Hunting Bugs
In Multithread C Programs

Omar Inverso University of Southampton, UK

Ermenegildo Tomasco University of Southampton, UK

Bernd Fischer Stellenbosch University, South Africa

Salvatore La Torre Università di Salerno, Italy

Gennaro Parlato University of Southampton, UK

UNIVERSITÀ DEGLI STUDI DI SALERNO

UNIVERSITY OF Southampton

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Finding Bugs in Software

- testing is still the most used approach in industry
  - can be slow, expensive, resource-consuming
  - no confidence that there are no missed bugs, or no bugs at all

- manual source-code inspection can spot very subtle bugs
  - inefficient, error-prone, unfeasible on big programs

- automatic source-code analysis is generally undecidable **[Church 1936] [Turing 1936]**
  - approximation: missed bugs *vs* false positives
  - restrictions: focus on *specific classes of programs* and *specific checks*
    (array bounds, division-by-zero, assertion violation, …)

Finding Bugs in Software

- testing is still the most used approach in industry
  - can be slow, expensive, resource-consuming
  - no confidence that there are no missed bugs, or no bugs at all

- manual source-code inspection can spot very subtle bugs
  - inefficient, error-prone, unfeasible on big programs

- automatic source-code analysis is generally undecidable [Church 1936] [Turing 1936]
  - approximation: missed bugs *vs* false positives
  - restrictions: focus on *specific classes of programs* and *specific checks* (array bounds, division-by-zero, assertion violation, …)

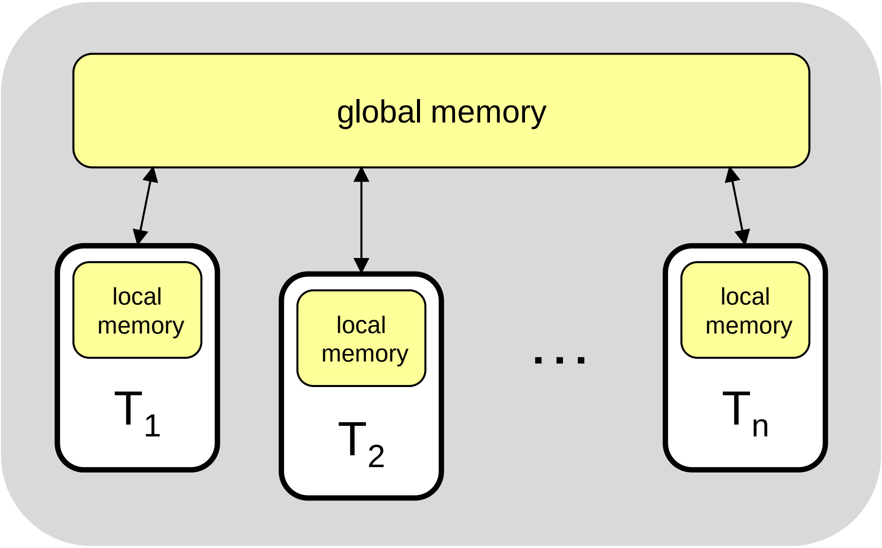
we focus on

automatic analysis of

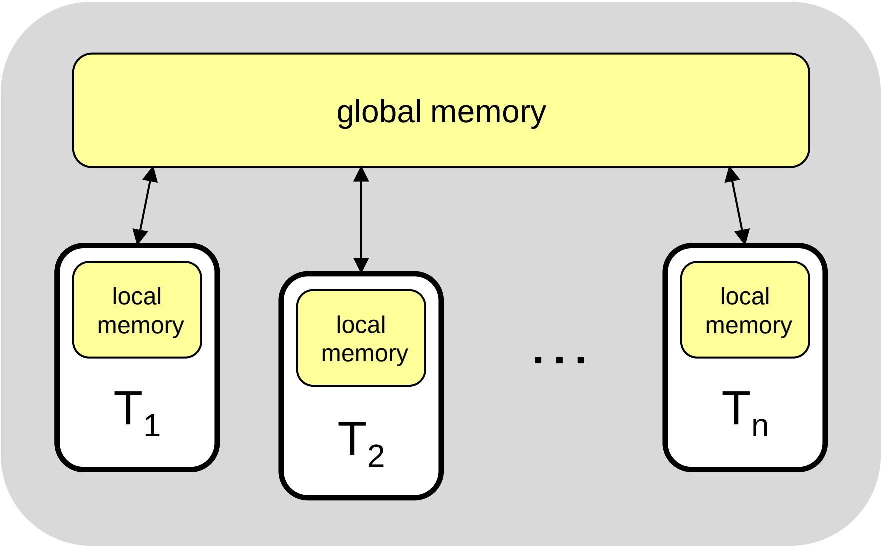
→ **multithread C programs** ←

(reachability + assertion violation)

