CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models

Sebastian Burckhardt
Rajeev Alur
Milo M. K. Martin

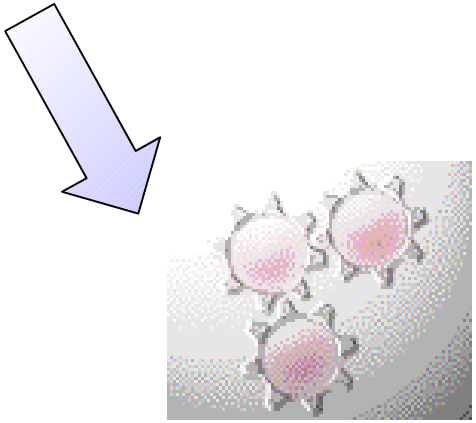
Department of Computer and Information Science
University of Pennsylvania

June 6, 2007

General Motivation

Multi-threaded Software

Shared-memory Multiprocessor



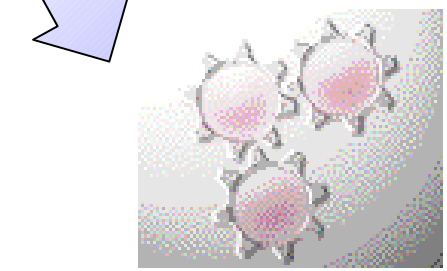
Concurrent Executions

Bugs

Specific Motivation

Multi-threaded Software
lock-free synchronization
(intentional races)

Shared-memory Multiprocessor
with relaxed memory model



Concurrent Executions
not sequentially consistent:
processors may
- buffer stores locally
- load from local buffer
- reorder loads
- reorder stores

Bugs

Specific Motivation

concurrency libaries with lock-free synchronization

... are simple, fast, and safe to use

- concurrent versions of queues, sets, maps, etc.
- more concurrency, less waiting
- fewer deadlocks

... are notoriously hard to design and verify

- tricky interleavings often escape reasoning and testing
- exposed to relaxed memory models
  on most multiprocessors, implementations do not work correctly unless appropriate memory fences are inserted

