UNIVERSITY OF TWENTE.

VerCors

THE VERCORS VERIFIER: A VERIFIER FOR MULTIPLE CONCURRENT PROGRAMMING LANGUAGES

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

SAVES

CHEOPS

VERIFIED CONSTRUCTION OF CORRECT AND OPTIMISED PARALLEL SOFTWARE

Mercedes

MAXIMAL RELIABILITY OF CONCURRENT AND DISTRIBUTED SOFTWARE

Pallas

SOFTWARE IS EVERYWHERE

All software has errors!

Software failures can have enormous impact

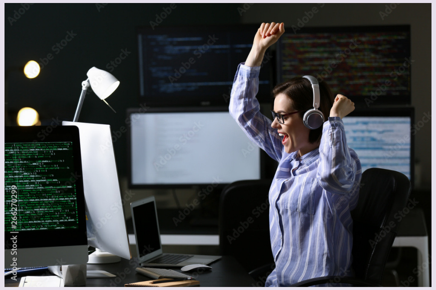
Dutch emergency line hit by KPN telecoms outage

25 June 2019

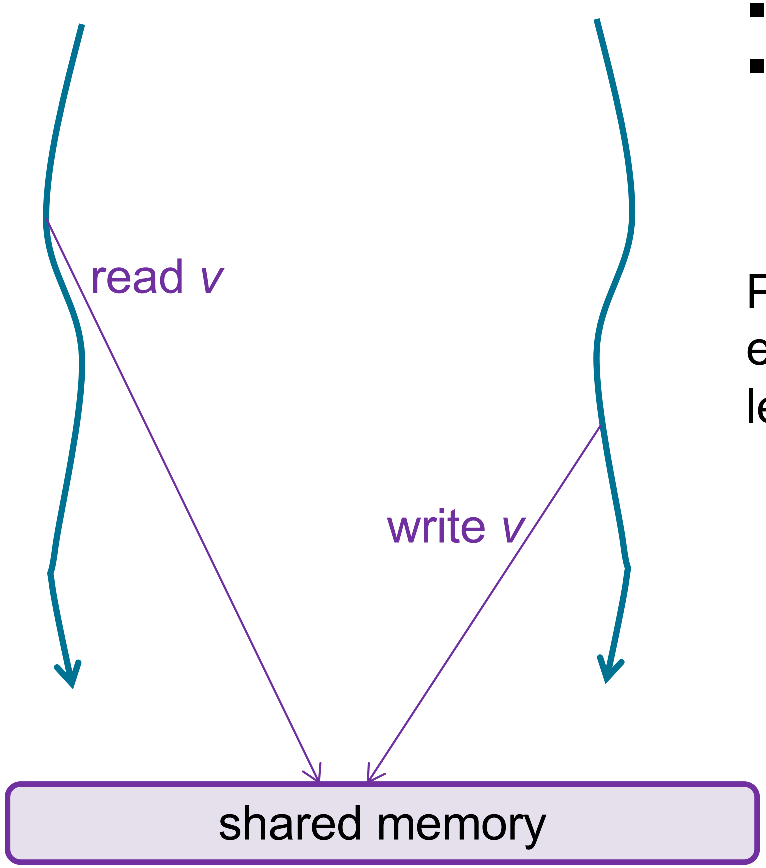
The disruption originated from the network of national carrier KPN

The Netherlands has been hit by its largest telecommunications outage in years, with the 112 emergency number knocked out across the country.

How can we avoid software failures in an effective way?



CONCURRENT SOFTWARE CHALLENGES



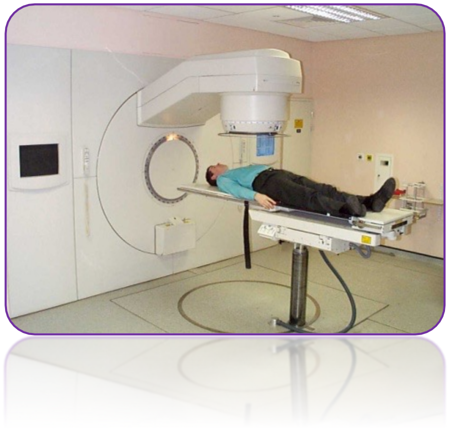
read *v*

write *v*

shared memory

- Order?
- More threads?

