

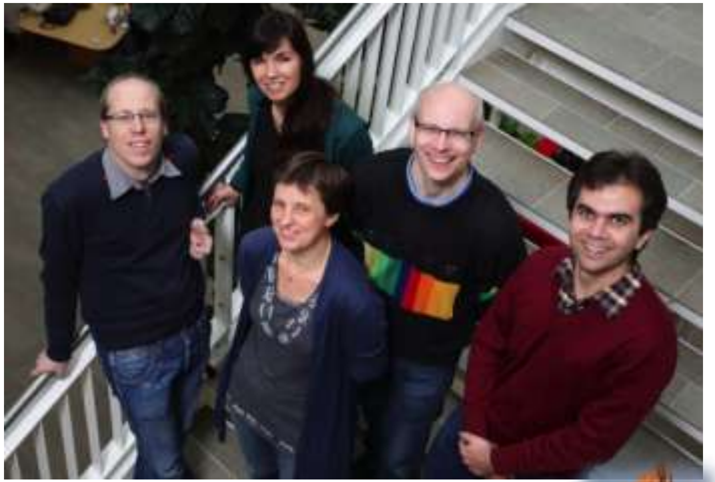
MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

CREDITS

Joint work with

- Christian Haack (aICAS, Germany) and
- Clément Hurlin (now at Prove & Run, France)
- Afshin Amighi
- Stefan Blom
- Matej Mihelcic
- Wojciech Mostowski
- Marina Zaharieva-Stojanovski



The VerCors team

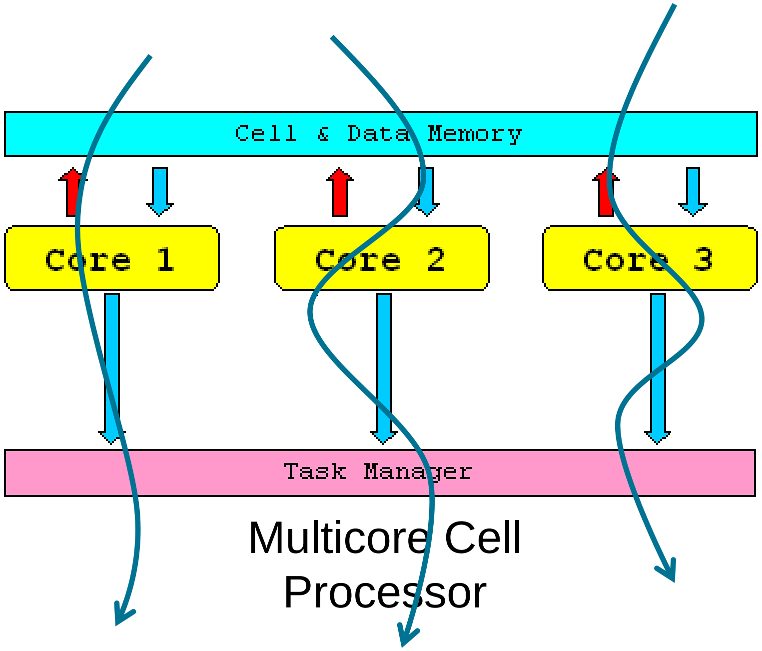
OVERVIEW

- Motivation: why reasoning about multithreaded programs
- Permission-based separation logic for Java
- First results of the VerCors project
  - Reasoning about different synchronisation mechanisms
  - Verification of lock-free algorithms
  - VerCors tool

