NEC Laboratories America
Relentless passion for innovation

www.nec-labs.com

Staged Concurrent Program Analysis*

Nishant Sinha

Joint work with **Chao Wang**

System Analysis and Verification Group,

NEC Labs, Princeton, NJ

* To appear in the Conference on Foundations of Software Engineering, November 2010.

Analyzing Concurrent Programs

- .. is HARD!
- Extensive work on analysis of Concurrent Programs
  - **Static** analysis: SPIN, Java Path Finder, …
  - **Dynamic/Runtime** analysis: Verisoft, Eraser, CHESS, …
  - **Combinations:** FUSION, …
- Wide variety of bugs: data race, deadlock, assertion violation, atomicity violation, ...

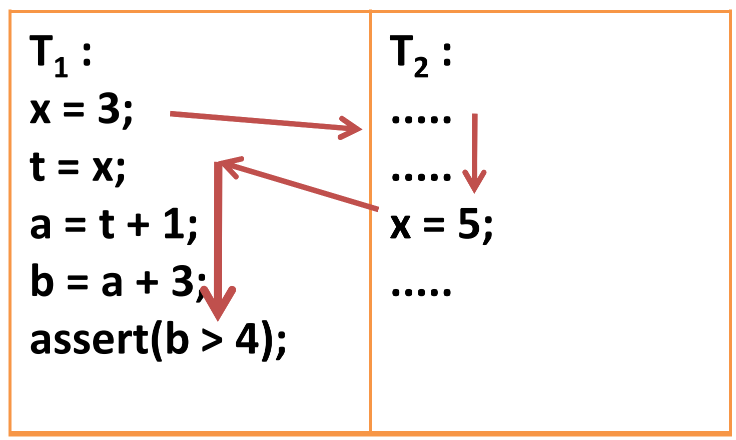
This talk

- A new **staged symbolic analysis** technique
  - static analysis
  - analyze multiple paths, schedules and inputs **simultaneously**
  - find bugs
  - sometimes, absence of bugs too
  - rethinking from basics

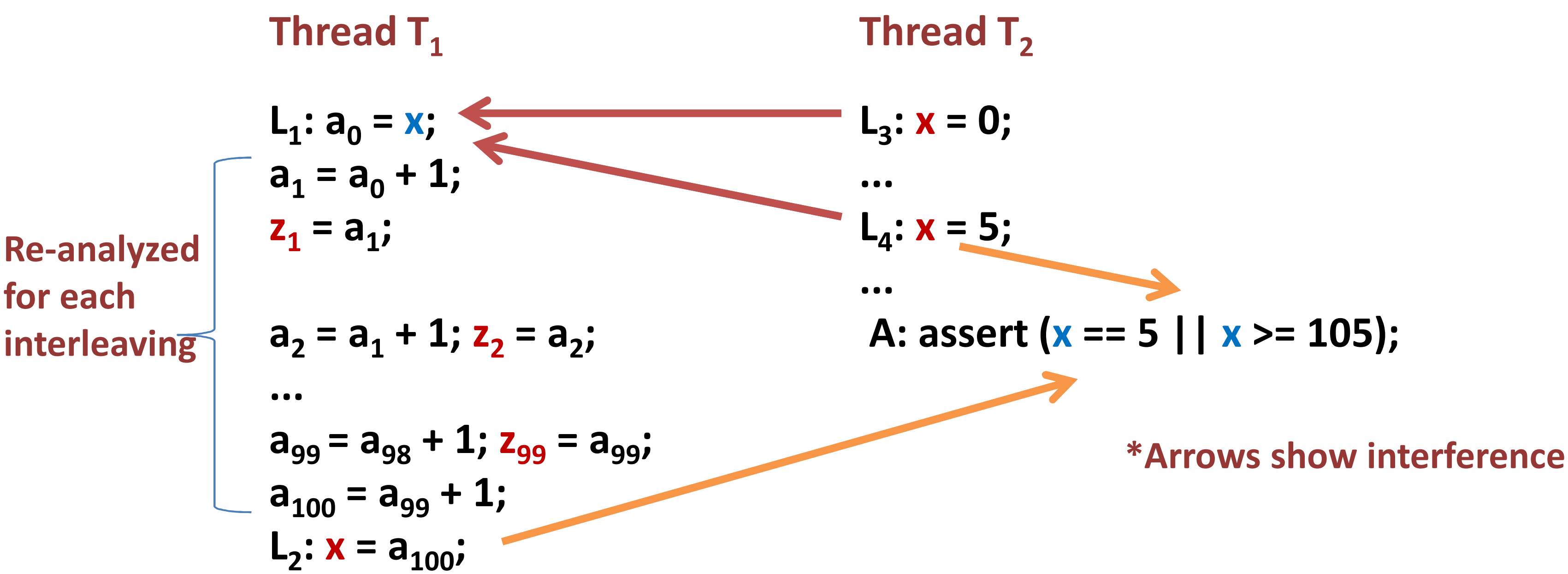
Two Sources of Inefficiency

- Bi-modal Reasoning
  - alternating intra- and inter-thread reasoning
  - duplicated intra-thread reasoning