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

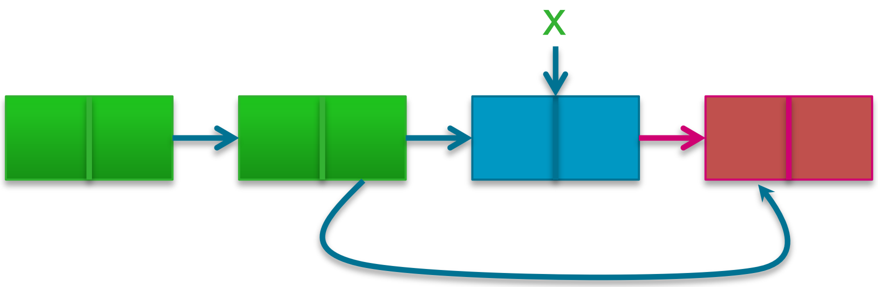


CONCURRENT SOFTWARE: FUNCTIONAL BEHAVIOUR

```
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

THIS TALK

- VerCors: verification of concurrent software
    - Overview
    - Examples
    - Annotation-aware optimisations for GPU programs
    - Verification of SystemC designs
    - Verification of LLVM programs
- Future ideas and plans

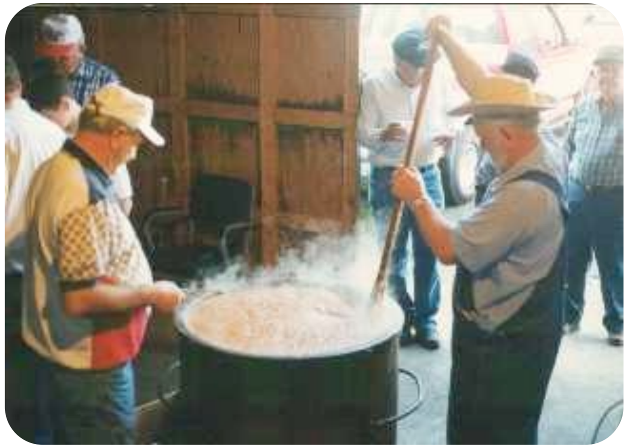
VerCors

VERCORS VERIFIER: VERIFICATION OF CONCURRENT SOFTWARE

Permission-based Separation Logic

- Separation logic for sequential Java
- Concurrent Separation Logic (with variations/extensions)
- Permissions
- JML specifications
- Dynamic frames
- ...

Separation logic developed to reason about programs with pointers



PERMISSION-BASED SEPARATION LOGIC

Assertions: extension of predicate logic:

$\varphi ::= \mathrm{Perm}(x, \pi) \mid \varphi \ast \varphi \mid \ldots$

- $\mathrm{Perm}(x, \pi)$ – thread has permission π to access field x on heap

All formulas should be properly framed, i.e. you can only reason about heap locations that you have access to

- $\varphi 1 \ast \varphi 2$ – heap can be split in disjoint parts, satisfying $\varphi 1$ and $\varphi 2$

Supports local reasoning

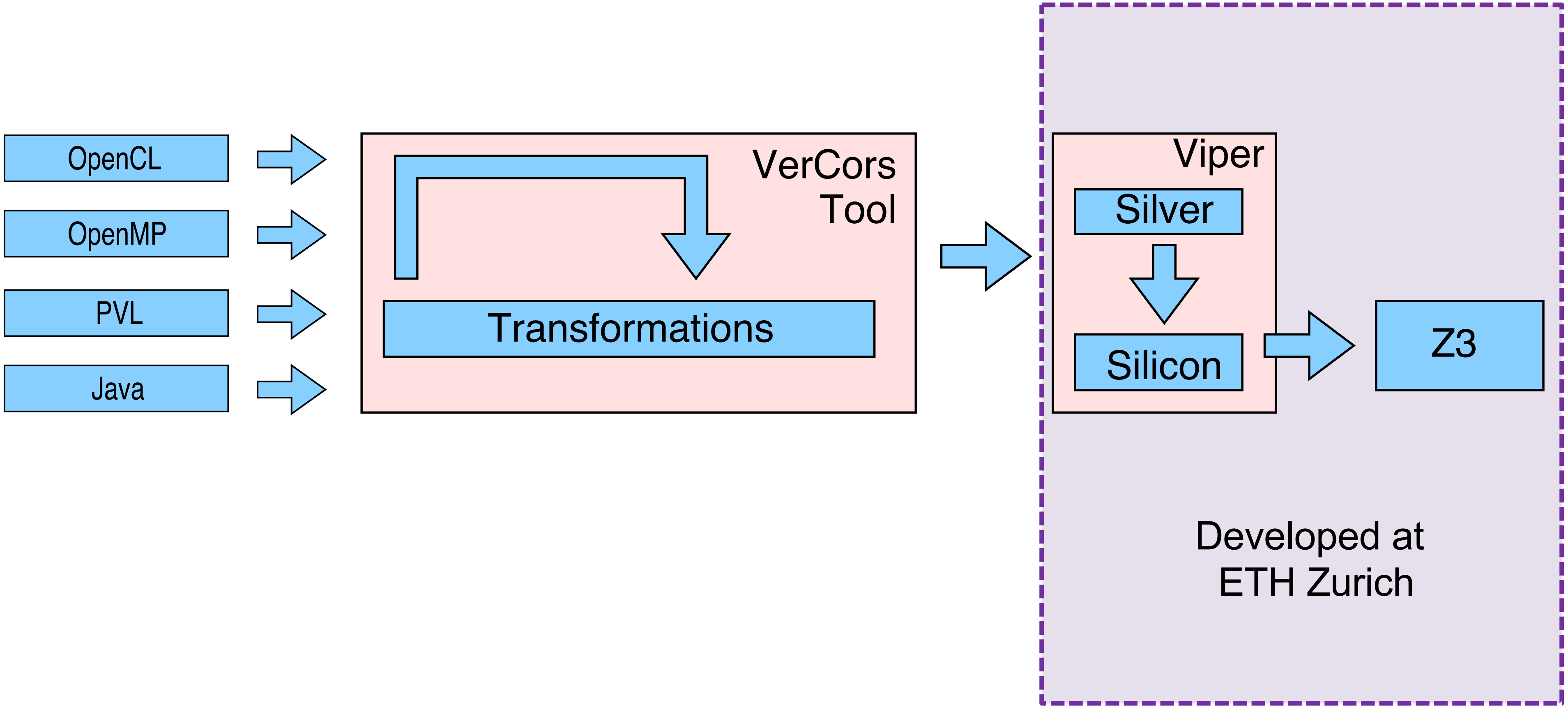
REASONING WITH PERMISSIONS

- Permissions: fractional value between 0 and 1
  - Write permission: exclusive access (encoded by 1)
  - Read permission: shared access (encoded by fractional value between 0 and 1)
- Global invariant: for each heap location, the sum of all the permissions in the system is never more than 1
- Read and write permissions can be exchanged whenever threads synchronise
- Permissions can be split and combined

$$\mathrm{Perm}(x, 1) \ast - \ast \ \mathrm{Perm}(x, \tfrac{1}{2}) \ast \mathrm{Perm}(x, \tfrac{1}{2})$$

- Permission specifications frame functional properties

VERCORS TOOL ARCHITECTURE



See iFM 2017

VERCORS HIGHLIGHTS

Automated verification of concurrent software

- Different concurrency programming languages and paradigms

Java NVIDIA CUDA OpenCL OpenMP SYSTEMC SYCL

- Correctness preservation of program transformations [TACAS 2022, 2024]
- Reasoning about many language features [FMICS 2021]
- Functional program properties by means of abstraction [VMCAI 2020]
- Annotation generation [JSS 2024, HCVS 2024]

VERCORS CASE STUDIES

- GPU examples
    - Prefix sum
    - Summed area table
    - Parallel Bellman--Ford Algorithm
- Parallel nested depth-first search
- Red-black tree and parallel merge
- Kahn’s topological sort
- ArrayList
- Tunnel control software
- Distributed locks

EXAMPLE VERIFICATIONS

EXAMPLE: CLEAR ALL ELEMENTS

```
context_everywhere A != null;
context_everywhere (\forall* int j; 0 <= j && j < A.length; Perm(A[j],write));
ensures (\forall int j; 0 <= j && j < A.length; A[j] == 0);
void clear(int[] A) {
  int i = 0;

  loop_invariant 0 <= i && i <= A.length;
  loop_invariant (\forall int j; 0 <= j && j < i; A[j] == 0);
  while (i < A.length) {
    A[i] = 0;
    i = i + 1;
  }
}
```

context_everywhere:
throughout the method

CLEAR IN PARALLEL

```
context_everywhere A != null;
context (\forall* int j; 0 <= j && j < A.length; Perm(A[j], write));
ensures (\forall int j; 0 <= j && j < A.length; A[j] == 0);
void clearPar(int[] A) {
    par (int tid = 0 .. A.length)
        requires Perm(A[tid], write);
        ensures Perm(A[tid], write);
        ensures A[tid] == 0;
    {
        A[tid] = 0;
    }
}
```

context:
requires + ensures

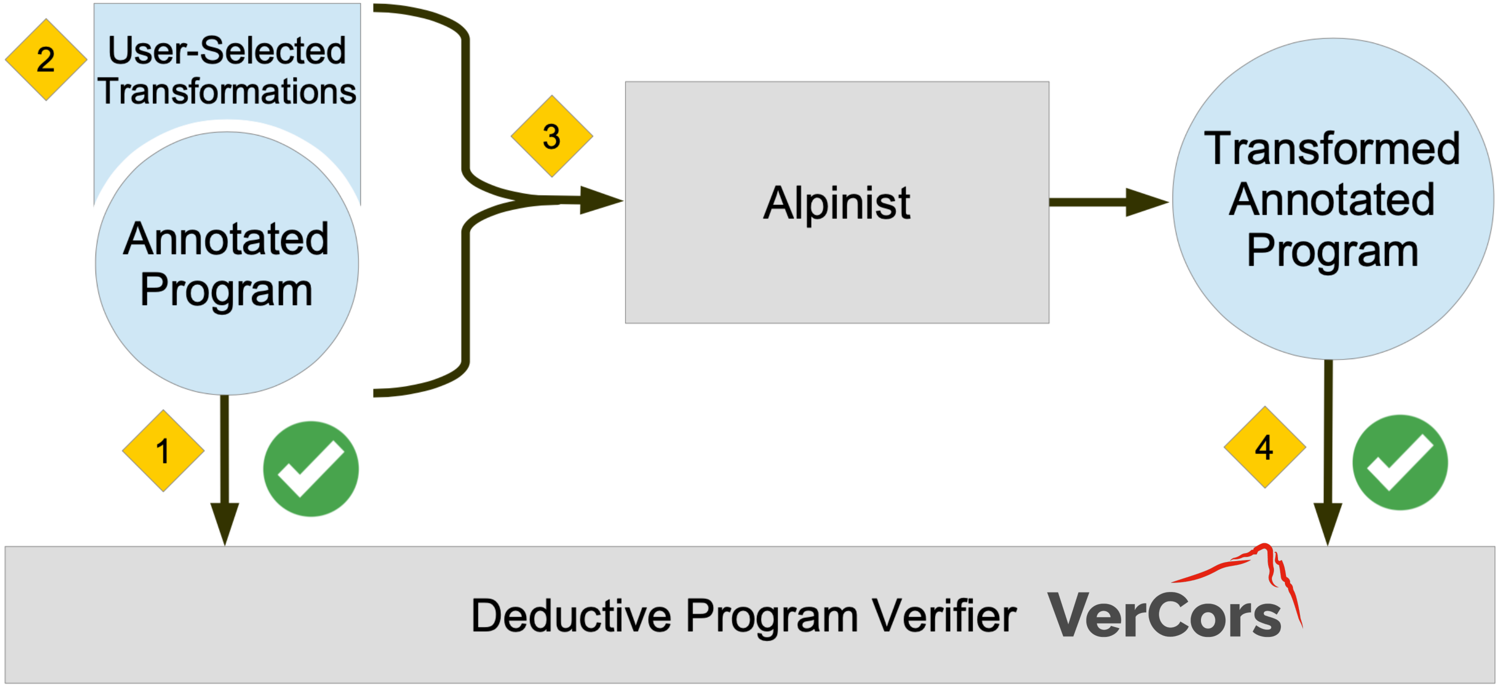
SUMMING AN ARRAY IN PARALLEL