THE FUTURE OF COMPUTING IS MULTICORE

Single core processors:
The end of Moore's law

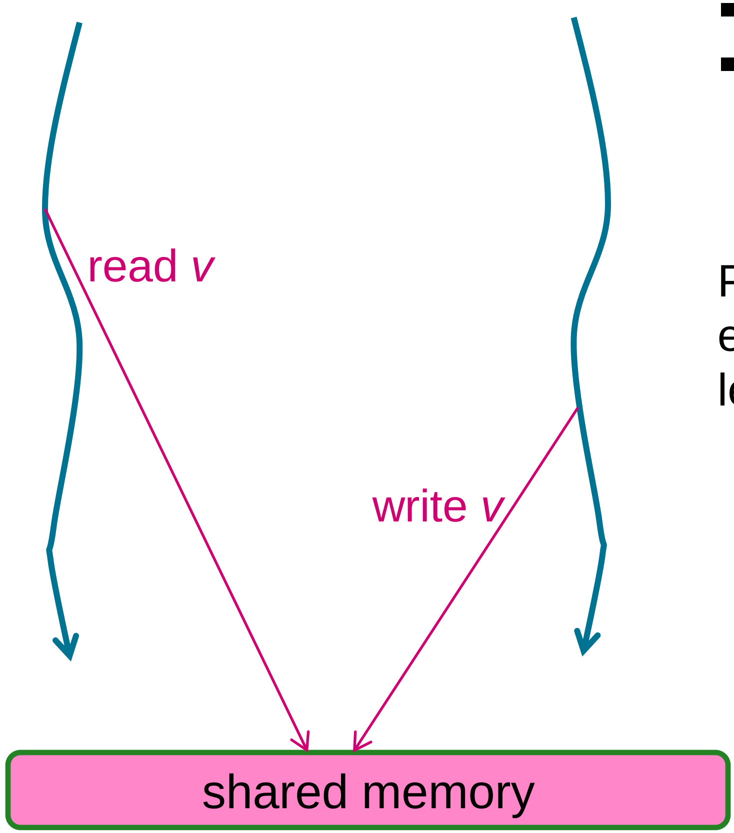
Solution:
Multi-core processors



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

PROGRAM LOGICS CHASE BUGS

Floyd - Hoare

LOOP FOREVER

KeY

ESC/Java2
Spec#

JML

concurrency

Owicki - Gries

Jones

O'Hearn

2007
separation logic

with Haack &
Hurlin

VerCors

PERMISSION-BASED SEPARATION LOGIC FOR JAVA

Challenges

- Reasoning about multi-threaded Java programs
- Allow multiple simultaneous reads
- Dynamic thread creation
- Reentrant locks

Two ingredients:

- Separation logic
- Permissions

BACKGROUND: 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)
- Locations on heap can be aliased

- Store (or call stack) contains data related to method call (parameters, local variables)

- Main idea: assertions about state can be decomposed into assertions about disjoint substates

SEPARATION LOGIC

John Reynolds – Peter O'Hearn

Syntactic extension of predicate logic to reason about the heap:

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

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 of 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

ADVANTAGES OF SEPARATION LOGIC

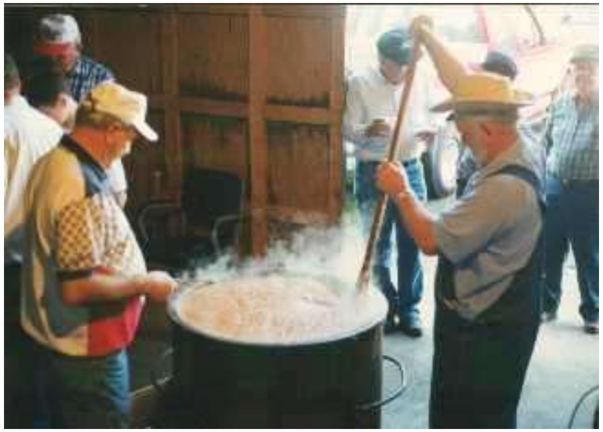
- Reasoning about programs with pointers
- Two interpretations $e.f \rightarrow v$
  - Field $e.f$ contains value v
  - Permission to access field $e.f$

    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 aren't aliases
- Local reasoning
  - only reason about heap that is actually accessed by code fragment
  - rest of heap is implicitly unaffected: frame rule

RECIPE FOR REASONING ABOUT JAVA

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



Permission-based Separation Logic for Java

PUT INGREDIENT 1 IN A BOWL:
SEPARATION LOGIC FOR JAVA

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

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

where X and Y are logical variables

Matthew
Parkinson

ADD INGREDIENT 2, AND STIR WELL:
JOHN REYNOLDS'S 70TH BIRTHDAY PRESENT

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

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

COOK FOR A WHILE:
WHY IS THIS NOT SUFFICIENT?

- Number of parallel threads is fixed
- Simultaneous reads not allowed
- Side-condition: predicates must be precise

ADD LAST INGREDIENT: PERMISSIONS

John Boyland

- 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
- Thus: simultaneous reads allowed, but no read-write or write-write conflicts (data races)

SERVE WHILE HOT:
PERMISSION-BASED SEPARATION LOGIC FOR JAVA

Syntactic extension of predicate logic:

$\varphi ::= e.f \xrightarrow{\pi} e' \mid \varphi \ast \varphi \mid \varphi -\!\!\ast\ \varphi \mid \ldots$

Meaning:

- $e.f \xrightarrow{\pi} e'$ – heap contains location pointed to by *e.f*, containing the value given by the meaning of *e'* and thread has access right π on *e.f*
- $\varphi 1 \ast \varphi 2$ – 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