- Scheduler
  - does not model interference directly



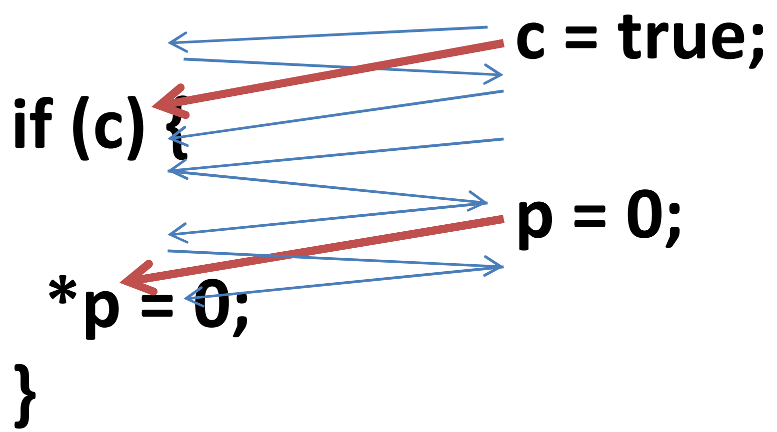
Bi-modal Reasoning



- **Goal:** Infer (a_{100} = a_0 + 100) only once (not for each interleaving)
- **Path compression** methods only work **inside atomic `transactions'**

Scheduler

- Omnipresent in concurrent analysis
  - Explicit: context-switching
  - Symbolic: auxiliary variable [V_i (sch = i) => R_i)]
- Does not model interference directly



Context-bounding helps but is not property-driven

Background: Bounded Programs

- Verifying Concurrent Programs is not decidable
  - even with finite data (Boolean Programs)
- Our focus: **Bounded Programs**
  - Loops, Recursion unrolled finitely
  - therefore, bounded thread creation and heap
  - Real programs (not Boolean)
    - contain pointers, arrays, structures, etc.
    - may contain infinite datatypes (with decidable theory)
  - Decidable
  - Witnesses found are real but Proofs may be spurious

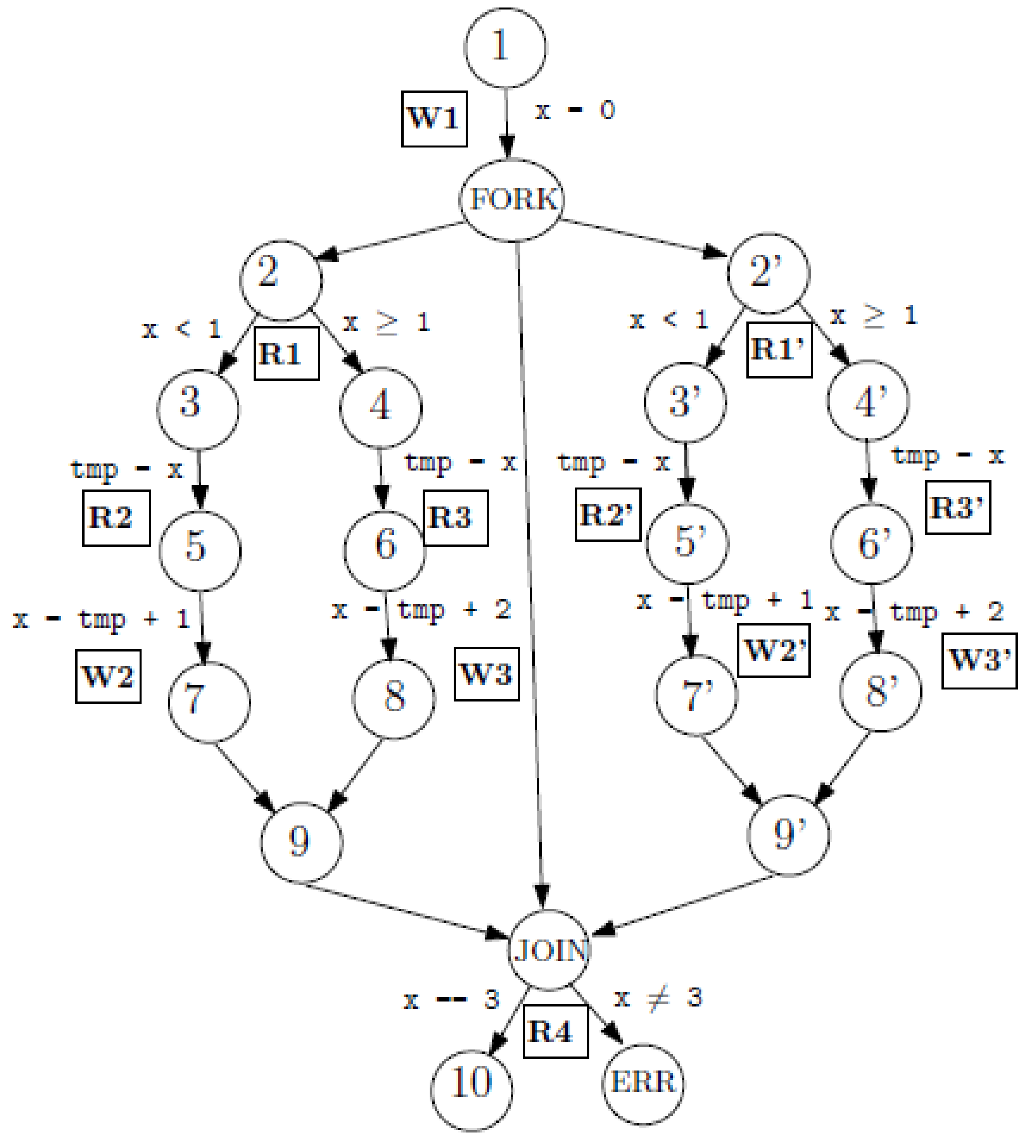
Program Representation

- Concurrent Control Flow Graph (CCFG)
  - Extension of sequential CFGs
  - Thread Fork, Join nodes
  - Functions modeled with call/return edges
  - Locks/Synchronization as shared variables
    - guarded assignments to model test-and-set

