A Personal Perspective on Concurrency

Jayadev Misra

Department of Computer Science
University of Texas at Austin

PLDI, Edinburgh
June 11, 2014
This talk is not about:

- A survey of the concurrency literature.
- Efficient algorithms for multicore computing.
- Proofs of protocols.
This talk is about:

- My view of asynchronous concurrency.
- How my views have shaped my research (with subtle commercials).
- Why the entire area needs significant new research.
Traditional view of concurrency in the 70s, early 80s

- add-on to traditional sequential programming,
- handled by enumerating threads:
  device controller, background monitor, timer, ...
- threads can fork to create new threads (not too many), interrupt others, ...
- fork, join paradigm for structuring.
Relevant questions Then

• How many machines?

• Topology of connection?

• Synchronization mechanism:
  Semaphore, Conditional critical region, Monitor, ...?

• Communication mechanism:
  Shared memory/message passing/broadcast interaction?

• Latency and bandwidths compared to in-memory interactions?
Reasoning methods

- Mostly ad-hoc.
- Enumeration of scenarios.
- Assertions at/before/after program control points.
- Interesting properties:
  - Mutual exclusion, absence of deadlock, starvation, ...
Some powerful ideas from the 80s

- Process calculi: CSP, CCS: structured enumeration of threads.
- Reasoning method of Owicki and Gries, using thread non-interference.
- Classification of program properties: Safety, progress
- Temporal logic.
CSP, 1978

- Process network with message passing. Shared memory regarded as a process.
- Rendezvous-based synchronization and communication.
- Simultaneous waiting on multiple channels.
- Process spawning.

Concurrency questions can be cast in completely machine-independent form.
Simulation of a Telephone switch

- Multiple processors in the switch.
- Multiple customer devices: handsets, fax machines, local switches.
- Multiple calls.

Possible partitioning into processes along many dimensions, none attractive.
Alternate view of structuring

A common misconception in program structuring is that a *process*—whose code can be executed on a single processor, or which can be viewed as a unit of computation, as in a transaction processing system, for instance—constitutes a “natural” decomposition of a system; therefore, it is argued that a system should be understood (i.e., specified) process by process.

*Parallel Program Design: A Foundation* (1988)
K. Mani Chandy, J. Misra
Pamela Zave’s view of Networking

**CLASSIC LAYERS OR OSI REFERENCE MODEL**

- there is a fixed number of layers
- each layer has a distinct and indispensable function

**THE GEOMORPHIC VIEW OF NETWORKING**

- each layer is a microcosm of networking, containing all the basic functions (state components and mechanisms)
- there can be any number of levels, each with any number of layers

the layers are modules, providing orderly, fine-grained separation of concerns
Unity

Goals:

- Structure a solution so that it admits: multiple views of problem decomposition.
- Structure a solution so that it admits: implementations on multiple platforms.
- Strong reasoning methods for safety and progress properties.
- Strong reasoning methods about program composition.
Common Meeting Time

• A set of students each with a personal calendar function $f$:
  
  $f(t)$ is the next time at/after $t$ when the student can meet.

• Find the earliest common meeting time, if one exists.
Some solutions

- Passing around a scratch pad.
- Bidding.
- Recursive decomposition.
An abstract solution

- Initially \( t = 0 \) —— earliest common meeting time \( \geq t \)

\[
t := f_0(t) \land t := f_1(t) \land \cdots \land t := f_n(t)
\]

- Execute the actions in arbitrary order forever.

Execute each action infinitely often.

- There is a common meeting time: eventually \( t \) set to the earliest. No action has any effect thereafter; so, \( t \) does not change.

- There is no common meeting time: \( t \) keeps increasing.

- Previous solutions are restrictions of this program.
Unity

- Discard the notion of a process.

- Replace concurrency by non-determinacy. Each indivisible unit of computation in any thread is an action.

- Reason about infinite sequences of actions using Unity logic, a form of linear temporal logic.

- An implementation will restrict non-determinacy.

- Any restriction that is fair retains all properties of the original program.
Structuring is manual

- Partition the set of actions into processes.
  Partitioning of variables across process boundaries.
  Shared variables.

- Hierarchies of partitioning.

- Point-point channel: variable with *puts* and *gets*.

- Broadcast channel: *get* does not remove the item from the channel.
Reasoning

- Safety: Invariant-based; induction on the number of execution steps.
- Progress (liveness): Induction on elementary proof steps.
- Reasoning about composite programs: Modular proofs.
  - Properties, not code, of components used in the proof.
  - Safety properties completely handled.
  - Progress properties almost completely handled.
Unity is applicable:

- Event processing is the primary function.
- In process control (telephony, train controllers ...)

- Mars Rover software: A fixed set of threads. Each thread is in an infinite loop:
  Receives a message.
  Processes the message; may send messages.
  Accepts the next message only on completion of processing.
  Each processing step guaranteed to terminate.