PERMISSION TRANSFER

- Permissions are transferred between threads upon
  - Thread creation (fork) [actually start in Java]
  - Thread termination (via join)
  - Ensured by method contracts for the thread's run method
- Locks have permissions associated with them
  - Permissions obtained upon initial acquire
  - Permissions given back upon final release
  - Requires reasoning about reentrant locks

EXAMPLE: THREAD CREATION AND JOIN

Abstract predicates (with inheritance)

class Fib extends Thread { int number;
pred preFork = number $\xrightarrow{1}$ _;
pred postJoin<perm p> = number $\xrightarrow{p}$ _;

Standard specification

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

Permission to access f1.number and f2.number transferred to f1 resp. f2

Permission to access f1.number and f2.number transferred back to this thread

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 abstract predicate

  pred inv = true;
- Lock object extends inv
- In rules: if inv is resource invariant of *x*

  {true} lock *x* {*x*.inv}

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

WHY IS LOCK REENTRANCY USEFUL

- Method implementor does not have to make assumptions about locking being (not) held
- In particular useful for libraries

- We need to reason about reentrant locks

EXTRA PREDICATES

For lock initialisation

- Add extra predicates to logic
- $\varphi ::= e.f \overset{\pi}{\rightarrow} e' \mid \varphi \ast \varphi \mid \varphi -\!\ast\ \varphi \mid$

  Lockset(S) | S contains e | e.fresh | e.initialized
- Lockset (S) - S is the multiset of locks held by current thread
- S contains e - multiset S contains e
- e.fresh - e's resource invariant not yet initialized
- e.initialized - e's resource invariant initialized

RULES FOR LOCKING AND UNLOCKING

$$\frac{}{\{\text{Lockset}(S) \ast \neg(S \text{ contains } u) \ast u.\text{initialized}\}\ \text{lock } u\ \{\text{Lockset}(u.S) \ast u.\text{inv}\}}$$

$$\frac{}{\{\text{Lockset}(u.S)\}\text{lock } u\{\text{Lockset}(u.u.S)\}}$$

$$\frac{}{\{\text{Lockset}(u.S) \ast u.\text{inv}\}\text{unlock } u\{\text{Lockset}(S)\}}$$

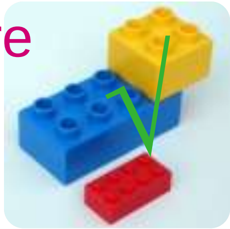
$$\frac{}{\{\text{Lockset}(u.u.S)\}\text{unlock } u\{\text{Lockset}(u.S)\}}$$

Drawback: reasoning about aliases back in the logic

erc
European Research Council
Established by the European Commission

GOALS OF VERCORS PROJECT

Automated verification of concurrent software



- Collection of verified concurrent data structures
- Generic verification theory of concurrent programming
  - Different concurrency and synchronisation techniques
  - Effects of changes to locking policy
  - Different programming languages
  - Distributed setting