- Memory modeling
  - Compute shared location accesses using *flow-insensitive pointer analysis*
  - *Global* heap array + *Local* heap for each thread
  - Transform statements
    - one global access per statement
    - *p = l ⇝ MemG[p] = l; (if p accesses a shared location)

Example

```
int x;
void add_global ()
{
if ( x < 1 ) x = x + 1;
else x = x + 2;
}

int main (int argc, char *argv[])
{
        pthread_t t1, t2;
        x = 0;
        pthread_create(&t1, NULL,
        NULL,add_global);
        pthread_create(&t2, NULL,
        NULL, add_global);

        pthread_join(t1);
        pthread_join(t2);
        assert(x == 3);
}
```



Avoid Bi-modal Reasoning

- Obvious idea: Summarize each thread first!
- But, summarize in presence of concurrency?

```
int x; //global
int func (int a) {
   if (a) return x;
  else return x + 1;
}
```

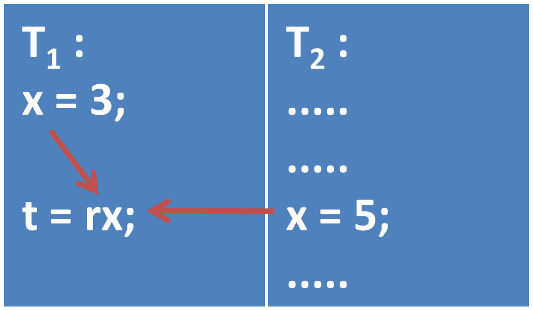
||

```
int func2 () {
   x = 3;
  .....
}
```

ret -> ite (a0, x0, x0+1) ?

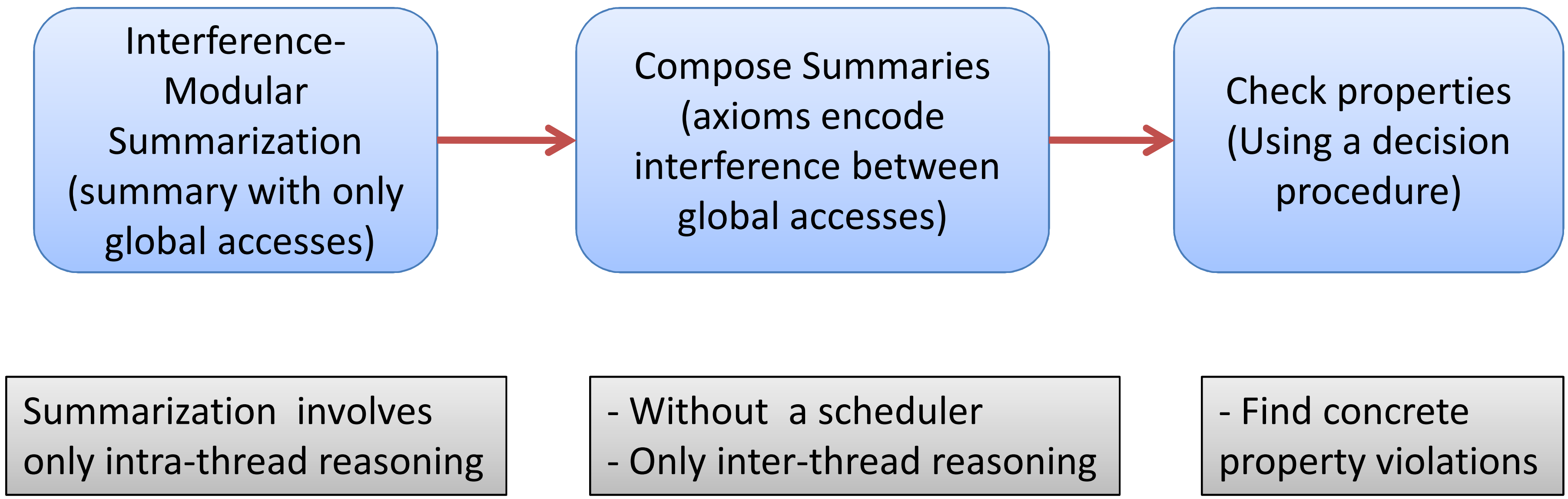
Interference Abstraction

- Reading a shared location **x** may not correspond to last write to **x** in the same thread
  - *interfering* concurrent write to **x**
- Idea: **Interference Abstraction**
  - introduce a symbolic variable for each read
  - *decouple* reads and writes
  - couple them *later*



- Contrast with **state abstraction** at a program point by duplicating shared variables
  - e.g., translation to sequential program under context bounds
  - num of shared accesses $\times$ num of shared vars
  - Interference Abstraction: linear in the number of reads