- Event-B
- TLA+
- Admits simpler model checking: UV, Murphi

Disclaimer: Unity may not have inspired any of this work.
Structured Programming

• **Structured Programming:**
  
  Structured programming circa 1968 (Dijkstra)
  
  = Component integration in a sequential world.

• **Structured Programming:**
  
  Structured programming circa 2014
  
  = Component integration in a concurrent world.
Concurrency is fundamental

- As fundamental as sequencing, branching and looping.

- Dynamic thread creation essential:
  Reasoning about individual threads is not scalable.

- A spectrum of synchronization and communication:
  from tightly-coupled OS processes to loosely-coupled applications.

- Separation of logical and physical across vast magnitude:
  mobility, names and domains, connections, ...
How to structure large concurrent programs

- Extremely simple structuring mechanism in UNITY.
- Consequently, UNITY inadequate for complex problems.
- More elaborate structuring in CSP, CCS, \( \pi \)-calculus.
Orc: a component integration system

Components:

- from many vendors,
- for many platforms,
- written in many languages,
- may run in real-time.

Integration: Sequential and/or concurrent executions of components.
Component Integration; contd.

- **Data types and Structures:** Boolean, List, Relational database, Objects
- **Small components:** add two numbers, print a file ...
- **Large components:** Linux, email server, a simulator, Web services ...
- **Time-based components:** clock, alarm, stopwatch, ...
- **Cyberphysical devices:** actuators, sensors, robots ...
- **Fast and Slow components**
- **Short-lived and Long-lived components**
- **Humans**
Orc Calculus

- **Site**: Basic component. *External to the calculus.*

- **Combinators** for integrating expressions.
  
  Interleaving: \( f \mid g \)
  
  Spawning/sequencing: \( f >x> g \)
  
  Interruption: \( f <x< g \)
  
  Bulk Synchronization: \( f ; g \)

- **Definitions** of sites in Orc.
Orc Calculus

- Calculus includes nothing other than the combinators and definitions.
- No data type, thread, process, channel, rendezvous
- New concepts are programmed in Orc using sites.
- No mutable store in the calculus.
Sites

- A site is called like a procedure with parameters.

- Site returns *(publishes)* any number of values at different times.

- A time-based site publishes at specific real time.
Examples of Sites

- **Constants**: 3, 7.2, true, "Orc" ...
- **Arithmetic, logical operators**: + − ∗ && || = ...
- **Data structures**: tuple, record, list, set, ...
- **Mutable actions**: println, Random, Prompt, Email ...
- **Instances of shared variables**: Ref, Semaphore, Channel, ...
- **Timer**
- **External Services**: Google Search, MySpace, CNN, Web services ...
- **Any Java Class instance, Any Orc Program**
- **Factory sites; Sites that create sites**: Semaphore, Channel ...
- **Humans** ... (Not a factory site, yet.)
Interleaving, Spawning

• Do $f$ and $g$ in parallel

Evaluate $f$ and $g$ independently. Publish all values from both.

• For all $x$ from $f$ do $g$

For all values published by $f$ do $g$. Publish only the values from $g$.

Notation: $f \gg g$ for $f \succ x \succ g$, if $x$ is unused in $g$.

Right Associative: $f \succ x \succ g \succ y \succ h$ is $f \succ x \succ (g \succ y \succ h)$
Interleaving, Spawning

- Do $f$ and $g$ in parallel
  \[ f \parallel g \]

  Evaluate $f$ and $g$ independently. Publish all values from both.

- For all $x$ from $f$ do $g$
  \[ f \gg x > g \]

  For all values published by $f$ do $g$. Publish only the values from $g$.

Notation: $f \gg g$ for $f \gg x > g$, if $x$ is unused in $g$.

Right Associative: $f \gg y > h \gg y > h$ is $f \gg x > (g \gg y > h)$
Interleaving, Spawning

- Do $f$ and $g$ in parallel $f \parallel g$

Evaluate $f$ and $g$ independently. Publish all values from both.

- For all $x$ from $f$ do $g$ $f \gg g$

For all values published by $f$ do $g$. Publish only the values from $g$.

Notation: $f \gg g$ for $f > x > g$, if $x$ is unused in $g$.

Right Associative: $f > x > g > y > h$ is $f > x > (g > y > h)$
Schematic of Sequential composition

Figure: Schematic of $f \rightarrow x \rightarrow g$
Subset Sum

Given integer $n$ and list of integers $xs$.

$\text{parsum}(n, xs)$ publishes all sublists of $xs$ that sum to $n$.

$\text{parsum}(5, [1,2,1,2]) = [1,2,2], [2,1,2]$

$\text{parsum}(5, [1,2,1])$ is silent.