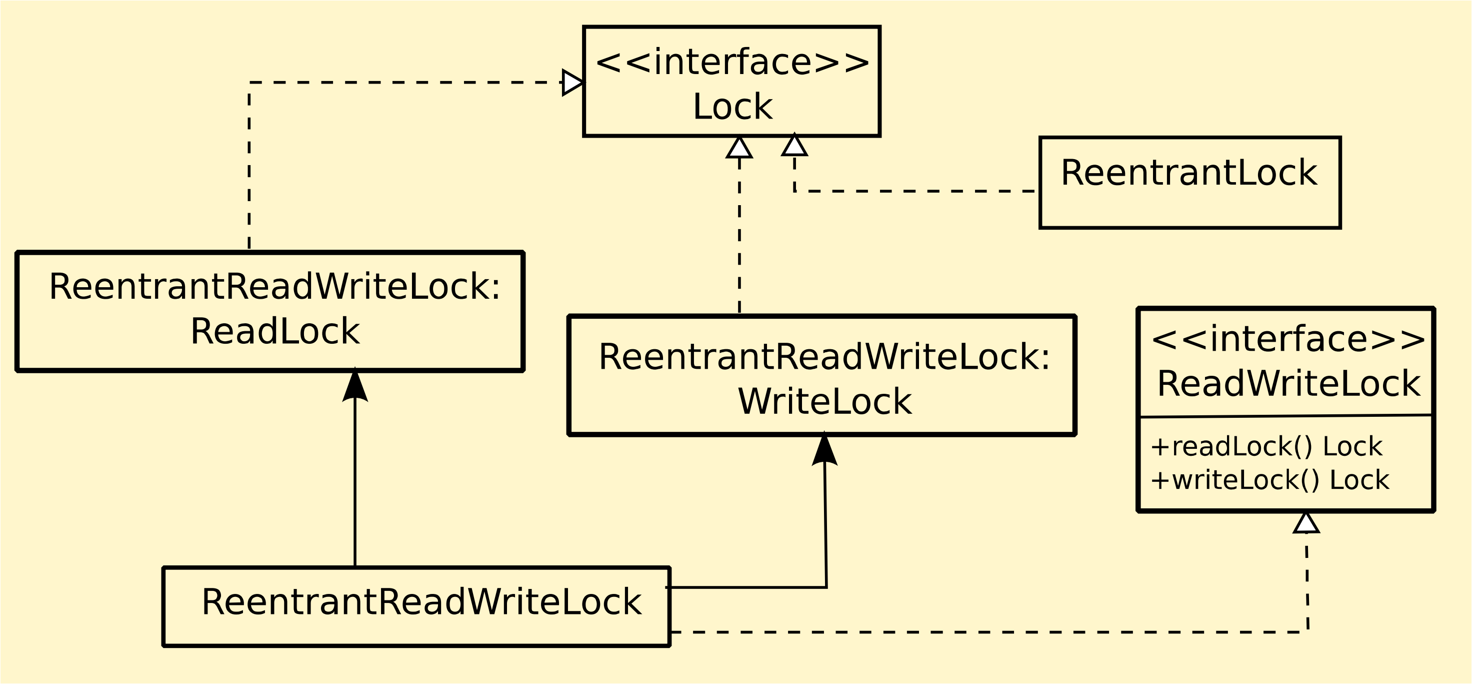
Java

- Automation
  - Tool support
  - Decision procedures for proof obligations
  - Generation of specifications

SYNCHRONISATION MECHANISMS

- Reasoning about synchronisation primitives typically built-in to the logic
- Every primitive: new rules needed
- Instead: lift this to the specification level
- Specification of synchronisation classes in Java API

LOCK HIERARCHY



SPECIFICATION OF LOCK INTERFACE

- Reentrancy not necessary for classes implementing Lock interface
- Access does not have to be exclusive
  - ReadLock
- Specifications of methods in lock interface generalise lock/unlock rules

```
public interface Lock /*@<pr : frac −> pred>@*/ {
    //@ pred inv<frac p> = pr<p>;
    //@ pred share<frac p>;
    //@ protected model instance boolean isExclusive;
    //@ protected model instance boolean isReentrant;
    …
}
```

SPECIFICATION OF METHOD LOCK

```
/*@  public normal_behavior
       requires isExclusive ∗ LockSet(S) ∗ !(S contains this) ∗ initialized;
       ensures   LockSet( this . S)∗  inv<1>;
     also
     public normal_behavior
       requires  ! isExclusive ∗ LockSet(S)  ∗  !(S contains this ) ∗
                 initialized ∗ share<q>;
       ensures LockSet(this . S) ∗ inv<q>;
     also
     public normal_behavior
       requires  isReentrant ∗  LockSet( this . S);
       ensures   LockSet( this . this . S);   @∗/
void lock();
```

SPECIFICATION FOR REENTRANT LOCK

This is all:

```
//@ private represents isReentrant <− true;
//@ private represents isExclusive <− true;
```

All other specifications directly inherited

Simplified reference implementation of reentrant lock verified

MORE SYNCHRONISATION PRIMITIVES

- ReadWriteReentrantLock
  - Contains ReadLock and WriteLock
  - Both implement Lock interface
  - Inherit specifications with appropriate instantiations
- Other Synchronisation Mechanisms
  - Semaphore
  - CountDownLatch
  - Essentially same approach
    - Resource invariant
    - Difference only in how/when the permissions are transferred

LOCK-FREE HASHTABLE

- Developed by Laarman, van de Pol and Weber for LTSmin tool set
- A shared state storage
- Used for efficient multi-core state space exploration