```
resource lock_invariant() = Perm(this.sum, 1);

context_everywhere A != null;
context (\forall* int i; 0 <= i && i < A.length; Perm(A[i], 1\2));
void sum(int[] A) {
    par (int tid = 0 .. A.length)
        requires Perm(A[tid], 1\2);
        ensures Perm(A[tid], 1\2);
    {
        lock this;
        sum = sum + A[tid];
        unlock this;
    }
}
```

ANNOTATION-AWARE OPTIMISATIONS

ALPINIST: ANNOTATION-AWARE OPTIMISATIONS



2 User-Selected Transformations

Annotated Program

3

Alpinist

Transformed Annotated Program

1 ✓

4 ✓

Deductive Program Verifier VerCors

SUPPORTED OPTIMISATIONS

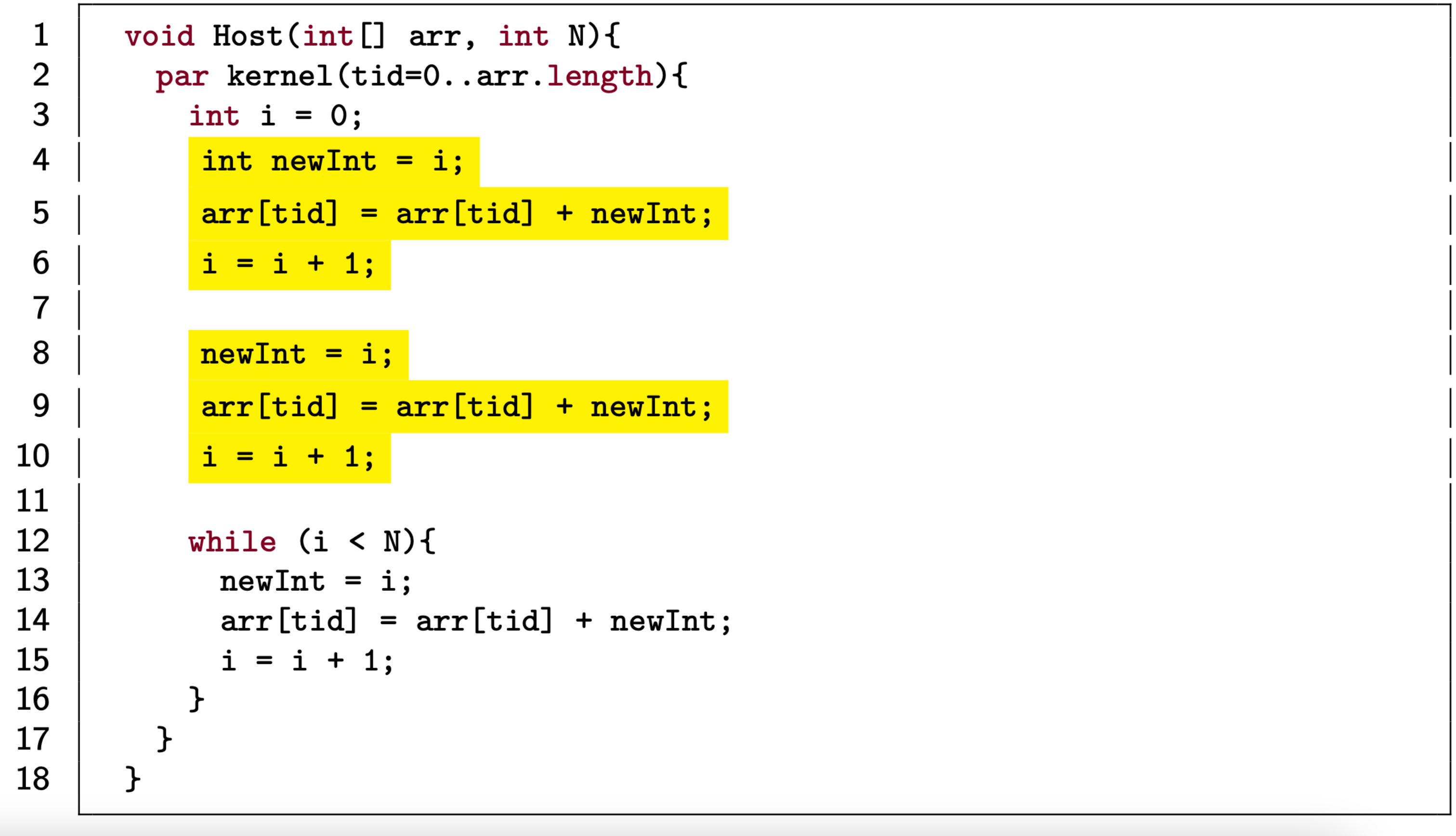
- Loop unrolling
- Kernel fusion
- Tiling
- Iteration merging
- Matrix linearization
- Data prefetching

LOOP UNROLLING

```
void Host(int[] arr, int N){
  par kernel(tid=0..arr.length){
    int i = 0;
    while (i < N){
      int newInt = i;
      arr[tid] = arr[tid] + newInt;
      i = i + 1;
    }
  }
}
```

Starting kernel

KERNEL WITH UNROLLING



```
void Host(int[] arr, int N){
  par kernel(tid=0..arr.length){
    int i = 0;
    int newInt = i;
    arr[tid] = arr[tid] + newInt;
    i = i + 1;

    newInt = i;
    arr[tid] = arr[tid] + newInt;
    i = i + 1;

