An Efficient Implementation of Guard-Based Synchronization for an Object-Oriented Programming Language

Shucai Yao, Huawei Technologies Canada, Markham, Ontario*
Emil Sekerinski, McMaster University, Hamilton, Ontario

***work performed while affiliated with McMaster University**

PLACES, May 2025

Shared Variable Model of Concurrency: … || …

Conditional Critical Regions [Hoare 72] , e.g. bounded buffer

region B: **record** in, out, n: int; buf: **array** 0 .. C – 1 **of** T **end**
procedure deposit(val: T)
  with B **when** n < C **do**
    buf[in] := val; in := (in + 1) mod C; n := n + 1
procedure fetch() → (val: T)
  with B **when** count > 0 **do**
    val := buf[out]; out := (out + 1) mod N; n := n – 1

guard body atomic

Proof Techniques of [Owicki & Gries 76]

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$$

atomic

Shared Variable Model of Concurrency: … || …

Unity [Chandy & Misra 88] , e.g. sorting
  $\langle [] \ i : 0 \le i < N :: A[i], A[i+1] := A[i+1], A[i] \ \ \textbf{if} \ \ A[i] > A[i+1] \rangle$

Joint Action Systems [Back & Kurki-Suonio 89]
  $G \rightarrow S$

TLA [Lamport 94]
  $G \wedge x' = E$

Event-B [Abrial 10]
  event e = **when** G **then** Q **end**

CIVL [Siegel et. al 15]

…

➜ guards ubiquitous in specification languages

Implementing Guarded Commands

Common wisdom: synchronization by guarded commands cannot be implemented efficiently

"The price that must be paid for this automatic scheme is performance" [Briot, Guerraoui, Lohr 98]

"Conditional critical regions … expensive to implement … evaluation [of conditions] results in numerous context switches … many of which may be unproductive because … because the condition [may be] false" [Andrews, Schneider 83]

Semaphores [Dijkstra 62, 67]
  $sem.P \ = \ \langle \ sem > 0 \ \rightarrow \ sem := sem - 1 \ \rangle$
  $sem.V \ = \ \langle \ sem := sem + 1 \ \rangle$

Monitors [Hoare 74]
  monitor *Buffer*
    var *in, out, n: int; buf:* **array** 0 .. *C* – 1 **of** *T*
    var *nonfull, nonempty: condition*
    procedure *deposit(val: T)*
      while *n* = *C* **do** *nonfull.wait*
      buf[in] := val; in := (in + 1) **mod** *C; n := n* + 1
      nonempty.signal

Message Passing [Hoare 78, Occam, Go]
  var *deposit, fetch*: **channel of** *T*
  process *Buffer*
    var *in, out, n: int*
    var *buf*: **array** 0 .. C – 1 **of** *T*
    do *n* < *C* & *deposit* ? *buf[in]* → …
    [] *n* > 0 & *fetch* ! *buf[out]* → …

➜ an experiment in language restrictions and implementation techniques for atomic guarded commands

Lime: An Action-Based Object-Oriented Programming Language

```
class Buffer
  var in, out, n: int; buf: array 0 .. C – 1 of int
  method deposit(val: int)
    when this.n < C do
      this.buf[this.in] := val; this.in := (this.in + 1) mod C; this.n := this.n + 1

var b: Buffer

class Producer
  var p: int
  action produce
    when true do p := p + 1;  b.deposit(p)

p := new Producer()
```

synchronization through guards

all objects are concurrent, c.f. Smalltalk, actors

active objects with actions

atomic execution up to method calls

objects are units of concurrency

c.f. Seuss [Misra 01]

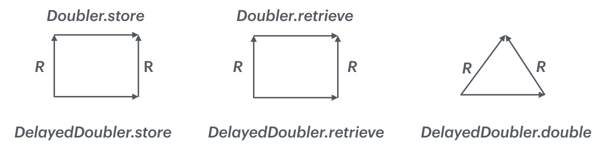
Delayed Doubler: Background Execution

```
class Doubler
  var x: int
  init()
    this.x := 0
  method store(u: int)
    this.x := 2 * u
  method retrieve(): int
    return this.x
```

invariant x mod 2 = 0

refines Doubler $R \equiv d \Rightarrow y = 2 * u$

```
class DelayedDoubler
  var y:int, d: bool
  init()
    this.y, this.d := 0, true
  method store(u: int)
    this.y, this.d := u, false
  method retrieve(): int
    when this.d do return this.y
  action double
    when not this.d do
      this.y, this.d := 2 * y, true
```

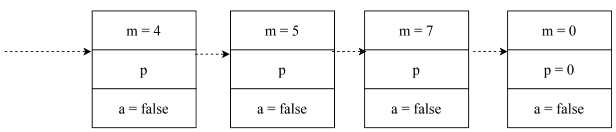


c.f. [Bonsangue et al 98]

Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```



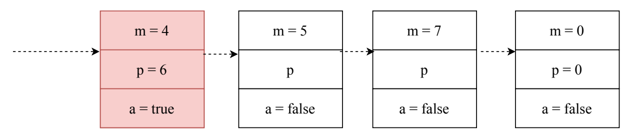

```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false

action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```

Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```



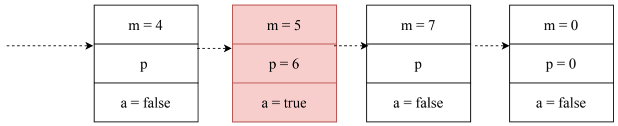

```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false

action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```

Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```



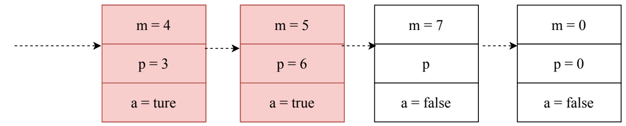

```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false
```

```
action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```


Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```

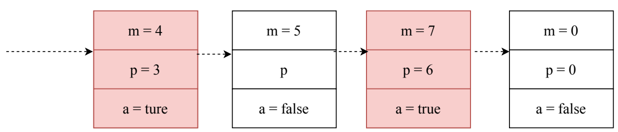

```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false
```

```
action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```

Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```

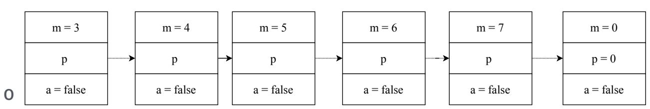

```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false
```

```
action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```

Priority Queue

communication & synchronization » computation

```
class PriorityQueue
  var m, p: int
  var l: PriorityQueue
  var a, r: bool
  init()
    l, a, r, m := nil, false, false, 0
  method empty() : bool
    when not r do
      return l = nil
  method add(e: int)
    when not a and not r do
      if l = nil then
        m, l := e, new PriorityQueue()
      else
        p, a := e, true
  method remove() : int
    when not a and not r do
      r := true
      return m
```

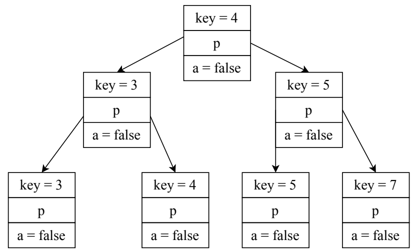




```
action doAdd
  when a do
    if m < p then
      l.add(p)
    else
      l.add(m)
      m := p
    a := false
```

```
action doRemove
  when r do
    if l = nil then
      r := false
      return
    elif l.empty() then
      l := nil
    else
      m := l.remove()
    r := false
```

Leaf-oriented tree

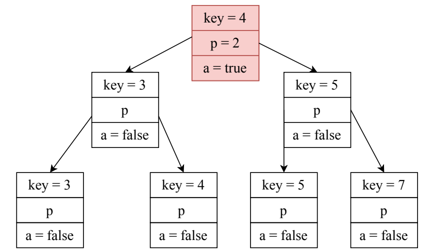

```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```




```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Leaf-oriented tree

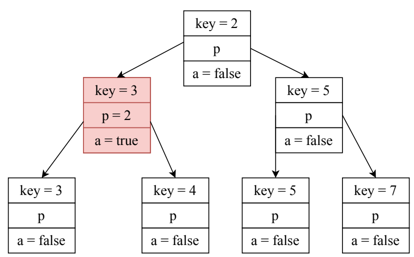

```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```




```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Leaf-oriented tree

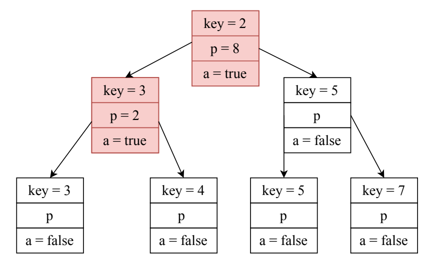

```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```




```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Leaf-oriented tree


```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```

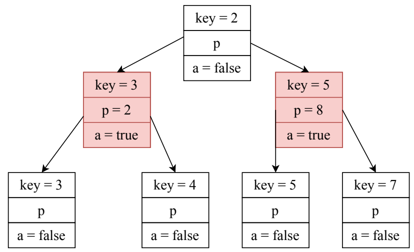



```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Leaf-oriented tree

communication & synchronization » computation