Specific Motivation

concurrency libaries with lock-free synchronization

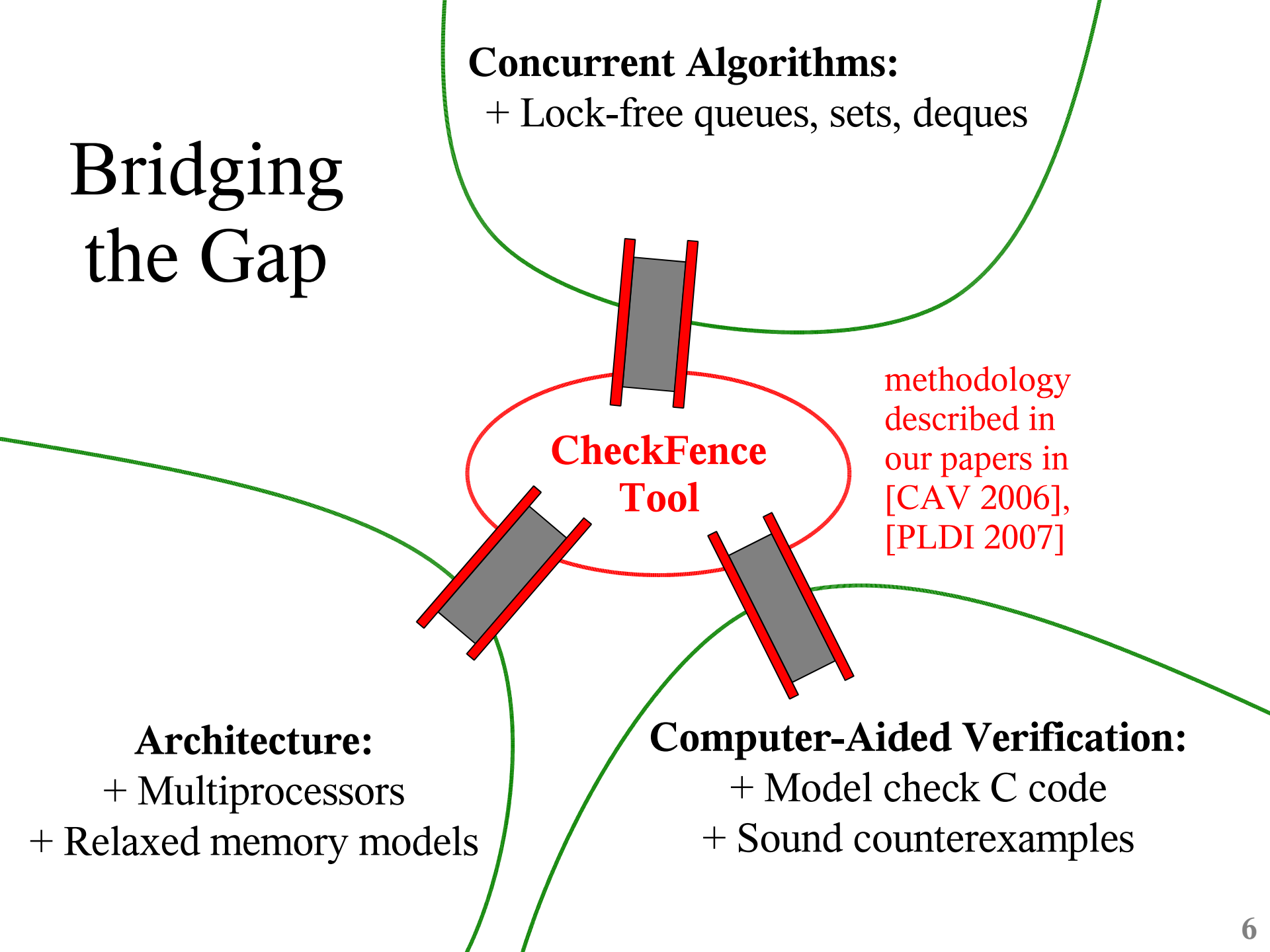
... are simple, fast, and safe to use

CLIENT PROGRAMS MAY BE LARGE
thousands, millions lines of code

... are notoriously hard to design and verify

DATA TYPE IMPLEMENTATIONS ARE TINY
tens to hundreds lines of code

Bridging the Gap



Example: Nonblocking Queue

Processor 1

```
....
... enqueue(1)
... enqueue(2)
....
....
   ....
```

Processor 2

```
....
...
...
a = dequeue()
b = dequeue()
```

The client program

- on multiple processors
- calls operations

```
void enqueue(int val) {
   ...
}

int dequeue() {
   ...
}
```

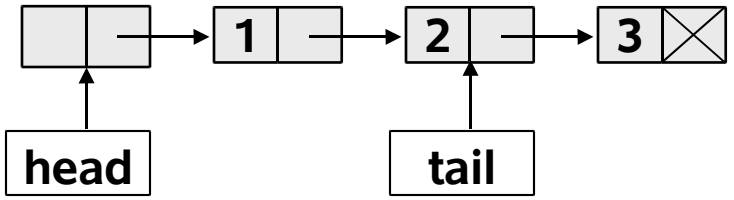
The implementation

- optimized: no locks.
- not race-free
- exposed to memory model

Michael & Scott's Nonblocking Queue

[Principles of Distributed Computing (PODC) 1996]

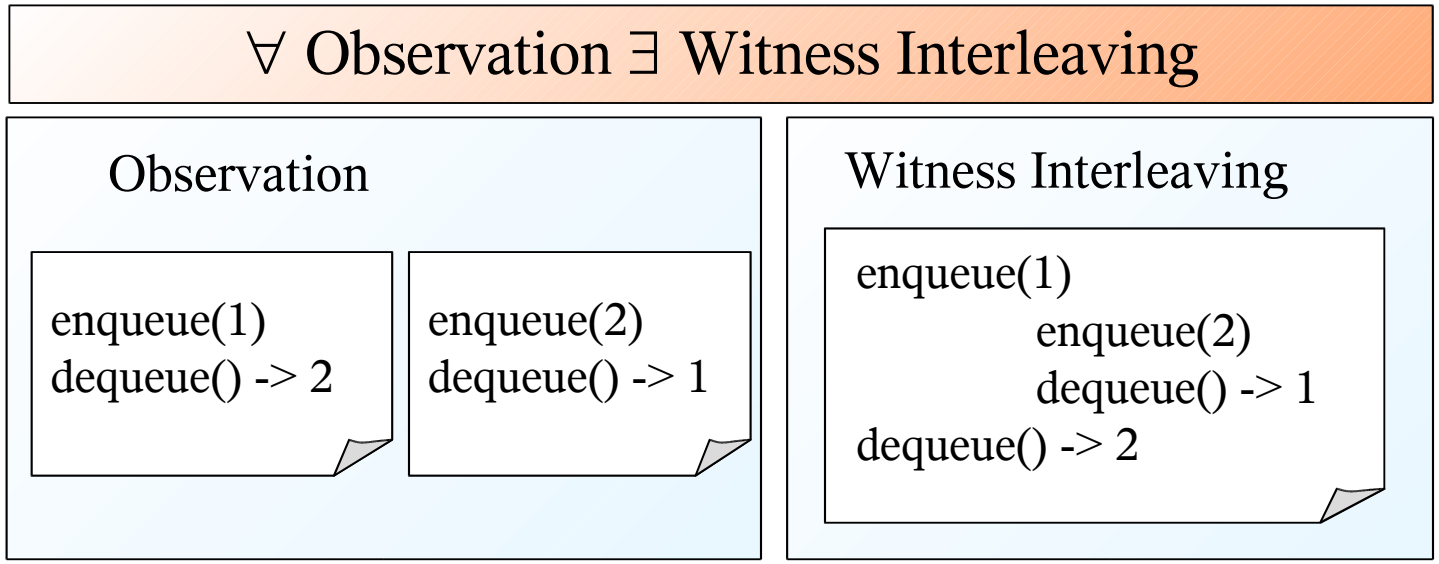
```
boolean_t dequeue(queue_t *queue, value_t *pvalue)
{
  node_t *head;
  node_t *tail;
  node_t *next;

  while (true) {
    head = queue->head;
    tail = queue->tail;
    next = head->next;
    if (head == queue->head) {
      if (head == tail) {
        if (next == 0)
          return false;
        cas(&queue->tail, (uint32) tail, (uint32) next);
      } else {
        *pvalue = next->value;
        if (cas(&queue->head, (uint32) head, (uint32) next))
          break;
      }
    }
  }
  delete_node(head);
  return true;
}
```