Multithread Programs

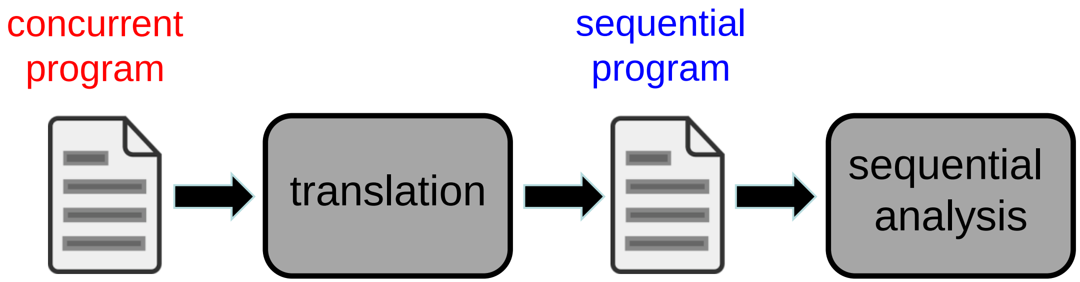


Multithread Programs

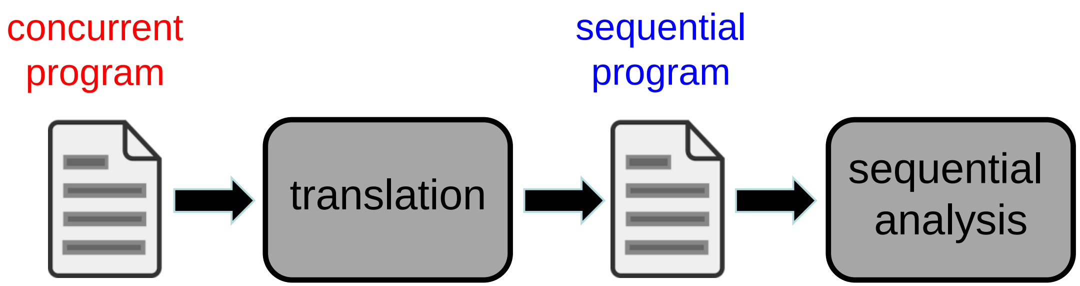


- harder development
  - thread interference must be considered
  - concurrency introduces further errors (e.g. deadlock)
- harder analysis
  - #interleavings exponential in (#threads x #statements)
  - testing not effective
  - gap with tools for sequential programs