Staged Concurrent Program Analysis



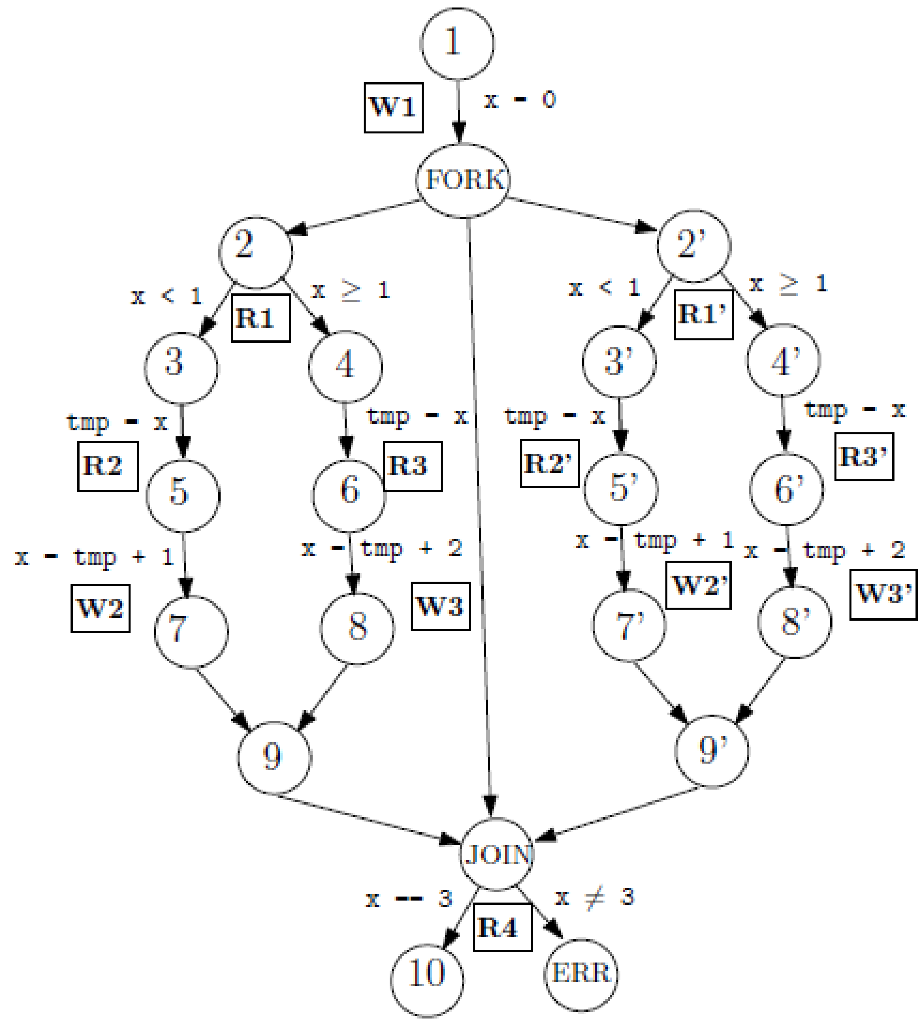
Stage 1: Summarization

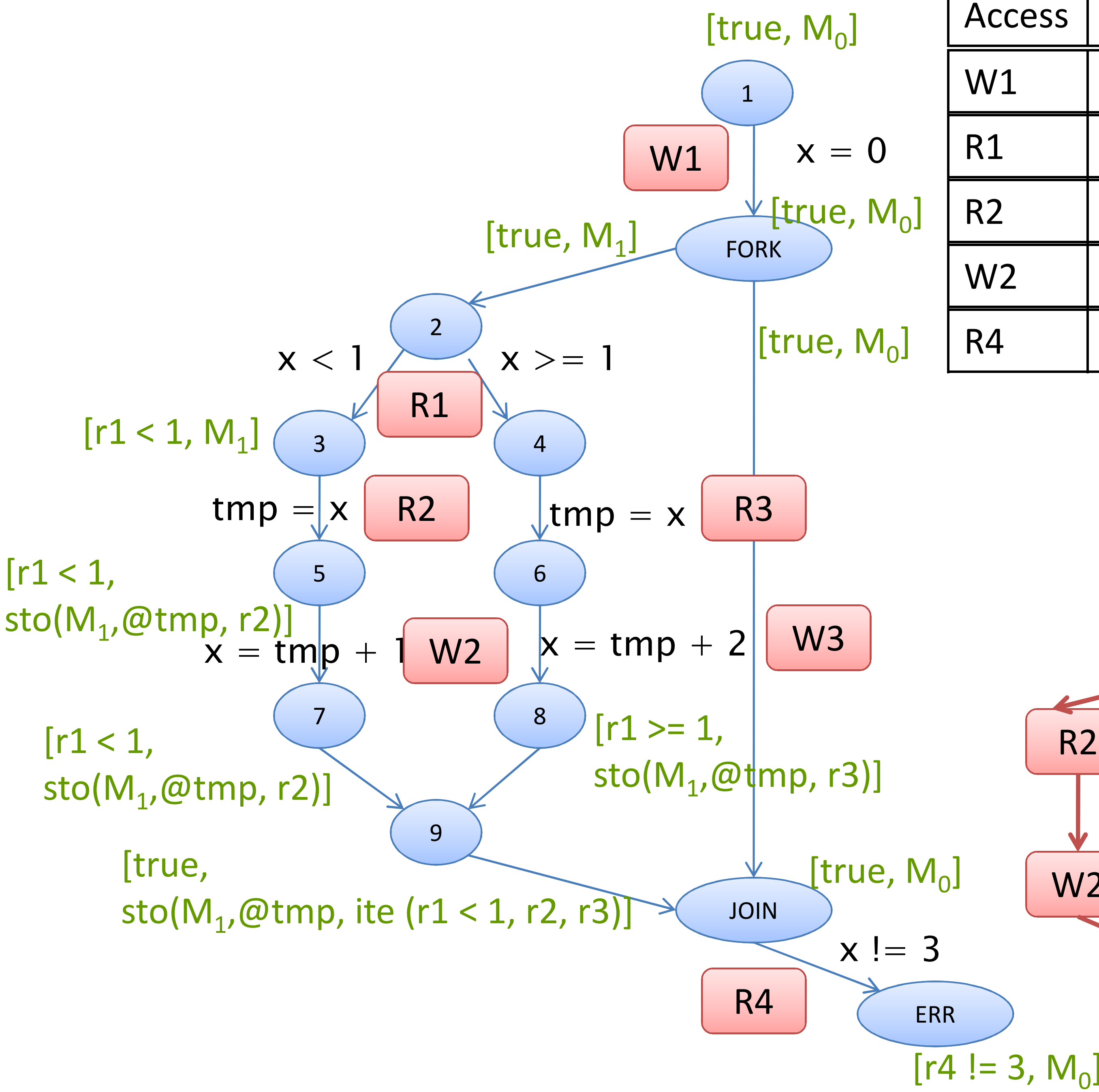
Stage 1: Summarization

- Interference-Modular Summarization
  - do away **precisely** with local control and data flow
  - keep the reads and writes of shared variables intact
- Why?
  - avoid bi-modal reasoning
  - because **only global accesses matter** for inter-thread reasoning
- How?