Correctness Condition

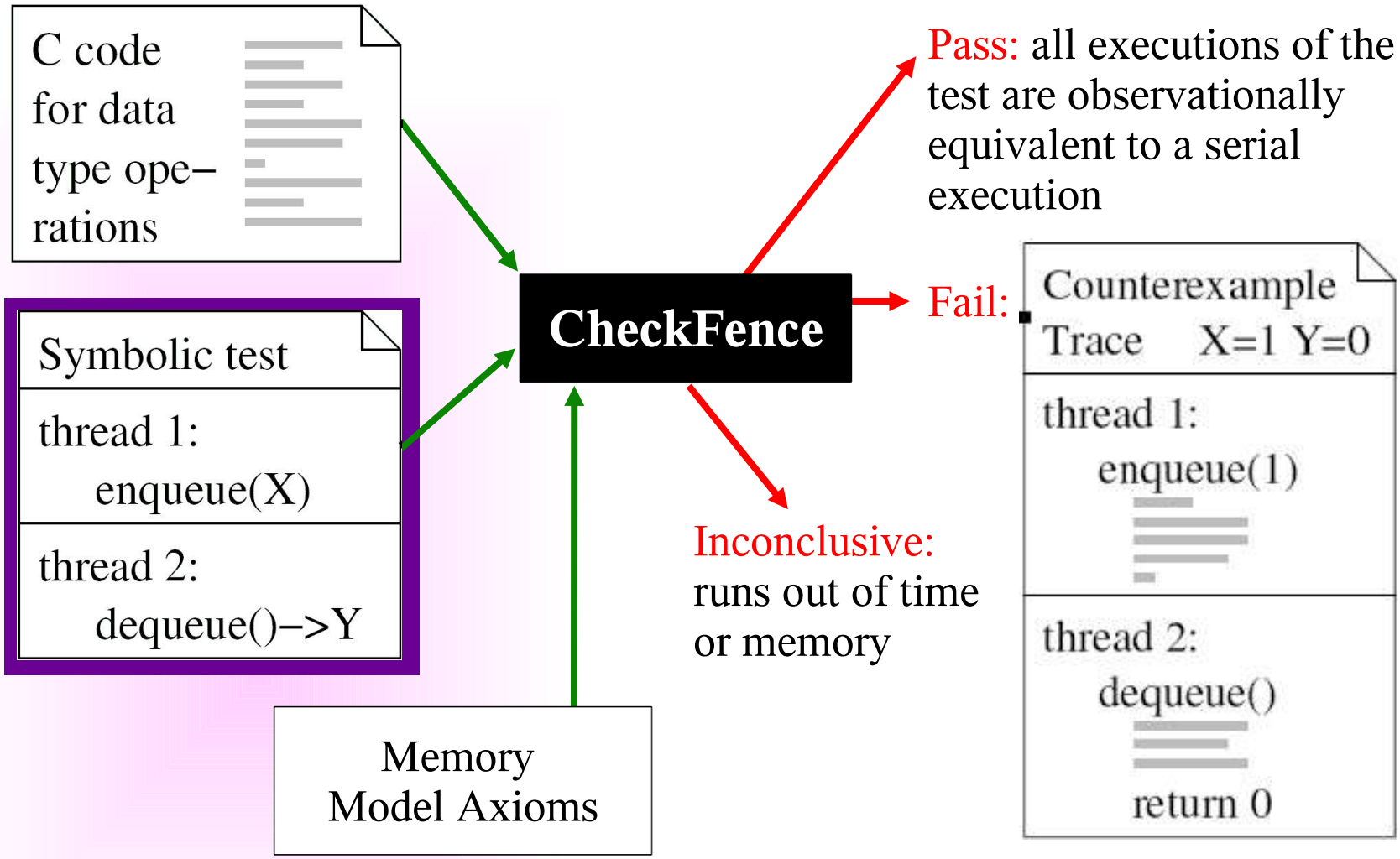
Data type implementations must appear sequentially consistent to the client program:

the observed argument and return values must be consistent with some interleaved, atomic execution of the operations.