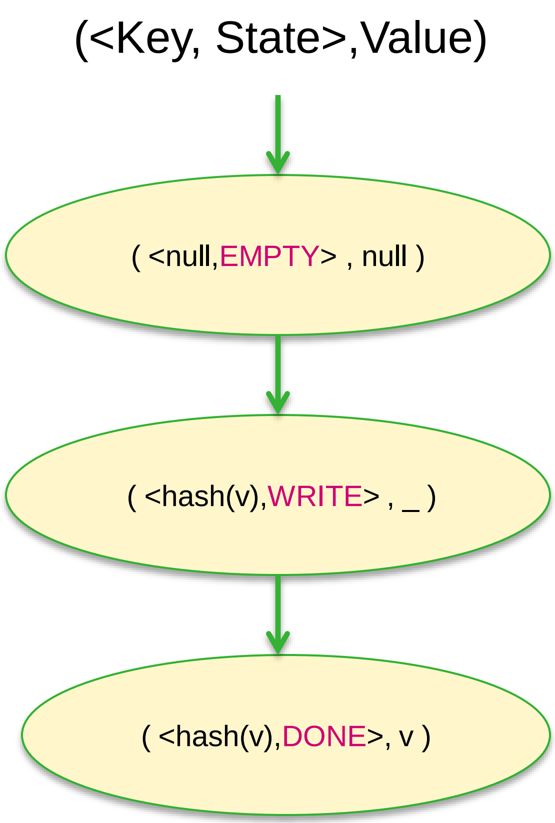
```
T:={S0};  V:={ };
while state:=T.get() do{
   count:=0;
   for succ in next_state(state) do{
      count:=count+1;
      if V.find_or_put(succ) then
            T.put(succ);
      }
}
```

HASHTABLE DESIGN

if not found
then
put;
return false;

if found
then
return true;

```
find_or_put(v){
  h := hash(v);
  …
  if Bucket[i] = <null,EMPTY> then {
      if CAS(Bucket[i], <null,EMPTY>, <h,WRITE>)
      then{
            Data[i] := v;
            Bucket[i] := <h,DONE>;
            return false;
      }
  }
  if Bucket[i] = <h,_> then {
       while Bucket[i] = <_,WRITE> do … wait …
done;
            if Data[i] = v then
                 return true;
  }
  …
}
```



Verified a simplified version: SingleCell

SINGLE CELL CODE

```
public int find_or_put(int d){
    boolean b = state.cas(E,W);
    if(b){ data = d; state.set(D); return 0; }    // value stored
    if (!b) {
        while(state.get()==W) {};
        if (state.get()==D){
            if (data == d) { return 1;} // value already stored
            return -2; // collision
        }
    }
    return -1; //error
}
```

State: AtomicInteger

- get
- set
- cas

OUR ATOMIC INTEGER SPECIFICATION

Immutable:
a value is stored in data and it cannot be change anymore

```
    private int data;

/*@  public predicate boolean assigned(role r, int val) =
          (r == AUTH && val == E ==> Perm(data,1)) *
          (r == AUTH && val == D ==> Immutable(data)) *
          (r == T && val == D ==> Immutable(data));
 */

    …
```

OUR ATOMIC INTEGER SPECIFICATION

```
/*@  requires assigned(role,last) * assigned(AUTH,v);
     ensures assigned(role,v); */
     void set(int v);

//@  requires assigned(role,last);
     ensures assigned(role,\result); */
     int get();

/*@  requires  assigned(role,last) * assigned(AUTH,n);
     ensures   \result ==> assigned(role,n) * assigned(AUTH,x);
     ensures   !\result ==> assigned(role,last) * assigned(AUTH,n); */
     boolean cas(int x, int n);
```

WHAT WE CAN PROVE

Cell will be immutable:

```
/*@
 ensures \result == 0 ==> assigned(T,D) && data == d;
 ensures \result == 1 ==> assigned(T,D) && data == d;
 // otherwise error or collision
 */
public int find_or_put(int d){
    …
}
```

VERCORS TOOL

- Support for different programming languages
  - Java
  - Teaching language PVL
- Specification language
  - Separation logic operators
  - Fractional access permissions
  - Abstract predicates
  - Commonly used part of JML
- Leverage existing verification tools
  - Boogie
  - Chalice
  - VeriFast?

VERIFICATION OF CONCURRENT DATA STRUCTURES

- Permission-based separation logic
- Important to cover all relevant aspects
  - Generic specifications with appropriate instantiations
- Functional property specifications
  - History-based approach
  - Stability of specifications
  - Theory of strong invariants and constraints
- Atomic references used to protect data
  - Different patterns
- Tool support
- Extension to reason about GPU programs

TO BE CONTINUED...

Automated verification of concurrent software

http://fmt.ewi.utwente.nl/research/projects/VerCors

VERCORS TOOL ARCHITECTURE