```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```

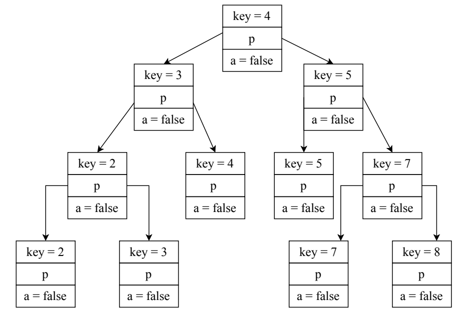



```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Leaf-oriented tree

communication & synchronization » computation

```
class Node
  var key, p: int
  var left, right: Node
  var a: bool
  init(x: int)
    key, left, right, a := x, nil, nil, false
  method add(x: int)
    when not a do
      if left ≠ nil then a, p := true, x
      elif x < key then
        left, right, key := new Node(x), new Node(key), x
      elif x > key then
        left, right := new Node(key), new Node(x)
  method has(x: int): bool
    when not a do
      if left = nil then return x = key
      elif x ≤ key then return left.has(x)
      else return right.has(x)
```



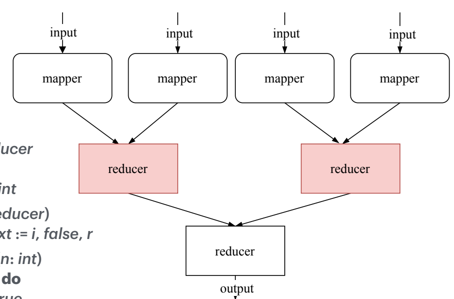

```
action addToChild
  when a do
    if p ≤ key then left.add(p)
    else right.add(p)
    a := false
```

Map-Reduce

communication & synchronization » computation

```
class Reducer
  var index: int
  var next: Reducer
  var a1, a2: bool
  var e1, e2: int
  init(i: int, r: Reducer)
    index, a1, a2, next := i, false, false, r
  method reduce1(x: int)
    when not a1 do
      e1, a1 := x, true
  method reduce2(x: int)
    when not a2 do
      e2, a2 := x, true
  action doReduce
    when a1 and a2 do
      if index = 1 then
        print(e1 + e2)
        e1, e2 := 0, 0
  elif index mod 2 = 0 then
    next.reduce1(e1 + e2)
  else
    next.reduce2(e1 + e2)
  a1, a2 := false, false
```

```
class Mapper
  var next: Reducer
  var a: bool
  var e, index: int
  init(i: int, r: Reducer)
    index, a, next := i, false, r
  method map(n: int)
    when not a do
      e, a := n, true
  action doMap
    when a do
      if index mod 2 = 0 then
        next.reduce1(e * e)
      else
        next.reduce2(e * e)
      a := false
```



Formal Definition

By guarded commands with ... || ... and 〈 ... 〉 , e.g. [Andrews 91]

```
o.v               =   C_v(o)
o.m(e)            =   this.lock := false ; C_m(o, e) ; 〈 ¬ this.lock → this.lock := true 〉

o := new C(e)     =   o := C_new(e)
class C           =   var C: set(Ref) := {}
  var v: V            var C_lock: Ref → bool
  init ()             var C_v: Ref → V
    I                 procedure C_new() → (this: Ref)
  method m(u: U)        〈 this :∉ C ∪ {nil}; C := C ∪ {this}; this.lock := true 〉 ; I; this.lock := false
    when g do M       procedure C_m(u: U, this: Ref)
  ...                   〈 g ∧ ¬this.lock → this.lock := true 〉 ; M; this.lock := false
  action a            ...
    when h do A       procedure C_a(this: Ref)
  ...                   〈 h ∧ ¬this.lock → this.lock := true 〉 ; A; this.lock := false
```

fields become functions

methods become procedures

actions become procedures "called by the scheduler"

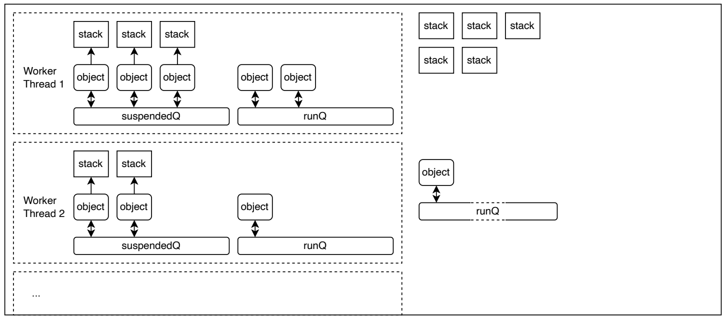
Program with classes C, ..., Start:

nondeterministic choice over all actions