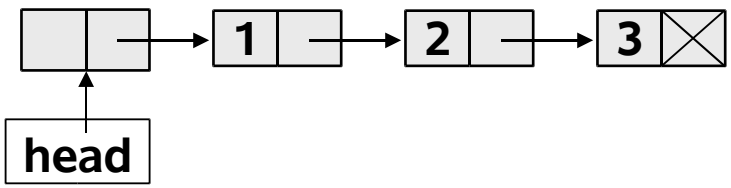
Part II: Solution

Bounded Model Checker



Demo: CheckFence Tool

Example: Memory Model Bug



Processor 1
links new node into list

Processor 2
reads value at head of list

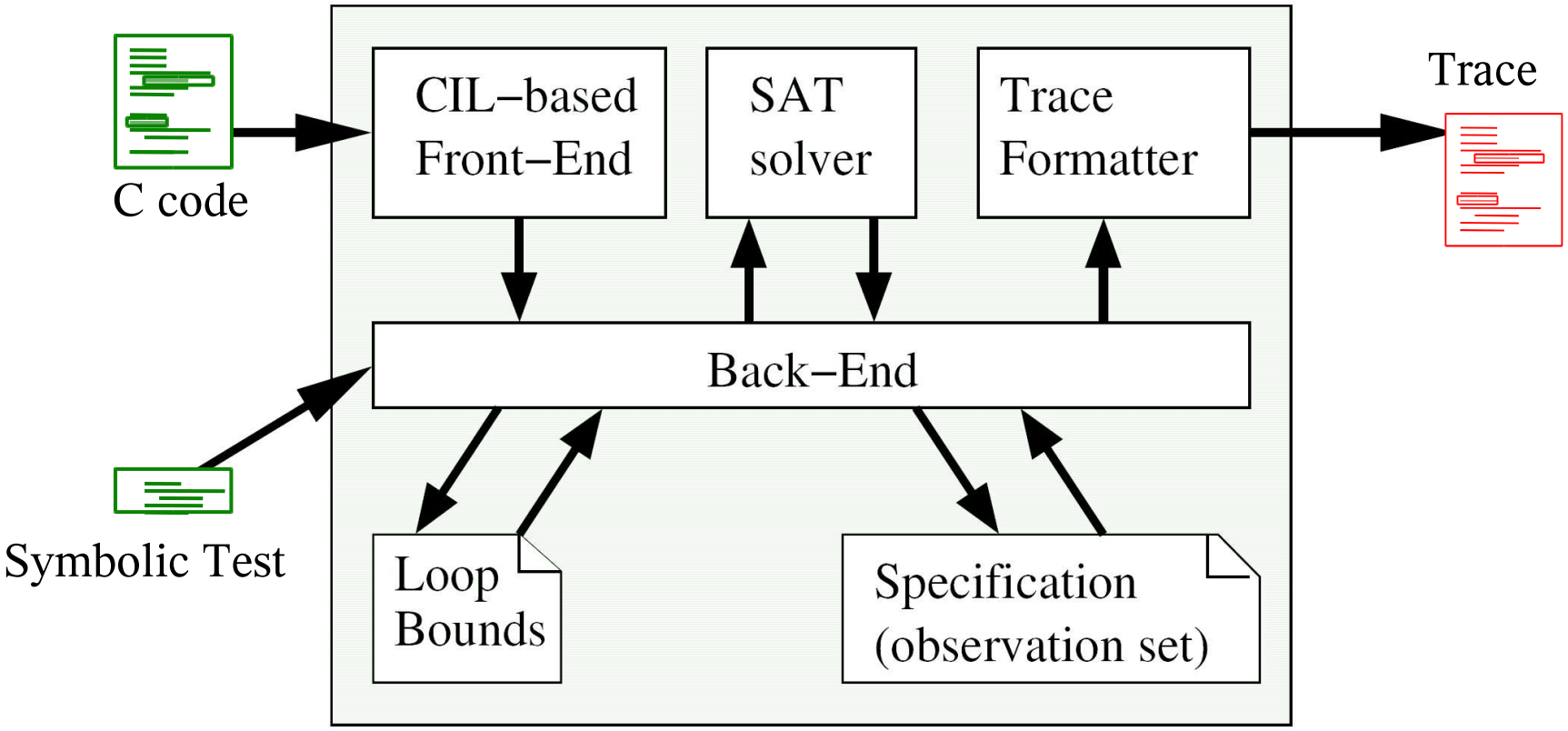
```
   ...                          ...
3  node->value = 2;          2 value = head->value;
   ...                          ...
1  head = node;
   ...
```