  - Data flow analysis **modulo** Interference Abstraction

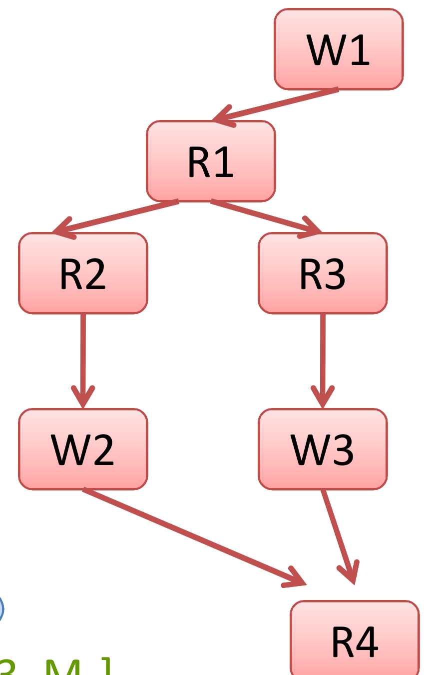
Example

```
int x;
void add_global ()
{
if ( x < 1 ) x = x + 1;
else x = x + 2;
}

int main (int argc, char *argv[])
{
        pthread_t t1, t2;
        x = 0;
        pthread_create(&t1, NULL,
        NULL,add_global);
        pthread_create(&t2, NULL,
        NULL, add_global);

        pthread_join(t1);
        pthread_join(t2);
        assert(x == 3);
}
```