Sequentialization

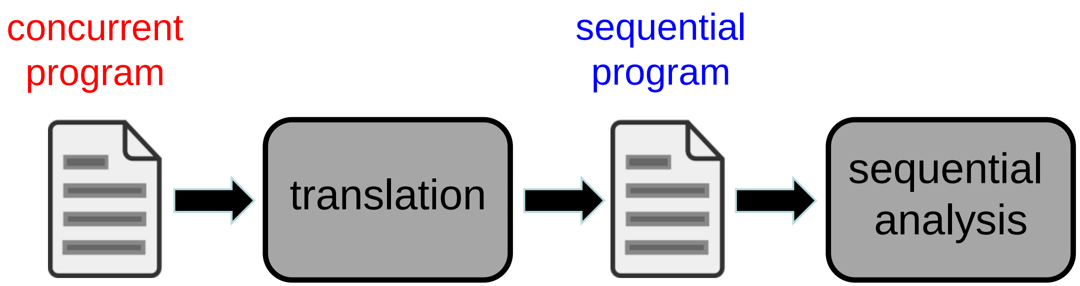


Sequentialization



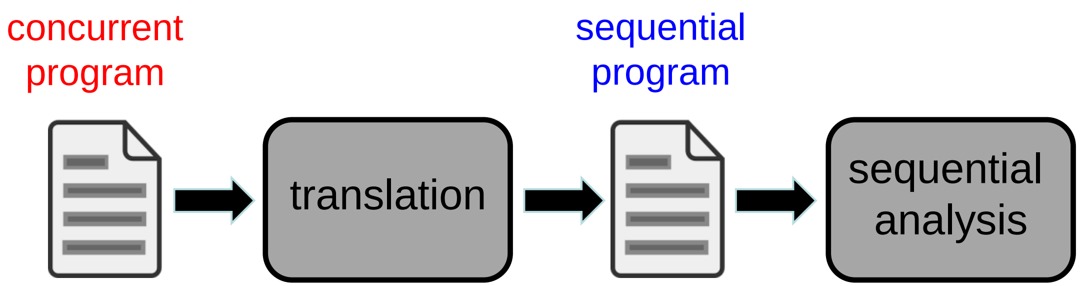
- convenience
  - re-use industrial-strength existing tools as backends
  - fast prototyping (designers can concentrate on concurrency)
  - can work with different backends

Sequentialization



- known sequentializations
  - initially proposed for up to 2 context-switches [Qadeer,Wu PLDI2004]
  - generalised to *k* round-robin context-switches [Lal,Reps CAV2008]
    - no dynamic memory allocation, dynamic thread creation, limited backend integration
    - implemented for C programs [Lahiri,Qadeer,Rakamaric CAV2009] [Fischer,Inverso,Parlato ASE2013]
  - *k* context-switches, lazy sequentialization [La Torre,Madhusudan,Parlato CAV2009]
    - not good for bounded model-checking backends
    - implemented for Boolean programs

Sequentialization

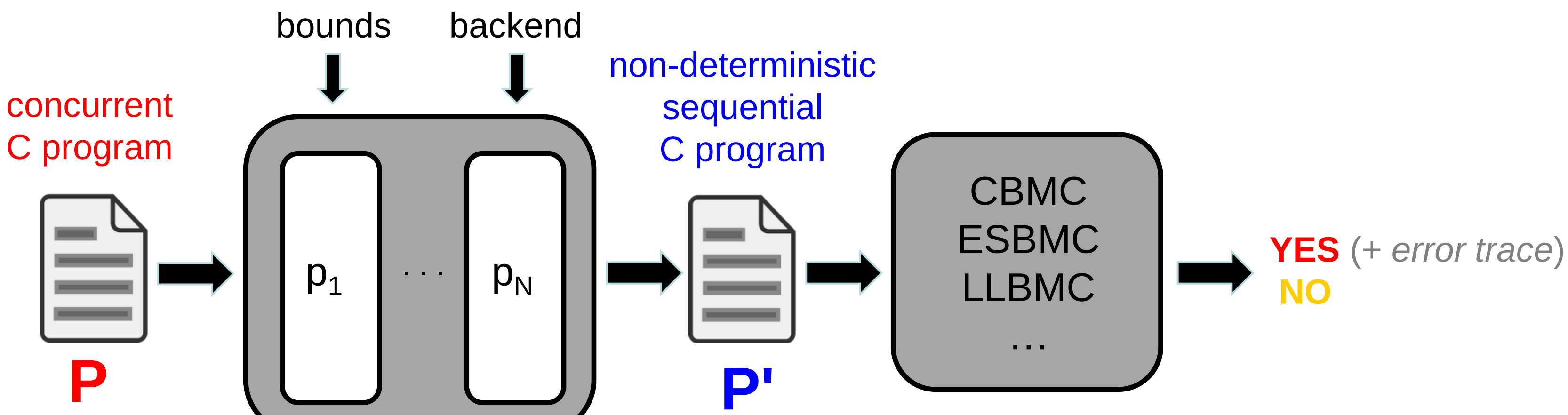


- known sequentializations
  - initially proposed for up to 2 context-switches [Qadeer,Wu PLDI2004]
  - generalised to *k* round-robin context-switches [Lal,Reps CAV2008]
    - no dynamic memory allocation, dynamic thread creation, limited backend integration
    - implemented for C programs [Lahiri,Qadeer,Rakamaric CAV2009] [Fischer,Inverso,Parlato ASE2013]
  - *k* context-switches, lazy sequentialization [La Torre,Madhusudan,Parlato CAV2009]
    - not good for bounded model-checking backends
    - implemented for Boolean programs

→ too many limitations both <u>in the schema</u> and <u>in the tool</u> ←

CSeq Sequentialization Framework

is a bug reachable in program **P** within the given bounds?