Processor 1 reorders the stores!
memory accesses happen in order **1 2 3**

--> Processor 2 loads uninitialized value

adding a *fence* between lines on left side prevents reordering.

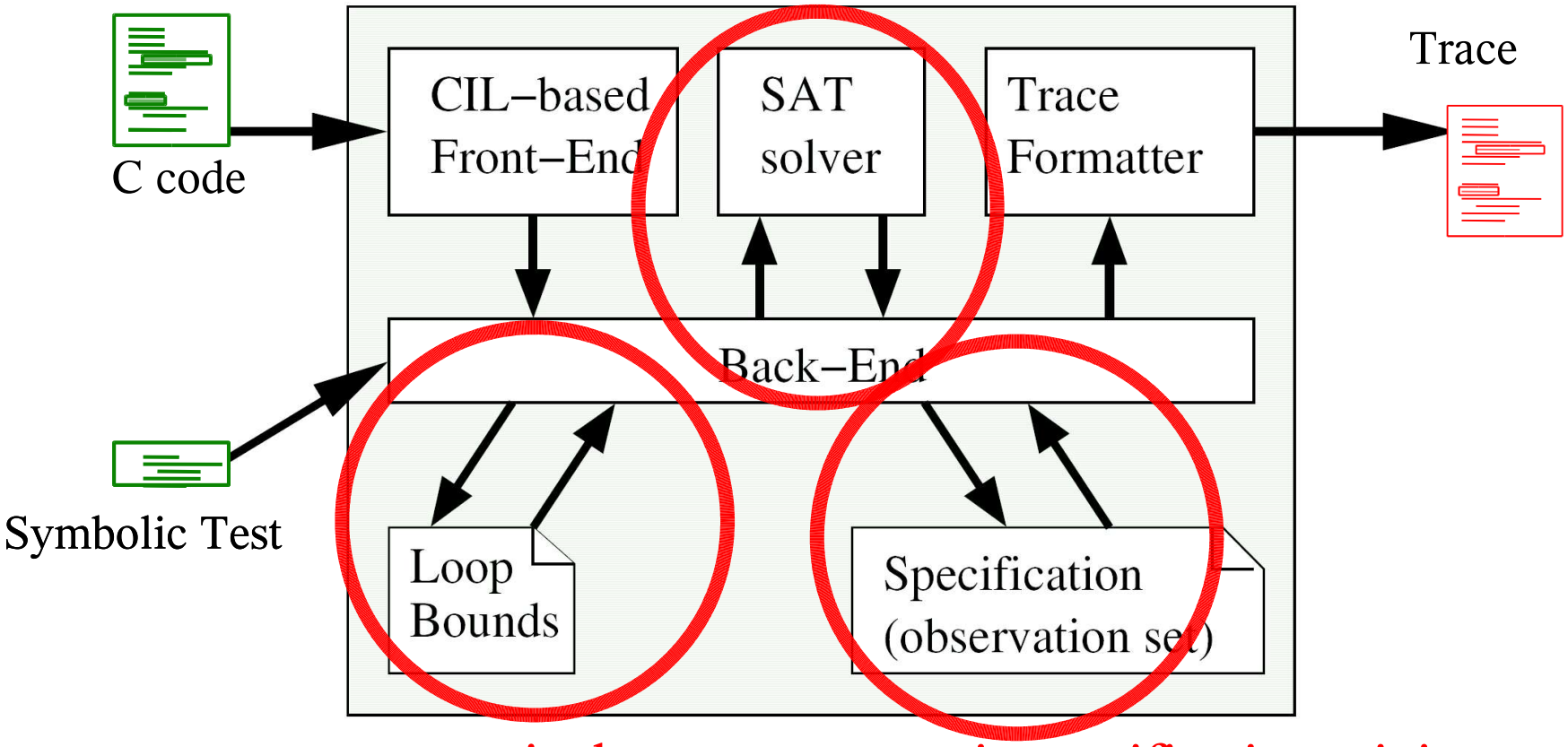
Tool Architecture



Symbolic test is nondeterministic, has exponentially many executions
(due to symbolic inputs, dyn. memory allocation, interleaving/reordering of instructions).