    while (i < N){
      newInt = i;
      arr[tid] = arr[tid] + newInt;
      i = i + 1;
    }
  }
}
```

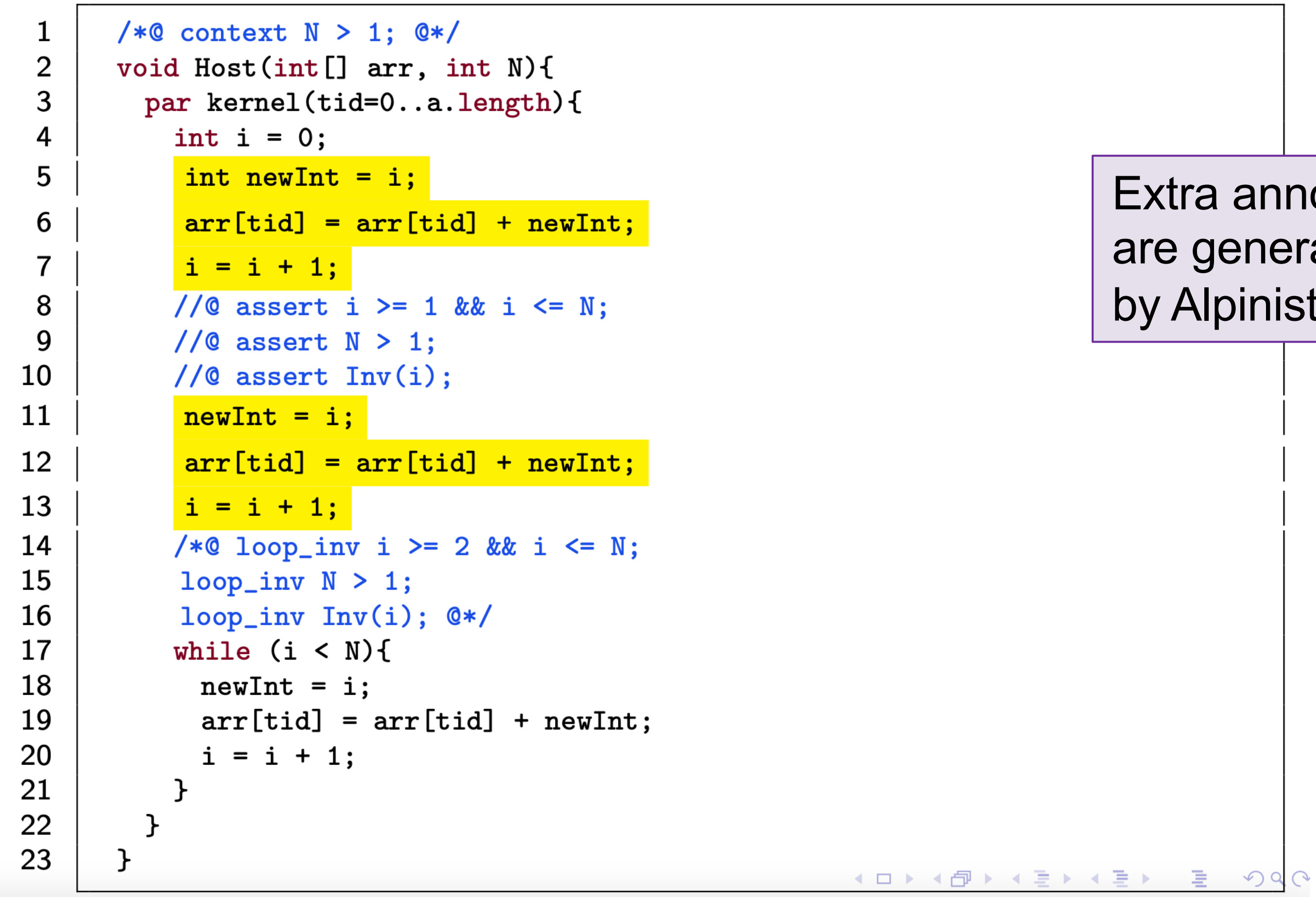
LOOP UNROLLING WITH ANNOTATIONS

```
/*@ context N > 1; @*/
void Host(int[] arr, int N){
  par kernel(tid=0..arr.length){
    int i = 0;
    /*@ loop_inv i >= 0 && i <= N;
      loop_inv N > 1;
      loop_inv Inv(i); @*/
    while (i < N){
      int newInt = i;
      arr[tid] = arr[tid] + newInt;
      i = i + 1;
    }
  }
}
```

N: array length, non-empty array

Alpinist checks that unrolling is possible (can be derived from precondition)

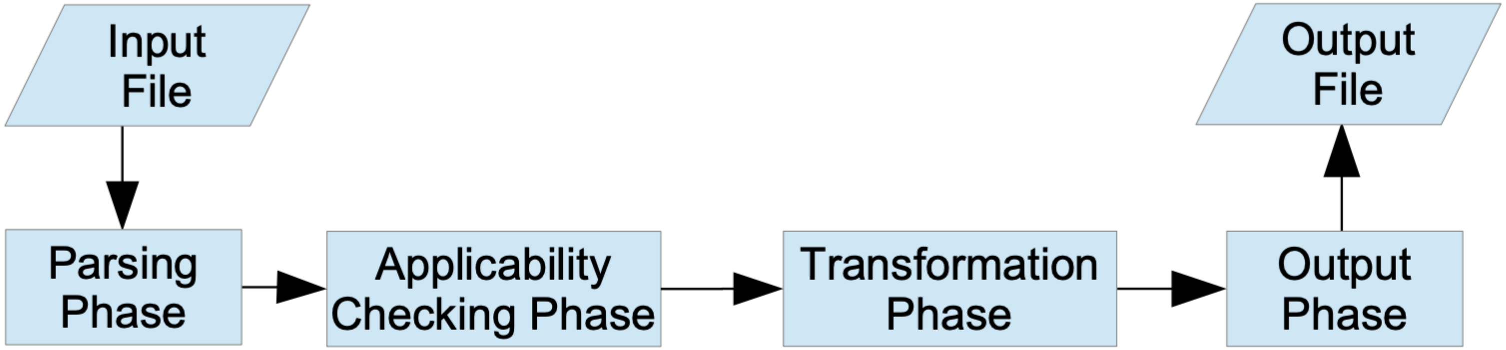
ANNOTATION-AWARE LOOP UNROLLING