- **schema I**: lazy context-bounded analysis (bounds no. of context-switches)
  Lazy-CSeq tool [Inverso,Tomasco,Fischer,La Torre,Parlato TACAS-SVCOMP2014,CAV2014]

- **schema II**: memory-unwinding (bounds no. of shared memory writes)
  MU-CSeq tool [Tomasco,Inverso,Fischer,La Torre,Parlato TACAS-SVCOMP2014]

Schema I:
Lazy
Sequentialization
(Lazy-CSeq)

Lazy-CSeq Sequentialization

Translation **P** ⇝ **P'**:

- unwinding, inlining
- thread **T** ⇝ function **T'**
- main driver:
  - ▹ for round in [1..K]
  - ▹ for thread in [1..N]
  - ▹ $\mathbf{T'_{thread}}$ ();

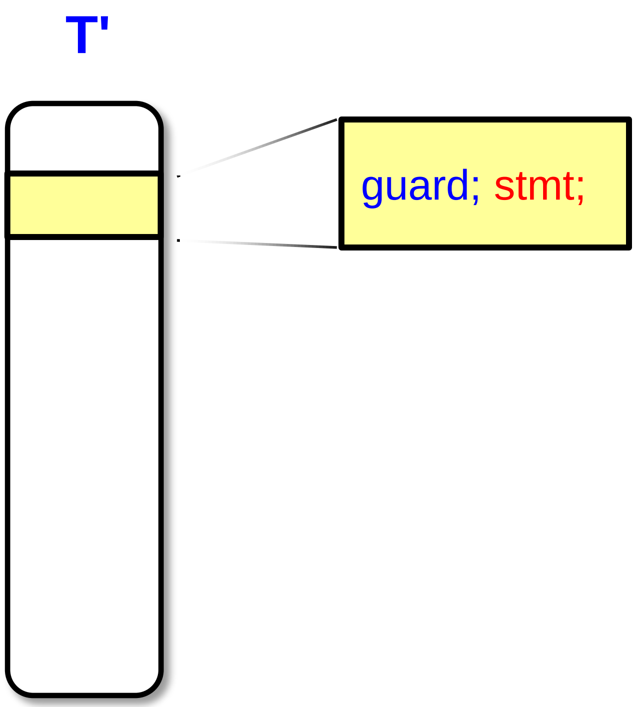
Lazy-CSeq Sequentialization

Translation **P** ⇝ **P'**:

- unwinding, inlining
- thread **T** ⇝ function **T'**
- main driver:
  - ▷ for round in [1..K]
  - ▷ for thread in [1..N]
  - ▷ $\mathbf{T'}_{\mathbf{thread}}$ ();

Thread **T** ⇝ function **T'**

- var x; ⇝ static var x; // persistency
- stmt; ⇝ guard; stmt; // context-switch



Lazy-CSeq Sequentialization

Translation **P** ⇝ **P'**:

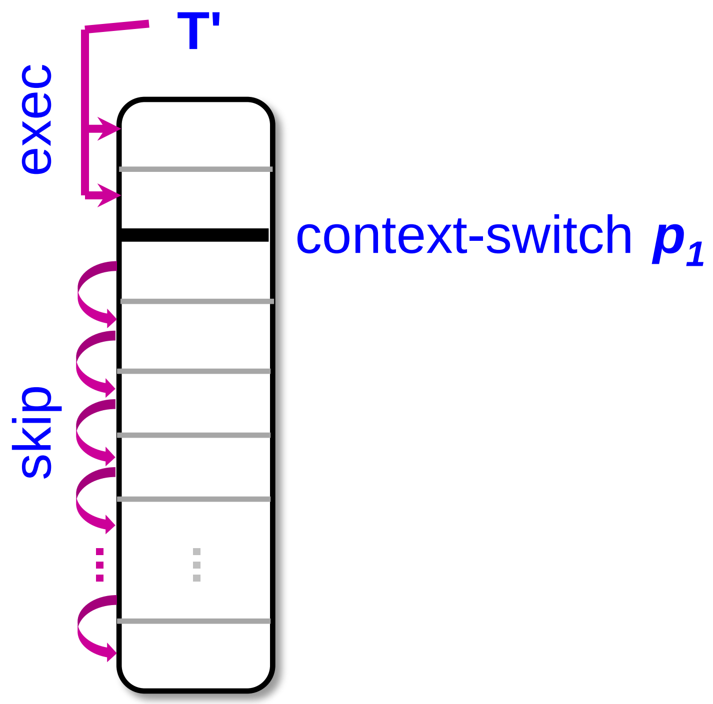
- unwinding, inlining
- thread **T** ⇝ function **T'**
- main driver:
  - ▹ for round in [1..K]
  - ▹ for thread in [1..N]
  - ▹ $\mathbf{T'}_{\mathbf{thread}}$ ();