All examples shown in this talk are irrelevant in practice.
An Orc program for Subset Sum

\[\text{def } \text{parsum}(0, []) = []\]

\[\text{def } \text{parsum}(n, []) = \text{stop}\]

\[\text{def } \text{parsum}(n, x: xs) =\]

\[\text{parsum}(n, xs) \quad \text{--- all sublists that do not include } x\]

\[| \text{parsum}(n - x, xs) >ys> x : ys \quad \text{--- all sublists that include } x\]
An Orc program for Subset Sum

\[
\text{def } \text{parsum}(0, []) = []
\]

\[
\text{def } \text{parsum}(n, []) = \text{stop}
\]

\[
\text{def } \text{parsum}(n, x : xs) =
\]

\[
\begin{align*}
\text{parsum}(n, xs) & \quad \text{--- all sublists that do not include } x \\
| \text{parsum}(n - x, xs) & >ys> x : ys \quad \text{--- all sublists that include } x
\end{align*}
\]
An Orc program for Subset Sum

\[
def \text{parsum}(0, []) = []
\]

\[
def \text{parsum}(n, []) = \text{stop}
\]

\[
def \text{parsum}(n, x : xs) = \\
\quad \text{parsum}(n, xs) \quad \text{-- all sublists that do not include } x \\
\quad | \text{parsum}(n - x, xs) > ys > x : ys \quad \text{-- all sublists that include } x
\]
Interleaving, Spawning, Bulk Synchrony

Do \( f \) and \( g \) in parallel

\[ f \parallel g \]

Evaluate \( f \) and \( g \) independently. Publish all values from both.

For all \( x \) from \( f \) do \( g \)

\[ f >x> g \]

For all values published by \( f \) do \( g \). Publish only the values from \( g \).

- Do \( f \). If \( f \) halts without publishing do \( g \).

\[ f ; g \]
Given integer $n$ and list of integers $xs$.

$\text{seqsum}(n, xs)$ publishes the first sublist of $xs$ that sums to $n$.

“First” is smallest by index, lexicographically.

$\text{seqsum}(5, [1, 2, 1, 2]) = [1, 2, 2]$

$\text{seqsum}(5, [1, 2, 1])$ is silent.
Program for Subset Sum, Backtracking

\[
def \text{seqsum}(0, []) = []
\]

\[
def \text{seqsum}(n, []) = \text{stop}
\]

\[
def \text{seqsum}(n, x : xs) =
\]
\[
    x : \text{seqsum}(n - x, xs)
\]
\[
; \text{seqsum}(n, xs)
\]
Interleaving, Spawning, Bulk Synchrony, Pruning

Do $f$ and $g$ in parallel $f \mid g$

Evaluate $f$ and $g$ independently. Publish all values from both.

For all $x$ from $f$ do $g$ $f \gg x \gg g$

For all values published by $f$ do $g$. Publish only the values from $g$.

Do $f$. If $f$ halts without publishing do $g$. $f ; g$

• For some $x$ from $g$ do $f$ $f < x < g$
Pruning:  $f < x < g$

For some value published by $g$ do $f$.

- Evaluate $f$ and $g$ in parallel.
- Site calls that need $x$ are suspended.
- When $g$ returns a (first) value:
  - Bind the value to $x$.
  - Kill $g$.
  - Resume suspended calls.
- Values published by $f$ are the values of $(f < x < g)$.

Notation: $f \ll g$ for $f < x < g$, if $x$ is unused in $f$.

Left Associative: $f < x < g < y < h$ is $(f < x < g) < y < h$
Pruning: $f \prec x \prec g$

For some value published by $g$ do $f$.

- Evaluate $f$ and $g$ in parallel.
- Site calls that need $x$ are suspended.
- When $g$ returns a (first) value:
  - Bind the value to $x$.
  - Kill $g$.
  - Resume suspended calls.
- Values published by $f$ are the values of $(f \prec x \prec g)$.

Notation: $f \ll g$ for $f \prec x \prec g$, if $x$ is unused in $f$.

Left Associative: $f \prec x \prec g \prec y \prec h$ is $(f \prec x \prec g) \prec y \prec h$
Subset Sum (Contd.), Concurrent Backtracking

Publish the first sublist of \( xs \) that sums to \( n \).

Run the searches concurrently.

\[
\begin{align*}
\text{def } \text{parseqsum}(0, []) &= [] \\
\text{def } \text{parseqsum}(n, []) &= \text{stop} \\
\text{def } \text{parseqsum}(n, x : xs) &= \\
&(p ; q) \\
&<p< x : \text{parseqsum}(n - x, xs) \\
&q< \text{parseqsum}(n, xs)
\end{align*}
\]

Note: Neither search in the last clause may succeed.
Publish the first sublist of $xs$ that sums to $n$.