CheckFence solves for "bad" executions.

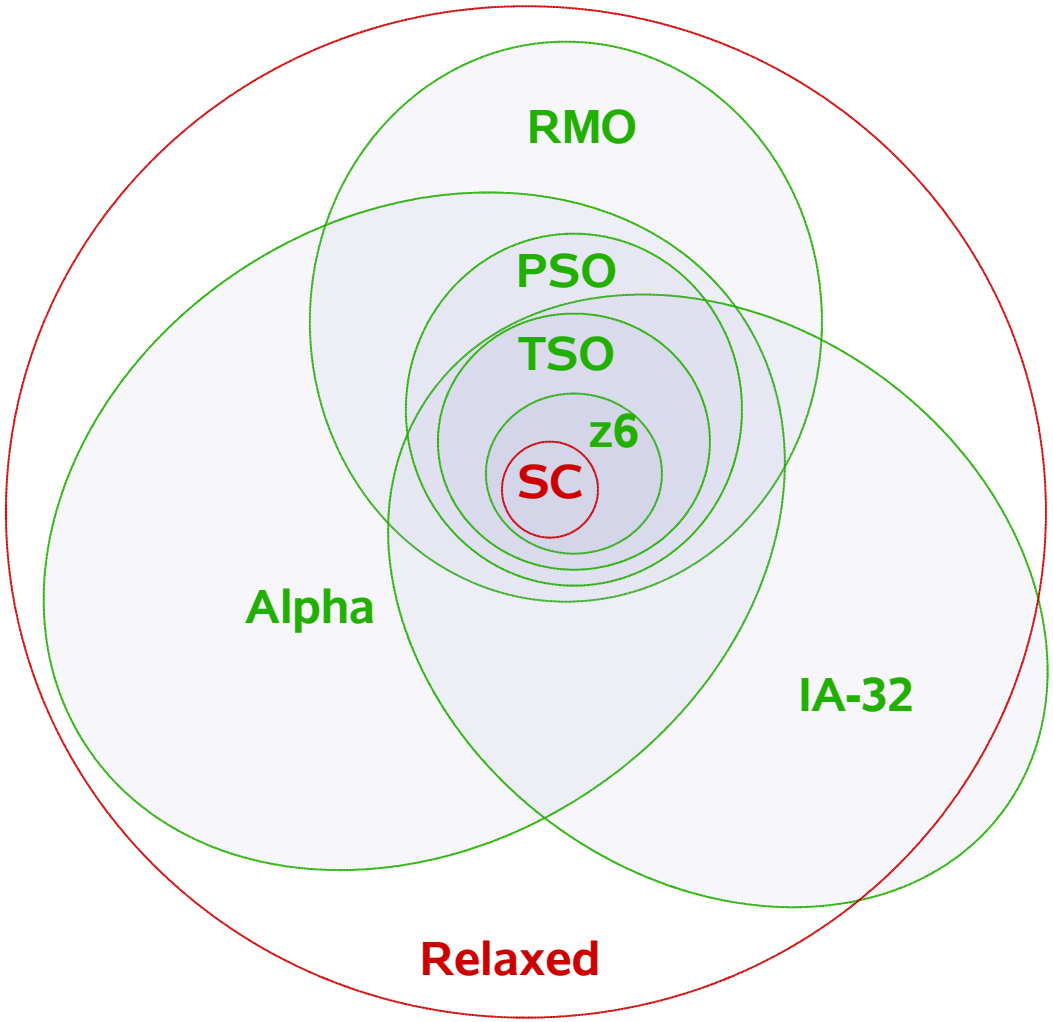
construct CNF formula whose solutions
correspond precisely to the concurrent executions



automatic, lazy
loop unrolling

automatic specification mining
(enumerate correct observations)

Which Memory Model?



- Memory models are platform dependent & ridden with details
- We use a conservative abstract approximation "*Relaxed*" to capture common effects
- Once code is correct for *Relaxed*, it is correct for stronger models
- Finding simple, general abstraction is hard (work in progress)

Part VI: Results

Studied Algorithms

Type	**Description**	**loc**	**Source**
Queue	Two-lock queue	80	Michael and Scott (*PODC* 1996)
Queue	Non-blocking queue	98	
Set	Lazy list-based set	141	Heller et al. (*OPODIS 2005)*
Set	Nonblocking list	174	Harris (*DISC* 2001)
Deque	“snark” algorithm	159	Detlefs et al. (*DISC* 2000)
LL/VL/SC	CAS-based	74	Moir (*PODC* 1997)
LL/VL/SC	Bounded Tags	198	

Results

- snark algorithm has 2 known bugs, we found them
- lazy list-based set had a unknown bug
  (missing initialization; missed by formal correctness proof [CAV 2006] because of hand-translation of pseudocode)