```
(|| o ∈ C → C_a(o) [] ...) || ... || (var s: Start; s:= new Start())
```

parallel composition over all objects

Runtime Structure



M active objects are mapped to *N* worker threads (operating system threads)
For each object with a running action, a coroutine with stack is created
Stack is segmented: initially, only 4 KB; sufficient for most objects; grows as needed
Coroutines are scheduled cooperatively

Translation Scheme

originator = thread id

o.m() ➜ *unlock(this.lock) ; C_m(e, o, originator) ; lock(this.lock)*

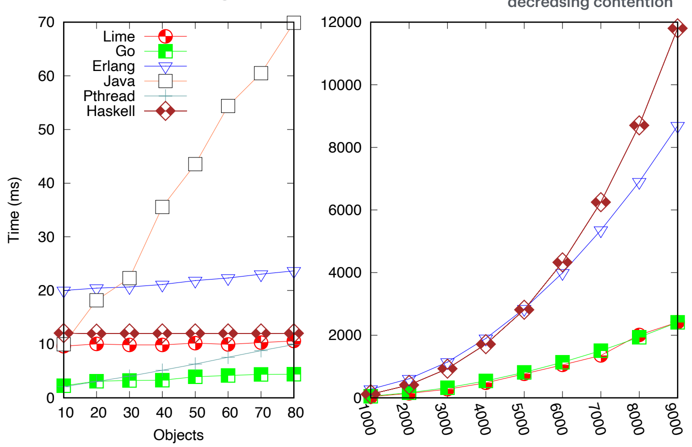
every object has field *lock*

```
method m(u: U)
    when g do M
➜
procedure C_m(u: U this: Ref, originator: Ref)
  while true do
    if lock(this.lock) then
    if g then
      M
      runQ.put(this)
      unlock(this.lock)
      return
    else
      unlock(this.lock)
  switch_to_sched(originator)
```

```
action a
    when h do A
➜
procedure C_a(this: Ref)
  const originator = this
  while true do
    if lock(this.lock) then
      if h then
        A
        unlock(this.lock)
      else
        unlock(this.lock)
  switch_to_sched(originator)
```

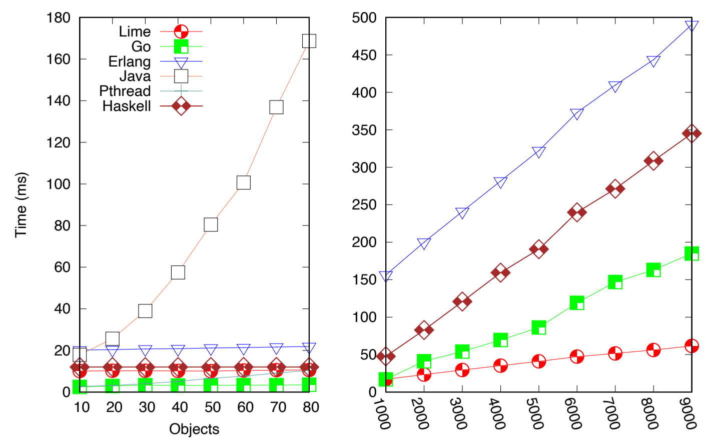
Priority Queue Timing

contention in root node
decreasing contention



Leaf-oriented Trees Timing

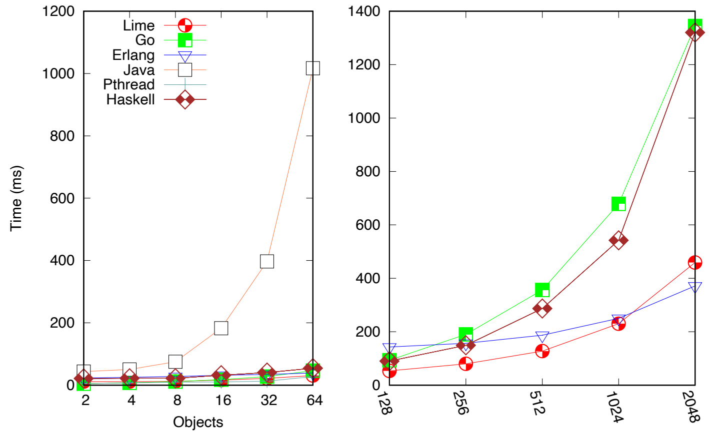
contention in root node
many objects, only few can execute



Map-Reduce Timing

contention in last reducer

many objects can execute



Conclusions

Experiment to evaluate language constraints

1. weakening atomicity to atomicity only up to method calls ➜ no need for backtracking
2. guards only over object fields ➜ reevaluated only after call to the object

and implementation techniques

1. allocating for each executing object a small stack that grows as needed
2. each object a user-level coroutine with fast cooperative scheduling (saving only three registers)
3. at most as many worker threads as there are cores
4. combining (lock-free) local and global queues for load balancing and work stealing
5. modifying the procedure call with efficient detection of stack overflow

for atomic guarded commands.

➜ Compares favourably with well-established implementations

Details in [Yao20]. Experiments so far only with small programs with simple guards.