```
/*@ context N > 1; @*/
void Host(int[] arr, int N){
 par kernel(tid=0..a.length){
  int i = 0;
  int newInt = i;
  arr[tid] = arr[tid] + newInt;
  i = i + 1;
  //@ assert i >= 1 && i <= N;
  //@ assert N > 1;
  //@ assert Inv(i);
  newInt = i;
  arr[tid] = arr[tid] + newInt;
  i = i + 1;
  /*@ loop_inv i >= 2 && i <= N;
  loop_inv N > 1;
  loop_inv Inv(i); @*/
  while (i < N){
   newInt = i;
   arr[tid] = arr[tid] + newInt;
   i = i + 1;
  }
 }
}
```

Extra annotations are generated by Alpinist

ALPINIST ARCHITECTURE



SYSTEM C VERIFICATION

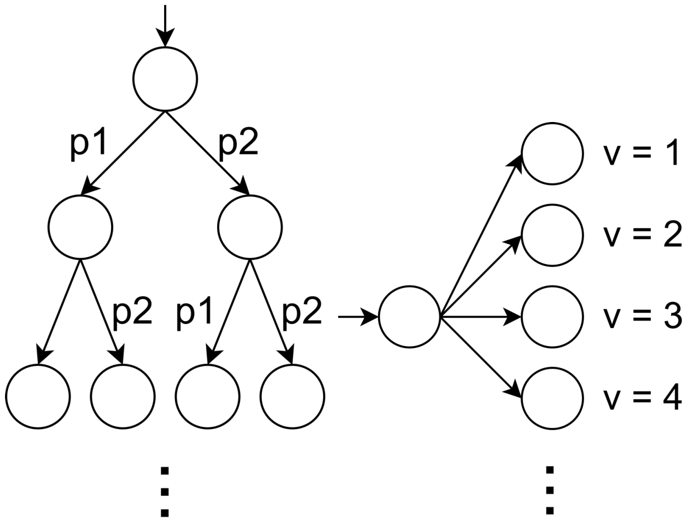
SYSTEM C

- C++ library with time, reactivity, hardware data types
- Used for hardware/software co-design
  - System design organized into modules
  - Communication via channels
  - Concurrent processes with cooperative scheduling
  - Synchronization via (time-delayed) events in discrete-event simulation

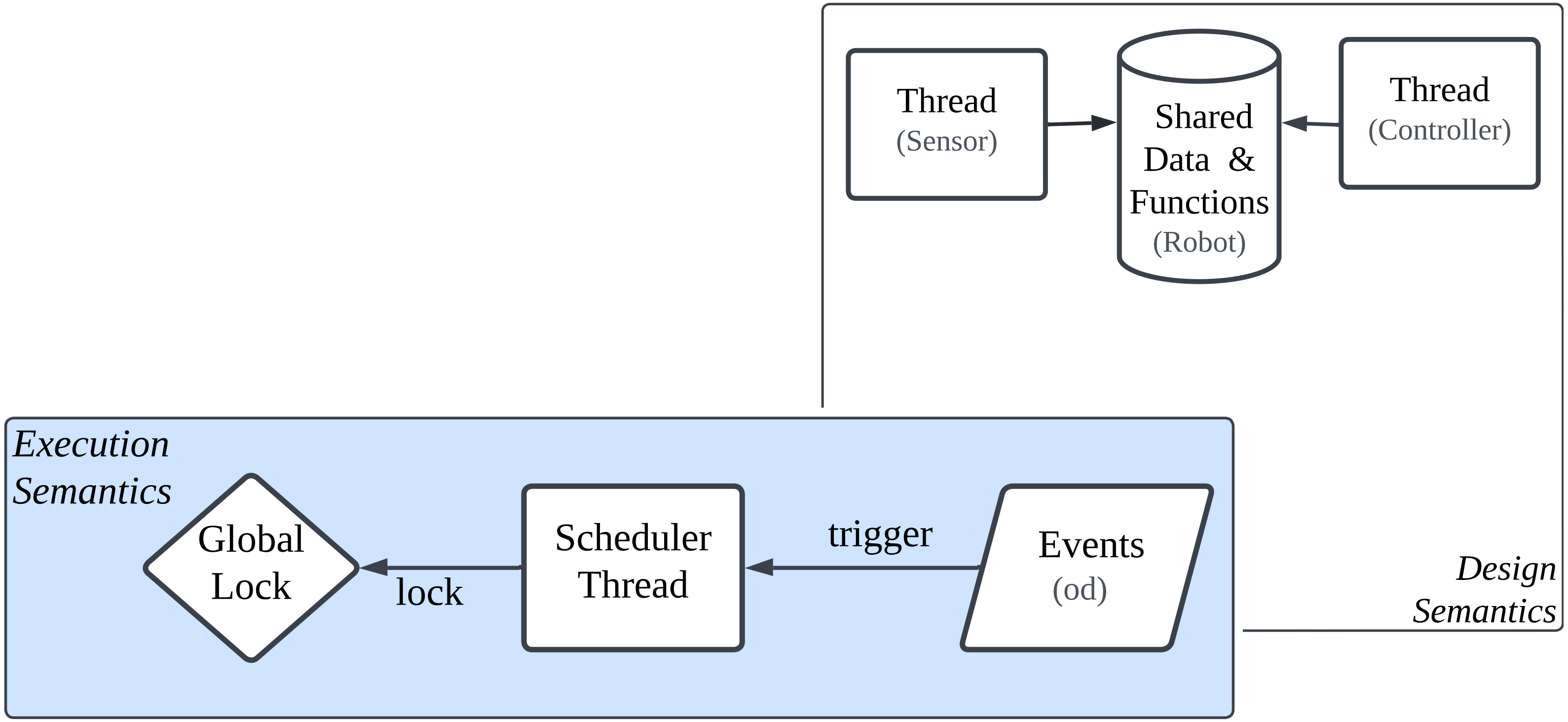
SYSTEM C VERIFICATION

- Current verification approaches for SystemC rely on model checking
- Highly automatic
- Limited scalability with regards to
  - State space explosion
  - Unbounded program data

→ Solution: Deductive verification!



ENCODING SYSTEM C DESIGNS IN PVL



ENCODING A PROCESS

```
void sensor() {
    while(true) {
        wait(2, SC_MS);
        dist = read_sensor();
        if(dist < MIN_DIST) {
            flag = true;
        }
    }
}
```

```
void run() {
   lock(m);
   while (true) {
        m.event_state[1] = 2;
        m.process_state[0] = 1;
        while (m.process_state[0] != RUNNABLE
                   || m.event_state[1] != OCCURED) {
            unlock(m);
            lock(m);
        }
       dist = read_sensor();
       if (dist < MIN_DIST) {
           flag = true;
       }
   }
   unlock(m);
}
```

ENCODING THE SCHEDULE

```
while (true) {
    lock(this);
    immediate_wakeup();
    reset_events_no_delta();
    if (no_process_ready()) {
        reset_occurred_events();
        int d = min_advance(event_state);
        advance_time(d);
        wakeup_after_wait();
        reset_all_events();
        }
    unlock (this)
}
```

- Which event should occur next
- Which processes should be woken up
- Advance time by subtracting the due time

VERIFYING PROPERTIES

Functional properties

- Local behavior
- Strength of deductive verification
- Function contracts, local assertions

`assert slack < THRESHOLD;`

Global properties

- Involve timing, process interaction, events
- Dependent on global behavior
- Hard to verify locally

`assert event_state[3] != -1 ==> other.pc == 4;`

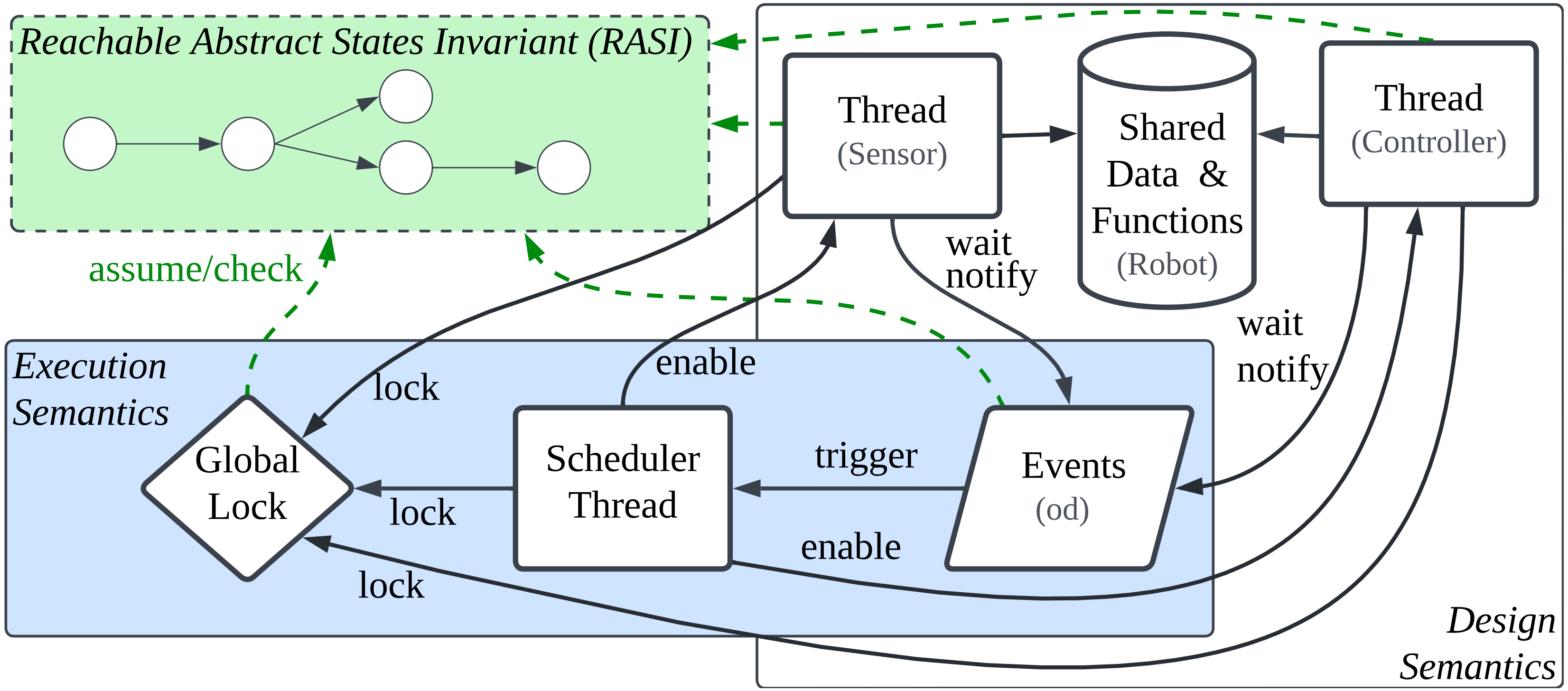
- To verify global properties, need connection between local and global state
- Solution: global invariant

REACHABLE ABSTRACT STATES INVARIANT