Thread **T** ⇝ function **T'**

- var x; ⇝ static var x;
- stmt; ⇝ guard; stmt;

Thread simulation: **round 1**

- guess context-switch point p_1
- execute stmts before p_1
- jump in mult. hops to the end



simulation round 1

Lazy-CSeq Sequentialization

Translation **P** ⇝ **P'**:

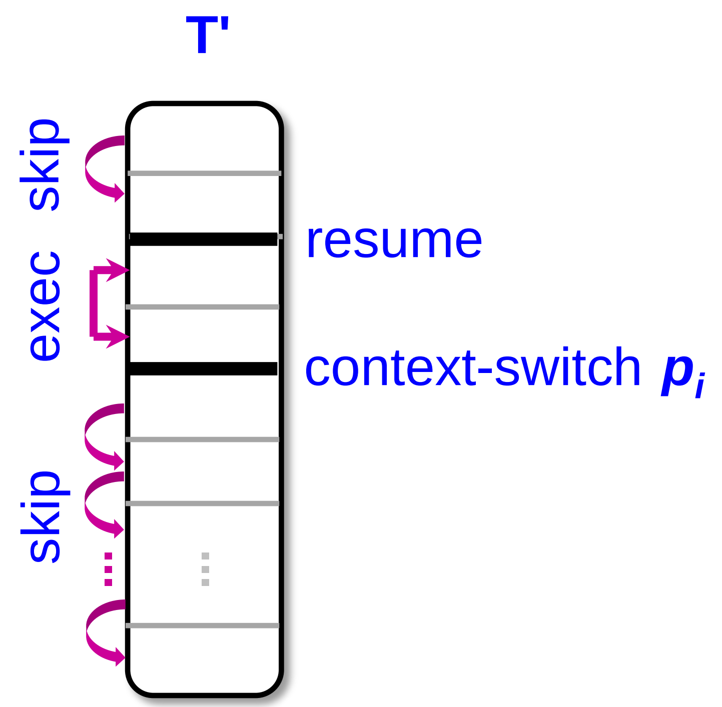
- unwinding, inlining
- thread **T** ⇝ function **T'**
- main driver:
  - ▷ for round in [1..K]
  - ▷ for thread in [1..N]
  - ▷ $\mathbf{T'}_{\mathbf{thread}}$ ();

Thread **T** ⇝ function **T'**

- var x; ⇝ static var x;
- stmt; ⇝ guard; stmt;

Thread simulation: **round *i***

- guess context-switch point p_i
- execute stmts from p_{i-1} to p_i
- jump in mult. hops to the end



simulation round *i* >1

Lazy-CSeq Example

```
pthread_mutex_t m; int c=0;
void P(void *b) {
  int tmp=(*b);
  pthread_mutex_lock(&m);
  if(c>0)
    c++;
  else {
    c=0;
    while(tmp>0) {
      c++; tmp--;
    }
  }
  pthread_mutex_unlock(&m);
}
void C() {
  assume(c>0);
  c--;
  assert(c>=0);
}
int main(void) {
  int x=1,y=5;
  pthread_t p0,p1,c0,c1;
  pthread_mutex_init(&m);
  pthread_create(&p0,P,&x);
  pthread_create(&p1,P,&y);
  pthread_create(&c0,C,0);
  pthread_create(&c1,C,0);
  return 0;
}
```

concurrent

analysis on the sequential program:

- fast bug finding
- low memory usage

```
bool active[T]={1,0,0,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define G(L) assume(cs>=L);
#define J(A,B) if(pc[ct]>A||A>=cs) goto B;
pthread_mutex_t m; int c=0;
void P0(void *b) {
 0:J(0,1) static int tmp=(*b);
 1:J(1,2) pthread_mutex_lock(&m);
 2:J(2,3) if(c>0)
 3:J(3,4)   c++;
          else { G(4)
 4:J(4,5)   c=0;
            if(!(tmp>0)) goto _l1;
 5:J(5,6)   c++; tmp--;
            if(!(tmp>0)) goto _l1;
 6:J(6,7)   c++; tmp--;
            assume(!(tmp>0));
            _l1: G(7);
          } G(7)
 7:J(7,8) pthread_mutex_unlock(&m);
          goto _P0; _P0: G(8)
 8:        return;
}
void P1(void *b) {...}
void C0() {
 0:J(0,1) assume(c>0);
 1:J(1,2) c--;
          assert(c>=0);
          goto _C0; _C0: G(2)
 2:        return;
}
void C1() {...}
int main0() {
          static int x=1,y=5;
          static pthread_t p0,p1,c0,c1;
 0:J(0,1) pthread_mutex_init(&m);
 1:J(1,2) pthread_create(&p0,P0,&x,1);
 1:J(2,3) pthread_create(&p1,P1,&y,2);
 2:J(3,4) pthread_create(&c0,C0,0,3);
 3:J(4,5) pthread_create(&c1,C1,0,4);
          goto _main; _main: G(4)
 5:        return 0;
}
int main() {...see Fig. 4...}
```

```
void main(void) {
 for(r=1; r<=K; r++) {
  ct=0;
  // only active threads
  if(active[ct]) {
   // next context switch
   cs=pc[ct]+nondet_uint();
   // appropriate value?
   assume(cs<=size[ct]);
   // thread simulation
   fseq_0(arg[ct]);
   // store context switch
   pc[ct]=cs;
  }
  .........
  ct=n;
  if(active[ct]) {
   .........
}}}
```

sequential

Schema II:
Memory-unwound Sequentialization
(MU-CSeq)

Sequentialization of Concurrent Programs

Basic Idea:

convert concurrent programs into *equivalent* sequential programs

Sequentialization of Concurrent Programs

Basic Idea:

convert concurrent programs into *equivalent* sequential programs

Mu-CSeq Approach:

Pc ⇝ M : Ps

- **M** is a guessed sequence of write operations into the shared memory
- **Ps** simulates all executions compatible with **M**
  - ▹ Simulates each thread s.t. its local computation is consistent with the memory sequence

Sequentialization of Concurrent Programs

Basic Idea:

> **convert concurrent programs into *equivalent* sequential programs**

Mu-CSeq Approach:

$$Pc \rightsquigarrow M : Ps$$

- **M** is a guessed sequence of write operations into the shared memory
- **Ps** simulates all executions compatible with **M**
  - Simulates each thread s.t. its local computation is consistent with the memory sequence

Our analysis is bounded on the number of memory write operations

Memory Unwinding

“memory”

x	y	...	z
0	0	...	0

Memory Unwinding

"memory"

pos	x	y	...	z
0	0	0	...	0
1	4	0	...	0
2	4	2	...	0
3	4	3	...	0
4	4	3	...	42
...	...	...	...	...
N	0	3	...	42

Guess and store sequence of individual write operations:

- add *N* copies of shared variables ("memory")
  `_memory[i,v]` is value of *v*-th variable after *i*-th write

Memory Unwinding

	"memory"				"writes"	
pos	x	y	…	z	thr	var
0	0	0	…	0	-	-
1	4	0	…	0	1	x
2	4	2	…	0	1	y
3	4	3	…	0	2	y
4	4	3	…	42	1	z
…	…	…	…	…	…	…
N	0	3	…	42	2	x

Guess and store sequence of individual write operations:

- add *N* copies of shared variables ("*memory*")
  `_memory[i,v]` is value of *v*-th variable after *i*-th write
- add array to record writes ("*writes*")
  i-th write is by `_thr[i]`, which has written to `_var[i]`

Simulation

Basic Idea:

simulate all executions compatible with guessed memory unwinding

Simulation

Basic Idea:

simulate all executions compatible with guessed memory unwinding

- uses auxiliary variables
  - `thread` (id of currently simulated thread)
  - `pos` (current index into unwound memory)

Simulation

Basic Idea:

simulate all executions compatible with guessed memory unwinding

- uses auxiliary variables
  - `thread` (id of currently simulated thread)
  - `pos` (current index into unwound memory)

- every thread is translated into a function
  - simulation starts from main thread

Simulation

Basic Idea:

simulate all executions compatible with guessed memory unwinding

- uses auxiliary variables
  - `thread` (id of currently simulated thread)
  - `pos` (current index into unwound memory)

- every thread is translated into a function
  - simulation starts from main thread