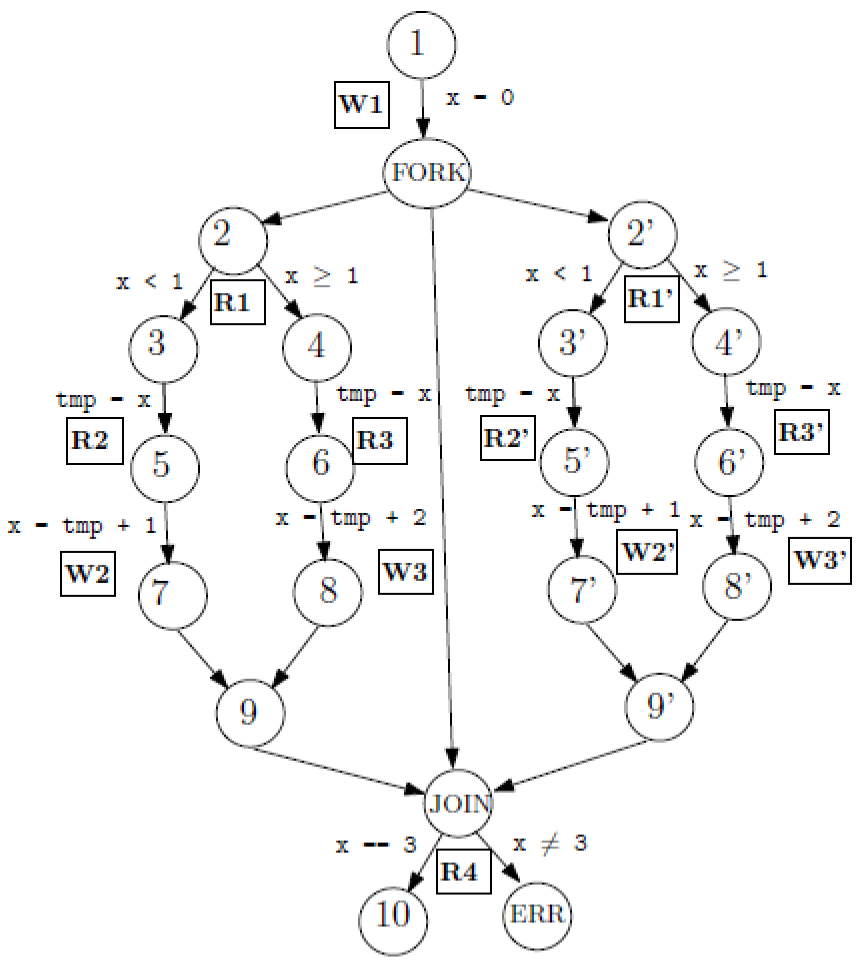




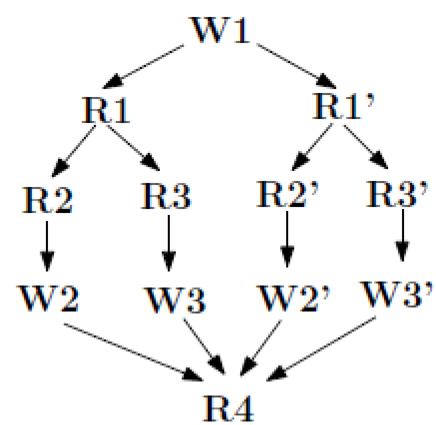
Access	loc	val	occ
W1	@x	0	true
R1	@x	r1	true
R2	@x	r2	r1 < 1
W2	@x	r2 + 1	r1 < 1
R4	@x	r4	true



Example: Summary



Access	loc	val	occ
$W1$	$@x$	0	$true$
$R1$	$@x$	r_1	$true$
$R2$	$@x$	r_2	$r_1 < 1$
$W2$	$@x$	$r_2 + 1$	$r_1 < 1$
$R3$	$@x$	r_3	$r_2 \geq 1$
$W3$	$@x$	$r_3 + 2$	$r_2 \geq 1$
$R4$	$@x$	r_4	$true$



Interference Skeleton (IS)

Summarization Rules

[Ψ, M, E]

↓ MemG[l] = r;

[Ψ, M, A]

l' = eval (l, M), v = eval (r, M)
Fresh access A = (Ψ, l', v)
Add E -> A to Skeleton

$[\Psi_1, M_1, E_1]$ $[\Psi_2, M_2, E_2]$

$[\Psi_1 \vee \Psi_2 , ite(\Psi_1 , M_1, M_2), E_1 \cup E_2]$

Intra-Thread Join

- Extends to standard Sharir-Pnueli, RHS style interprocedural analysis
  - Function Summarization and Reuse

Stage 2: Axiomatic Composition

Stage 2: Axiomatic Composition

- Interference Skeleton -> Feasible Program Executions?
  - need to couple the reads with writes
  - not via a scheduler!

- Idea: Compose Axiomatically
  - Axioms of Sequential Consistency (SC)
    - each read **must link** with **some write**
    - read must link with **last such write** in execution order
  - SC predominantly employed for straight line programs
    - how do we generalize to programs with branching?

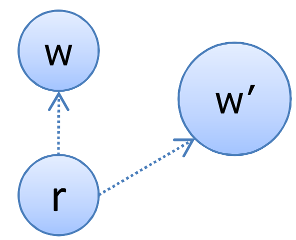
Sequential Consistency Axioms

- Specified in **typed first-order logic**
  - read **r**, write **w**: Access type
- **Link** Predicate: **link (r,w)**
  - holds if **r** obtains value from write **w** in an execution
  - Exclusive : link (r,w) => ∀ w’. ¬ link (r,w’)
- **Must-Happen-before** Predicate : **hb (w,r)**
  - **w** must happen before **r** in the execution
  - strict partial order

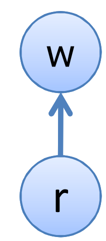
SC Axioms (contd.)

- $\Pi = \Pi_1 \wedge \Pi_2 \wedge \Pi_3$
- Π_1 (must link some, only if occurs)
  - $\forall$ r. occ(r) $\Leftrightarrow \exists$ w. occ(w) $\wedge$ link (r,w)

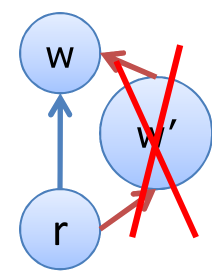
*Incorporate **occ** predicate to handle branching*



- Π_2 (local consistency)
  - $\forall$ r, w. link(r,w) $\Rightarrow$
    (loc(r) = loc (w) $\wedge$ val(r) = val(w) $\wedge$ hb(w,r))

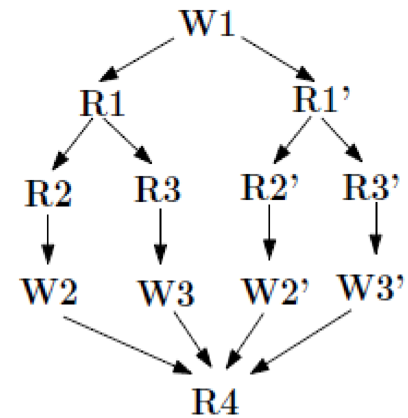


- Π_3 (global consistency)
  - $\forall$ r, w. link (r,w) $\Rightarrow$
    $\forall$ w'. (occ(w') $\wedge$ hbet(w, w',r)) $\Rightarrow$ loc(w) $\neq$ loc(w')