```
resource global_invariant() =
    ...
// Abstract state enumeration - potentially large, but automatable
** (event_state[0] != -1 && event_state[0] != 0)
** (event_state[2] <= -1)
** (sensor.pc == 0 ==> event_state[0] == -3)
** ((event_state[2] == -1 || event_state[2] == -2) ==> event_state[0] == 2)
** (event_state[1] >= -1 ==> event_state[0] == event_state[1] + 1)
** (!(event_state[0] < -1 && event_state[1] == -2))
// Some manual invariants are still necessary
** (event_state[2] >= -1 ==> sensor.dist < MIN_DIST)
** (event_state[2] == -2 ==> sensor.dist < MIN_DIST)
** (event_state[1] >= -1 ==> sensor.dist < MIN_DIST)
** (event_state[1] == -2 ==> sensor.dist < MIN_DIST);
```

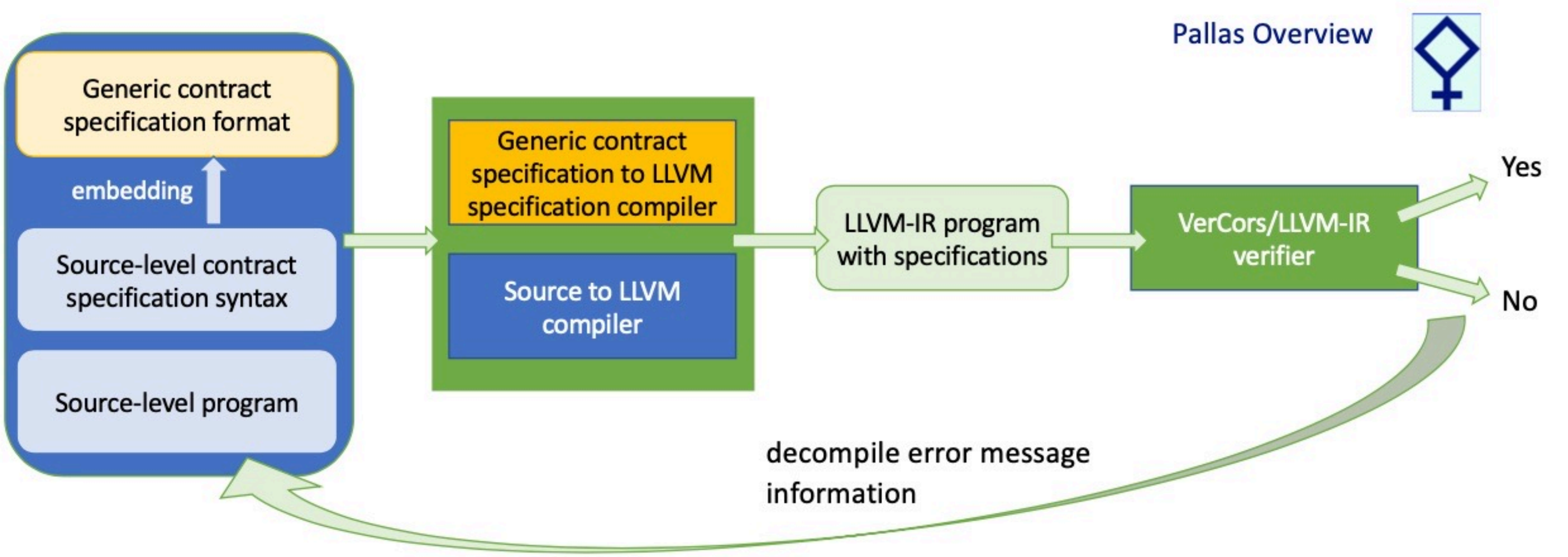
- User effort to connect local and global state is very high
- Use abstract state space enumeration to improve automation

PUTTING IT ALL TOGETHER



LLVM VERIFICATION

PALLAS: OVERALL IDEA



Pallas Overview

Generic contract specification format

embedding

Source-level contract specification syntax

Source-level program

Generic contract specification to LLVM specification compiler

Source to LLVM compiler

LLVM-IR program with specifications

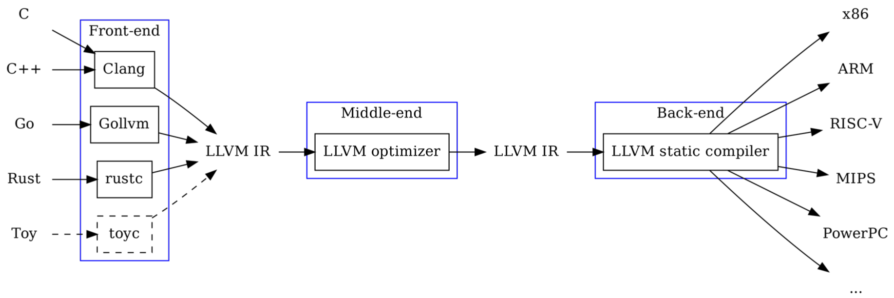
VerCors/LLVM-IR verifier

Yes

No

decompile error message information

LLVM IR



```
define i32 @addMult(i32 %x, i32 %y, i32 %z)
{ %1 = mul i32 %y, %x
  %res2 = add i32 %1, %z
  ret i32 %res2
 }
```

- Assembly language
- Single static assignment
- Block structure
- Basic types: int & float, aggregate types
- Stable API

CHALLENGES FOR LLVM IR DEDUCTIVE VERIFIER

- Instability of LLVM IR
- Suitable specification language
- Origin of user errors
- Control flow reconstruction (identify loop components)
- Low-level language features (loads, stores, ϕ-nodes)
- LLVM Concurrency Model
- Special constants: undef (undefined state), poison (erroneous state)

LLVM-IR VERIFICATION: CURRENT STATE

- Only works for C programs
- Compile C to LLVM IR
- Use opt tool to turn into suitable fragment of LLVM IR
- Annotate LLVM IR program manually
- Encode into interal VerCors format
- Verify with VerCors

SOME VERIFIED LLVM IR PROGRAMS

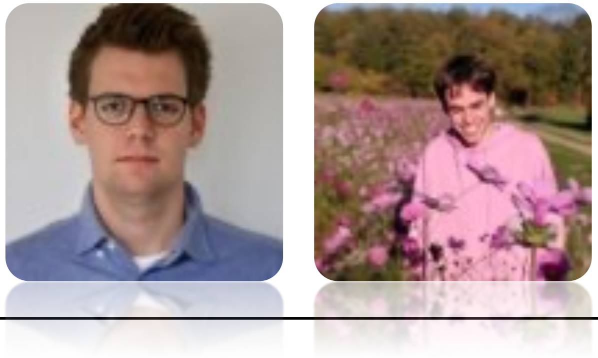
- Computation of triangular numbers and Cantor pairs
- Date comparison
- Fibonnaci and factorial, specified with support for pure functions

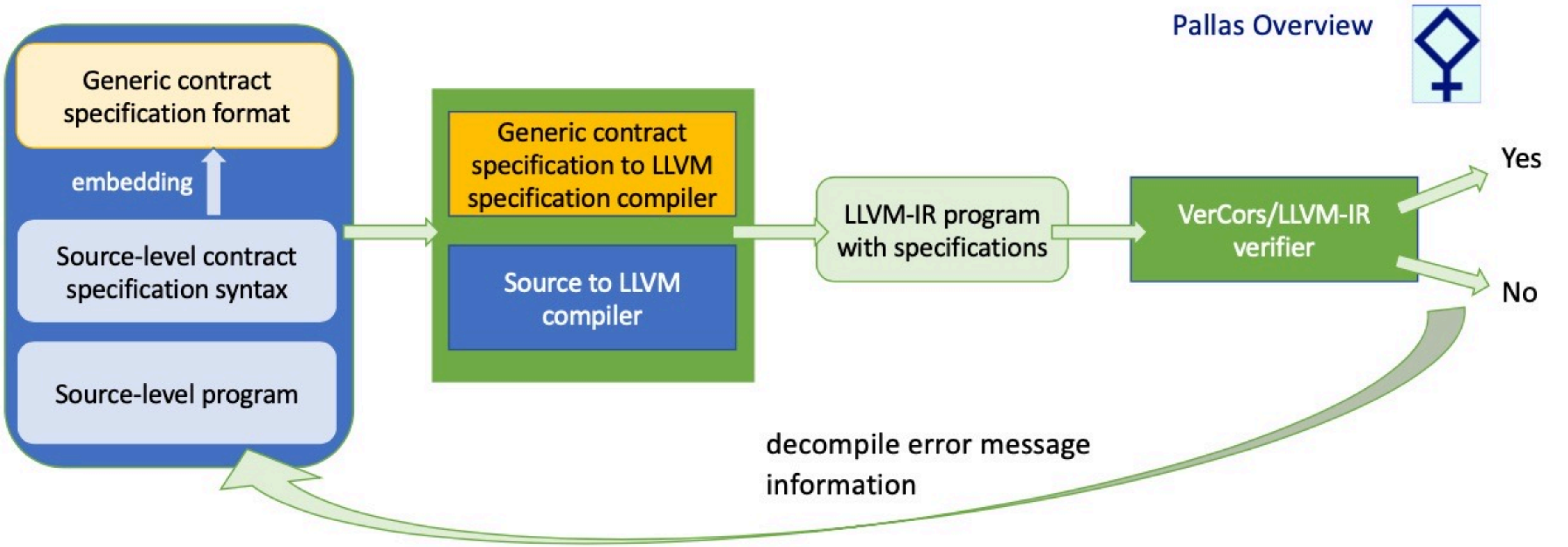
```
!VC.global = !{!0}
!0 = !{
 !"pure i32 @fib(i32 %n) =
     br(icmp(sgt, %n, 2),
        add(call @fib(sub(%n, 1)), call @fib(sub(%n, 2))),1);”}
```