Type	**Description**	**regular bugs**
Queue	Two-lock queue	
Queue	Non-blocking queue	
Set	Lazy list-based set	1 unknown
Set	Nonblocking list	
Deque	original “snark”	2 known
Deque	fixed “snark”	
LL/VL/SC	CAS-based	
LL/VL/SC	Bounded Tags	

Results

- snark algorithm has 2 known bugs, we found them
- lazy list-based set had a unknown bug
  (missing initialization; missed by formal correctness proof [CAV 2006] because of hand-translation of pseudocode)
- Many failures on relaxed memory model
  - inserted fences by hand to fix them
  - small testcases sufficient for this purpose

Type	Description	regular bugs	# Fences inserted			
			Store Store	Load Load	Dependent Loads	Aliased Loads
Queue	Two-lock queue		1		1	
Queue	Non-blocking queue		2	4	1	2
Set	Lazy list-based set	1 unknown	1		3	
Set	Nonblocking list		1		2	3
Deque	original "snark"	2 known				
Deque	fixed "snark"		4	2	4	6
LL/VL/SC	CAS-based					3
LL/VL/SC	Bounded Tags					4

Typical Tool Performance

- Very efficient on small testcases (< 100 memory accesses)
  Example (nonblocking queue): T0 = i (e | d) T1 = i (e | e | d | d)
  - find counterexamples within a few seconds
  - verify within a few minutes
  - enough to cover all 9 fences in nonblocking queue
- Slows down with increasing number of memory accesses in test
  Example (snark deque):
  Dq = (pop_l | pop_l | pop_r | pop_r | push_l | push_l | push_r | push_r)
  - has 134 memory accesses (77 loads, 57 stores)
  - Dq finds second snark bug within ~1 hour
- Does not scale past ~300 memory accesses

Related Work

Bounded Software Model Checking

Clarke, Kroening, Lerda (*TACAS'04)* Rabinovitz, Grumberg (CAV'05)

Correctness Conditions for Concurrent Data Types

Herlihy, Wing (TOPLAS'90) Alur, McMillan, Peled (LICS'96)

Operational Memory Models & Explicit Model Checking

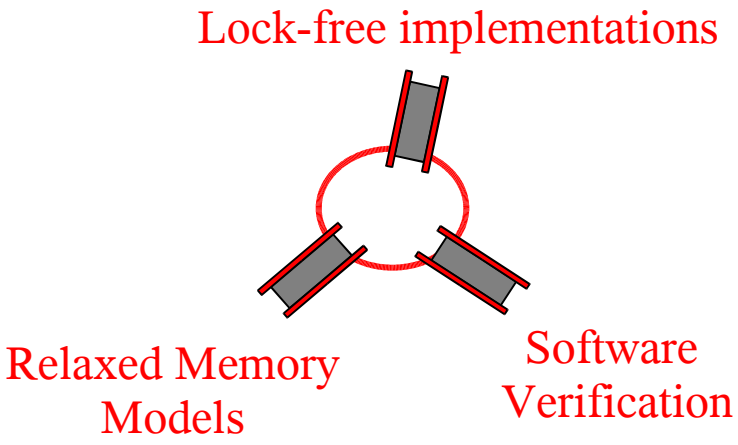
Park, Dill (SPAA'95) Huynh, Roychoudhury (FM'06)

Axiomatic Memory Models & SAT solvers

Yang, Gopalakrishnan, Lindstrom, Slind (IPDPS'04)