Instantiating Axioms

- Explicit instantiation for all reads and writes



- Π_1 := occ(r2) ⇔ (occ(w1) ∧ link (r2, w1)
  ∨ occ(w2') ∧ link (r2, w2') ...
- Π_2 := link (r2, w2') ⇒ loc(r2) = loc(w2') ∧ val(r2) = val(w2')
  ∧ hb (w2', r2)
- Π_3 := link (r2, w2') ⇒
  hbet (w2', w2, r2) ∧ occ(w2') ⇒ (loc(w2') ≠ loc(r2))

- At most **cubic** in number of reads and writes

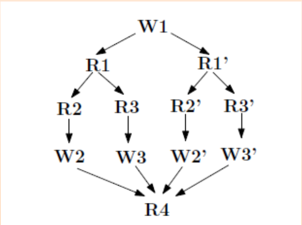
Efficient Encoding

- Employ UFs over theory of integers
  - avoid quantified axioms for **link** and **hb**
- Link Predicate:
  - **link (r,w) $\Leftrightarrow$ ID (r) = ID (w)**
  - assign unique IDs to all writes
- Must Happen-Before Predicate
  - **hb (w,r) $\Leftrightarrow$ Clk (w) < Clk (r)**
- Interference Pruning (few slides later)

Finding Bugs

Stage 3: Finding Bugs

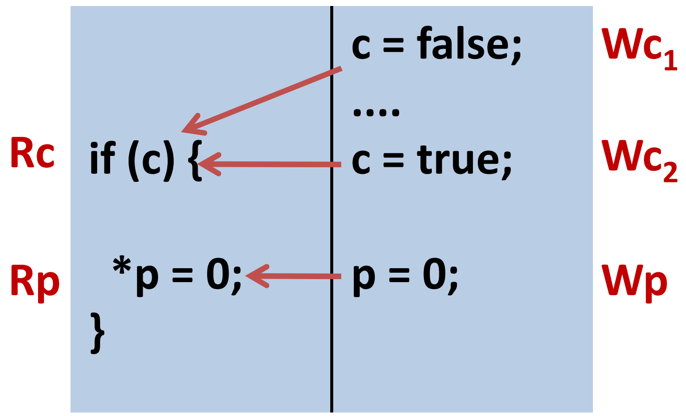
- Data races, say between r, w
  - $\Phi_P := \neg\, hb(r,w) \wedge \neg\, hb(w,r)$
- Assertion Violation
  - Φ_P := path condition for violation
- Full Encoding
  - $\Phi := \Phi_{IS} \wedge \Pi \wedge \Phi_P$
  - Discharged to an SMT solver



$\wedge \;\; \Pi \;\; \wedge \;\; r_4 \neq 3$

- **Theorem:** Φ is satisfiable iff property violated in the bounded program

Example



Goal: Detect NULL pointer access violation
- suppose the solver links Rp with Wp (Π_1 , Π_2)
- and, both occ(Rp) and occ(Wp) hold (Π_1)

occ(Rp) $\Rightarrow$ occ (Rc)
also, occ(Rp) $\Rightarrow$ val (Rc) = true (Φ_{IS})

link (Rc, Wc_1) $\vee$ link (Rc, Wc_2) (Π_1)

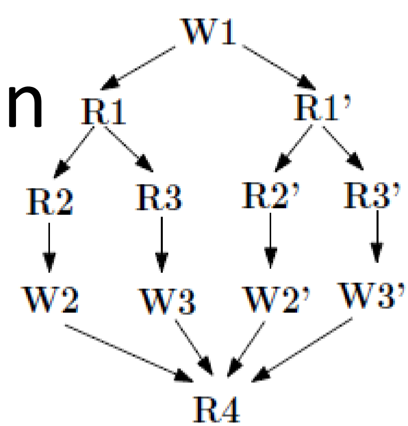
link (Rc, Wc_1) leads to **conflict** (Π_2)

so, link (Rc, Wc_2) and link (Rp, Wp)
so, hb (Wc_2, Rc) and hb (Wp, Rp) (Π_2)

linearize to obtain a feasible trace

Interference Pruning

- Π may have many redundant instantiations
  - Many r-w interferences are infeasible
  - Π_1: ¬ link(r,w) holds (w' occurs after w, before r in all runs)
  - Π_2: ¬ hb(w,r) holds (w occurs after r in all runs)
  - Π_3: ¬ hbet(w,w',r) (for some w, w', r)
- Static analysis of Interference Skeleton
  - Prune away infeasible r-w interferences



Implementation

- FUSION framework for analyzing concurrent programs
  - combines dynamic and symbolic analysis
  - used to obtain (bounded) program slices
- Yices SMT solver
- Compared with/without summarization (S), pruning optimization (O)

Experiments

Bm (#Thr)	|N|	|E|	|R|	|W|	-S (FSE'09)	+S
SB(2)	108	107	6	19	1	**1**
SB(3)	723	722	270	289	9	**3**
Ind (20)	1312	1439	110	291	0.1	**0.1**
Ind (29)	2446	2691	360	887	129	**6**
Ind (30)	2859	3149	468	1104	517	**7**
Ind (31)	3398	3747	594	1332	>1800	**13**
Ind (32)	4585	5065	888	1856	>1800	**104**
acc (11)	906	905	134	372	1	**1**
acc (21)	1748	1747	708	25	>1800	**10**