```
!"ensures icmp(eq, \result, call @fib(%0));"
```

NEXT STEPS

VerCors





Pallas Overview

Generic contract specification format

embedding

Source-level contract specification syntax

Source-level program

Generic contract specification to LLVM specification compiler

Source to LLVM compiler

LLVM-IR program with specifications

VerCors/LLVM-IR verifier

Yes

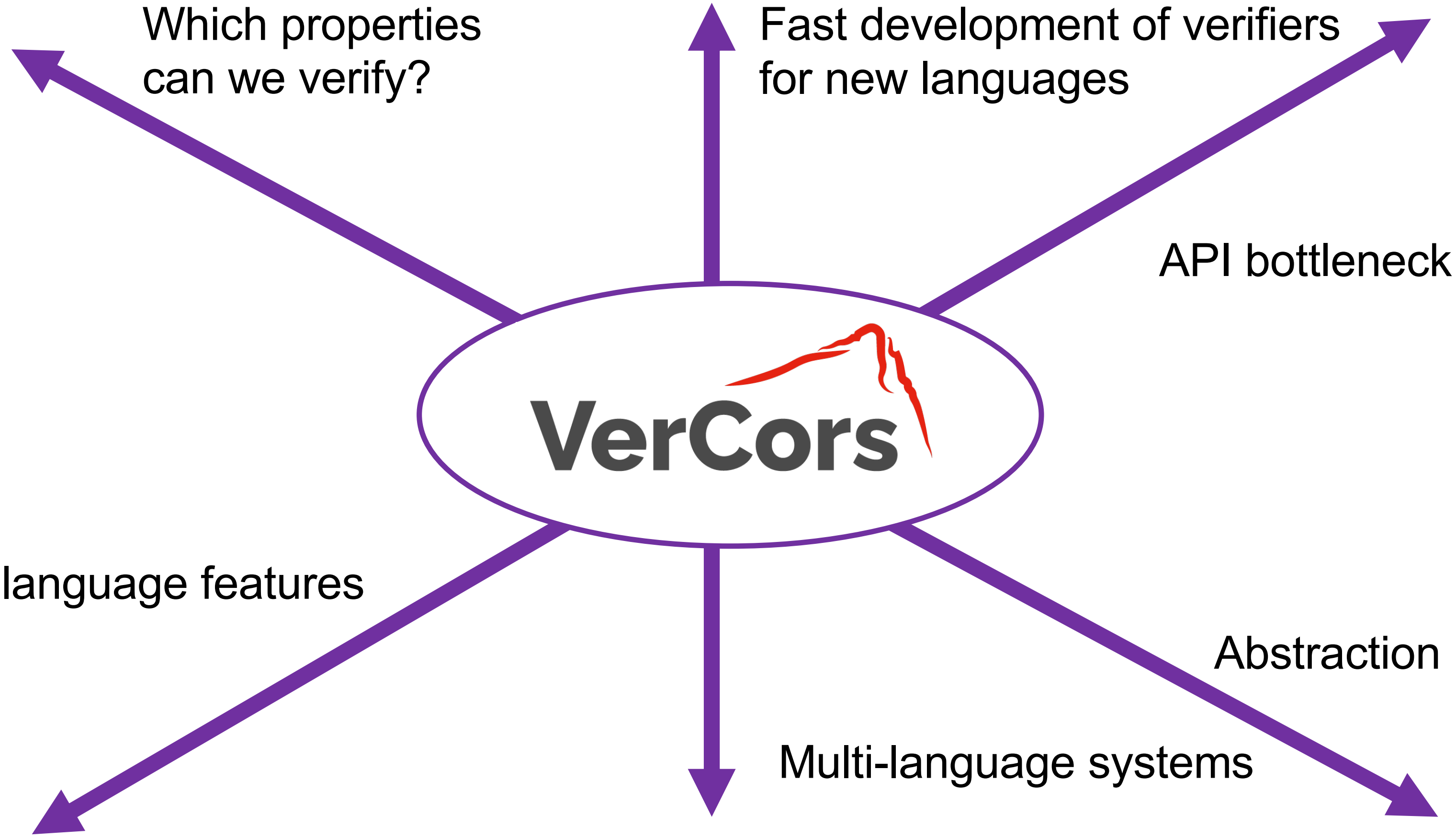
No

decompile error message information

Support for more language features
Generic specification format
Generic translation, parametrised by compiler
Effective feedback at source level

LONG-TERM IDEAS

DIFFERENT RESEARCH DIRECTIONS



Which properties can we verify?

Fast development of verifiers for new languages

API bottleneck

VerCors

New language features

Abstraction

Multi-language systems

Automate

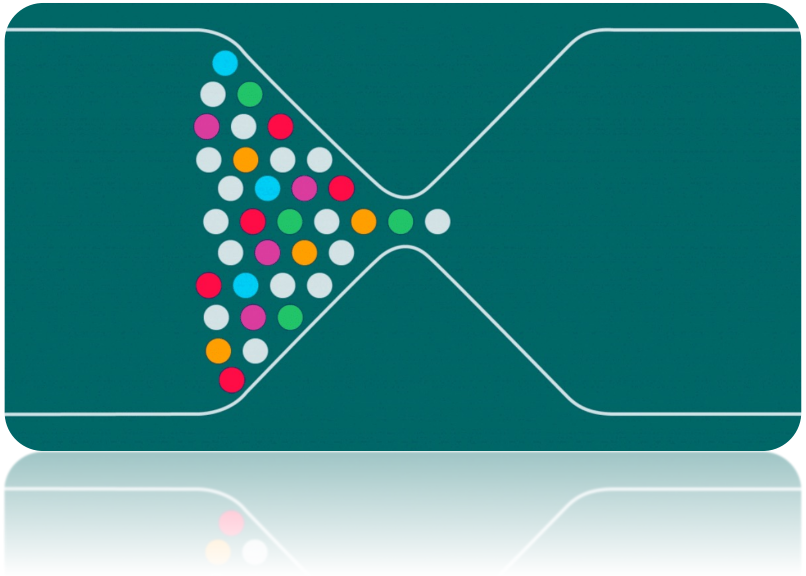
FAST DEVELOPMENT OF PROGRAM VERIFIERS

- Identify the well-understood core for program verification for reuse
- Use of LLMs to construct deductive verifiers or other forms of automation?
- Seamless integration between static and dynamic verification

Automate

API BOTTLENECK

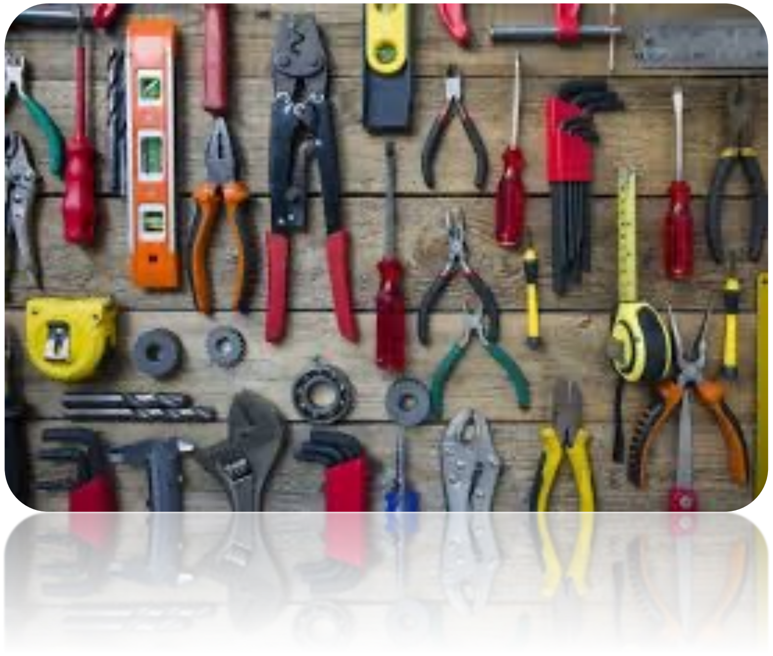
- Annotation generation
- Translation of annotations
- Common contract exchange format
- Potential use of LLM



Extend

MISSING AND COMBINING LANGUAGE FEATURES

- Structs and pointers
- Floats
- Dynamic typing
- Reflection
- Generics/templates