Contribution

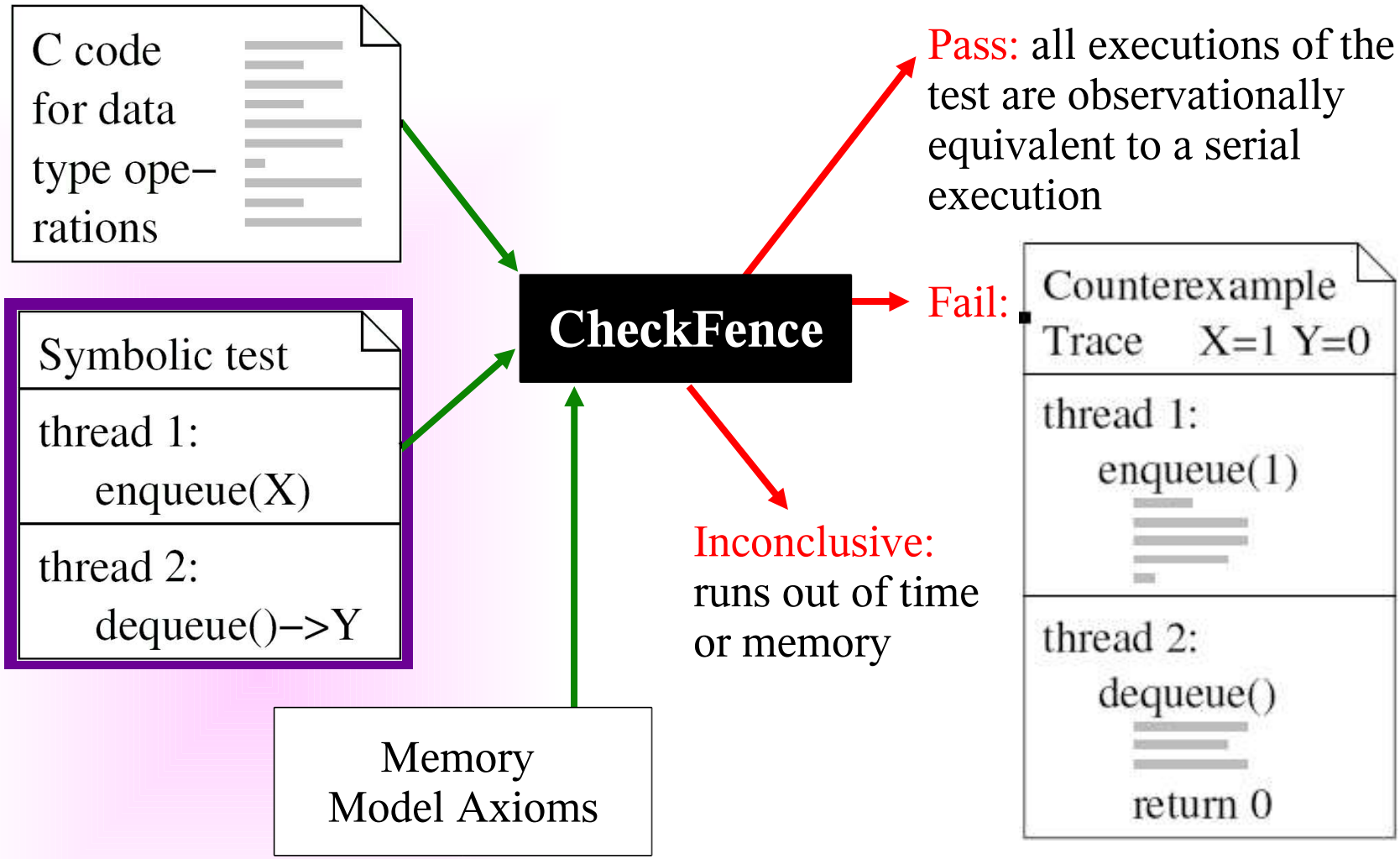
First model checker for C code on relaxed memory models.



- Handles ``reasonable'' subset of C
  (conditionals, loops, pointers, arrays, structures, function calls, dynamic memory allocation)
- No formal specifications or annotations required
- Requires manually written test suite
- Soundly verifies & falsifies individual tests, produces counterexamples


The End

Bounded Model Checker



Future Work

- Make CheckFence publicly available
- Experiment with more memory models
  - hardware (PPC, Itanium), language (Java, C++ volatiles)
- Improve solver component
  - enhance SAT solver support for total/partial orders
- Develop reasoning techniques for relaxed memory models
- Develop scalable methods for finding specific, common bugs
- Build concurrent library

Axioms for *Relaxed*

A set of addresses V set of values

X set of memory accesses $S \subseteq X$ subset of stores $L \subseteq X$ subset of loads

a(x) memory address of x v(x) value loaded or stored by x

$<_p$ is a partial order over X (program order)

$<_m$ is a total order over X (memory order)

For a load $l \in L$, define the following set of stores that are "visible to l":

$S(l) = \{ s \in S \mid a(s) = a(l) \text{ and } (s <_m l \text{ or } s <_p l) \}$

Executions for the model *Relaxed* are defined by the following axioms:

1. If $x <_p y$ and $a(x) = a(y)$ and $y \in S$, then $x <_m y$
2. For $l \in L$ and $s \in S(l)$, always either $v(l) = v(s)$ or there exists another store $s' \in S(l)$ such that $s <_m s'$

Relaxed Memory Model Example

- Example:

thread 1	thread 2	
`x = 1`	`print y`	→ **2**
`y = 2`	`print x`	→ **0**

output not sequentially consistent
(that is, not consistent with any interleaved execution) !

  - processor 1 may perform stores out of order
  - processor 2 may perform loads out of order
  - relaxed ordering guarantees improve processor performance

- **Q:** Why doesn't everything break?
  A: Relaxations are designed in a way to guarantee that
  - uniprocessor programs are safe
  - race-free programs are safe