Experiments

Bm (#Thr)	|N|	|E|	|R|	|W|	+S-O	+S+O
SB(2)	108	107	6	19	1	**1**
SB(3)	723	722	270	289	711	**3**
Ind (20)	1312	1439	110	291	355	**0.1**
Ind (29)	2446	2691	360	887	>1800	**6**
Ind (30)	2859	3149	468	1104	>1800	**7**
Ind (31)	3398	3747	594	1332	>1800	**13**
Ind (32)	4585	5065	888	1856	>1800	**104**
acc (11)	906	905	134	372	121	**1**
acc (21)	1748	1747	708	25	>1800	**10**

Conclusions

- **Avoiding Bi-modal** reasoning leads to significant (possibly exponential) speedups
- **Sequential Consistency** (SC) axioms to compose shared memory programs
  - model interference directly
  - avoid scheduler
- Future work: Automated axiom instantiations

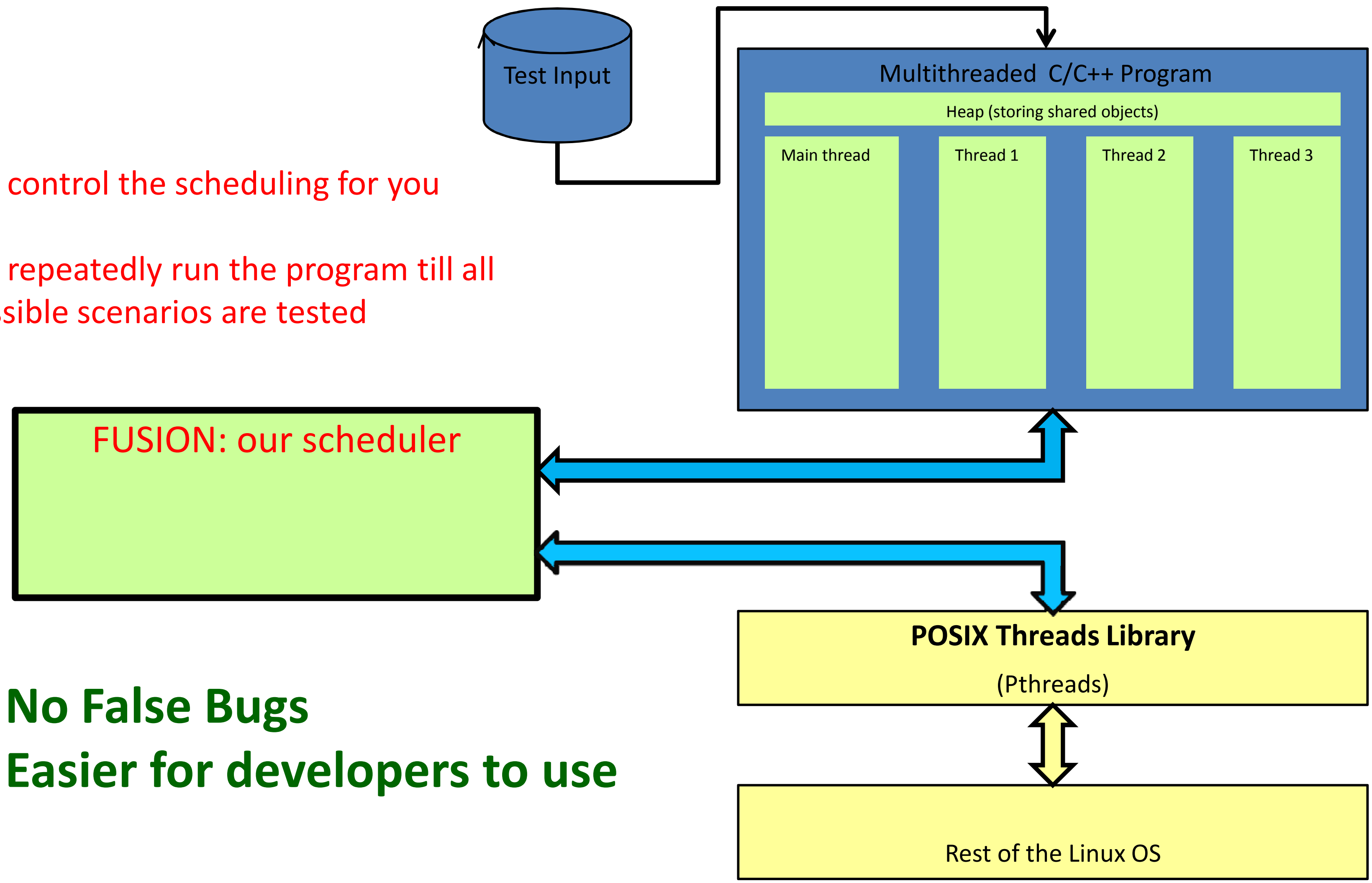
Thanks !

Questions?

FUSION framework

We control the scheduling for you

We repeatedly run the program till all possible scenarios are tested



- ✓ **No False Bugs**
- ✓ **Easier for developers to use**