- Streams
- …



Extend

BEYOND FUNCTIONAL CORRECTNESS

- Behavioural properties: global flow
- Security
- Energy consumption

Scale

ABSTRACTION LEVEL OF VERIFICATION

- Large systems are hard to verify
- Layers of verification
- Trusted refinement

Scale

VERIFICATION FOR MULTI-LANGUAGE SYSTEMS

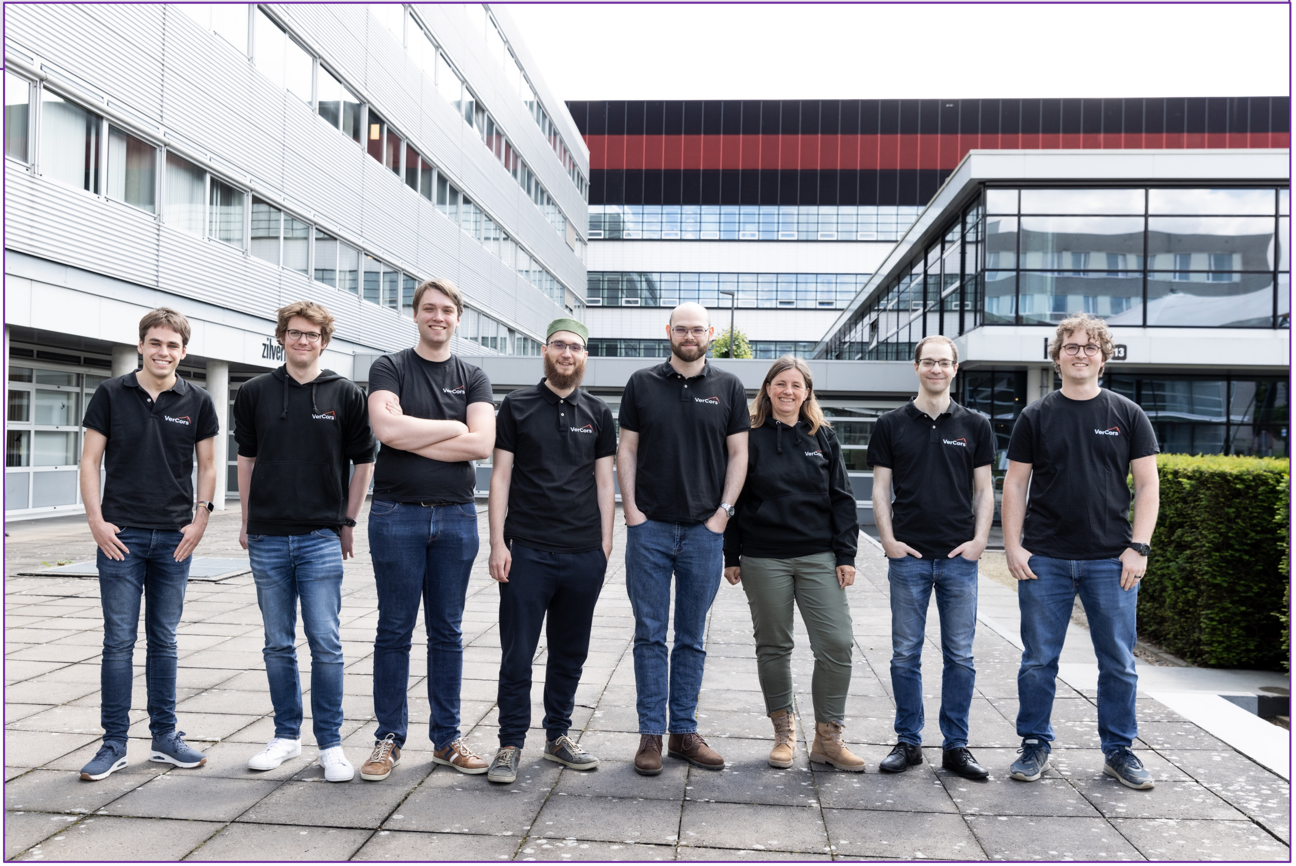
- Generic ways to target the semantic differences between different programming languages
- Verification of interaction with lowest layer (sensors…)

TO CONCLUDE

- Long line of work on tool-supported software verification
- VerCors: program verification for concurrent software
- Concurrency support:
  - Resource invariants to reason about lock-protected data (synchronisers)
  - Parallel blocks
- Alpinist: preserve verifiability of programs while optimizing for performance
- Pallas: Verification of LLVM programs
- Future work
  - Automate, extend and scale

ACKNOWLEDGEMENTS

Afshin Amighi, Lukas Armborst, Stefan Blom, Petra van den Bos, Pieter Bos, Saeed Darabi, Lars van den Haak, Paula Herber, Sebastiaan Joosten, Sophie Lathouwers, Robert Mensing, Raúl Monti, Wojciech Mostowski, Henk Mulder, Wytse Oortwijn, Bob Rubbens, Ömer Sakar, Alexander Stekelenburg, Philip Tasche, Naum Tomov, Anton Wijs, Marina Zaharieva, and many BSc and MSc students



THE END...

Automated verification of concurrent software

VerCors

More information and try the tool:
http://www.utwente.nl/vercors