- each **thread creation** is translated into a function call

Simulation

Basic Idea:

simulate all executions compatible with guessed memory unwinding

- every read / write is translated into a function call

Simulating reads and writes

Basic Idea:

simulate all executions compatible with guessed memory unwinding

- every read / write is translated into a function call

```
void write(uint var_name, int val)
  pos=next_write(pos,thread);
  assume(_var[pos]== var_name
   && memory[pos, var_name]==val);
}
```

	"memory"				"writes"	
pos	x	y	...	z	thr	var
0	0	0	...	0	-	-
1	4	0	...	0	1	x
2	4	2	...	0	1	y
3	4	3	...	0	2	y
4	4	3	...	42	1	z
...	...	...	...	...	...	...
N	0	3	...	42	2	x

Simulating reads and writes

Basic Idea:

> **simulate all executions compatible with guessed memory unwinding**

- every read / write is translated into a function call

```
int read(uint var_name) {
  uint jmp=*;
  assume(jmp>=pos
   && jmp<next_write(pos,thread);
  pos=jmp;
  return _memory[pos, var_name];
}
```

"memory"

pos	x	y	...	z
0	0	0	...	0
1	4	0	...	0
2	4	2	...	0
3	4	3	...	0
4	4	3	...	42
...	...	...	...	...
N	0	3	...	42

"writes"

thr	var
-	-
1	x
1	y
2	y
1	z
...	...
2	x

Improvements

- Explicit representation of the memory

"memory"

pos	x	y	...	z
0	0	0	...	0
1	4	0	...	0
2	4	2	...	0
3	4	3	...	0
4	4	3	...	42
...	...	...	...	...
N	0	3	...	42

"writes"

thr	var
0	0
1	x
1	y
2	y
1	z
...	...
2	x

- Read and Write can be simulated using a constant number of steps

- *"memory"* size depends on the number of shared variables

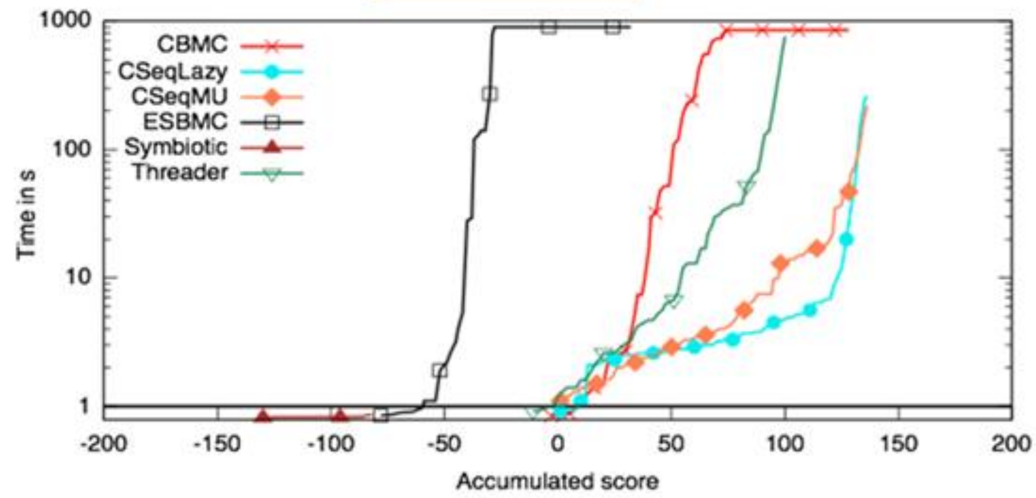
Evaluation
and
Future Work

Evaluation: SV-COMP2014

Lazy-CSeq won the **Gold Medal** and
MU-CSeq won the **Silver Medal**
in the Concurrency category

- 76 concurrent C programs
  - ▹ **UNSAFE** instances: 20 programs containing a bug
  - ▹ **SAFE** instances: all the others
- 4,500 l.o.c.

1) Lazy-Cseq: 1,000s, 136pts
2) MU-Cseq: 1,200s, 136pts
3) CBMC: 29,000s, 128pts



Results:

- small analysis times
- no missed bugs!

Future Work

- concurrency models
  POSIX threads model *Shared-Memory* concurrency,
  we plan to add support *Message Passing* (MP) programs

- memory models
  so far we assumed *Sequential Consistency* (SC),
  we plan to extend to *Weak Memory Models* (WMM)
  used in modern computer architectures

- backend support
  we have achieved fast bug-hunting with bounded model-checkers,
  we have started some preliminary work to support abstraction-based backends as well.

Thank You

users.ecs.soton.ac.uk/gp4/cseq