Run the searches concurrently.

$$
def \text{parseqsum}(0, []) = []$$

$$
def \text{parseqsum}(n, []) = \text{stop}$$

$$
def \text{parseqsum}(n, x : xs) =
(p ; q)
\langle p < x : \text{parseqsum}(n - x, xs) \rangle
\langle q < \text{parseqsum}(n, xs) \rangle$$

Note: Neither search in the last clause may succeed.
Angelic + Demonic non-determinism

Programs to solve combinatorial search problems may often be simply written by using multiple-valued functions. Such programs, although impossible to execute directly on conventional computers, may be converted in a mechanical way into conventional backtracking programs.


• $f > x > g$ —— Angelic: Explore all paths

• $f < x < g$ —— Demonic: Explore some path, prune others.

• Any combination of these.
On Evaluating Programming Theories

- **(Easier)** Establish properties of the theory: Introspection
  - Internal consistency, through a semantics
  - Basic Identities
  - Proof theory

- **(Harder)** Establish applicability of the theory: Extrospection
  - Encode accepted programming paradigms
  - Explore limitations
  - Validate intentions empirically
Identities of $|$, $\gg$, $\ll$ and $;$

(Zero and $|$) $f \mid \text{stop} = f$

(Commutativity of $|$) $f \mid g = g \mid f$

(Associativity of $|$) $(f \mid g) \mid h = f \mid (g \mid h)$

(Left zero of $\gg$) $\text{stop} \gg f = \text{stop}$

(Associativity of $\gg$) if $h$ is $x$-free

$(f \gg x > g) \gg y > h = f \gg x > (g \gg y > h)$

(Right zero of $\ll$) $f \ll \text{stop} = f$

(generalization of right zero)

$f \ll g = f \ll (\text{stop} \ll g) = f \mid (\text{stop} \ll g)$

(relation between $\ll$ and $<x<$)

$f \ll g = f < x < g$, if $x \not\in \text{free}(f)$.

(commutativity)

$(f < x < g) < y < h = (f < y < h) < x < g$

if $x \not\in \text{free}(h)$, $y \not\in \text{free}(g)$, and $x$, $y$ are distinct.

(associativity of $;$)

$(f ; g) ; h = f ; (g ; h)$
Distributivity Identities

( | over >x> ; left distributivity)
\[(f \mid g) >x> h = f >x> h \mid g >x> h\]

( | over <x< )
\[(f \mid g) <x< h = (f <x< h) \mid g, \text{ if } x \not\in \text{free}(g).\]

( >y> over <x< )
\[(f >y> g) <x< h = (f <x< h) >y> g\]
if \(x \not\in \text{free}(g)\), and \(x\) and \(y\) are distinct.

( <x< over ; )
\[(f <x< g) ; h = (f ; h) <x< g, \text{ if } x \not\in \text{free}(h).\]
Identities that don’t hold

(Idempotence of $|$) \[ f \mid f = f \]

(Right zero of $\gg$) \[ f \gg \text{stop} = \text{stop} \]

(Left Distributivity of $\gg$ over $|$) \[ f \gg (g \mid h) = (f \gg g) \mid (f \gg h) \]
Baby steps towards a Proof Theory

Example: Commutative, associative fold \(+\) on a set

```python
def fold(S, n):
    # set S has n elements.
    # f(k) folds and reduces set size by k.
    def f(0) = stop
    def f(1) = (S.get(), S.get()) > (x, y) > S.put(x + y) >> stop
    def f(k) = f(1) | f(k - 1)

    f(n - 1)
```

Surprisingly: Unity proof theory seems appropriate.
Baby steps towards a Proof Theory

Example: Commutative, associative fold $+_{\text{on a set}}$

\[
def \text{fold}(S, n) = \quad \text{-- set } S \text{ has } n \text{ elements.}
\]

\[
\text{-- } f(k) \text{ folds and reduces set size by } k.
\]

\[
def f(0) = \text{stop}
\]

\[
def f(1) = (S.\text{get}(), S.\text{get}()) > (x, y) > S.\text{put}(x + y) \gg \text{stop}
\]

\[
def f(k) = f(1) \mid f(k - 1)
\]

\[
f(n - 1)
\]

Surprisingly: Unity proof theory seems appropriate.
I will show some programming paradigms encoded in Orc

- Implicit concurrency
- Synchronization
- Process network
- Real time
- Virtual time
Orc Language

- **Data Types**: Number, Boolean, String, with Java operators

- **Data structures**: Tuple, List, Record

- **Pattern Matching; Clausal Definition**

- **Closure**

- **Class; active objects**

All features implemented as macros and/or calls to library sites.
Site arguments are concurrently evaluated

• $f(g, h)$ is translated to:

• $(f(x, y) < x < g) < y < h$

• From semantics of $\ll$, $g$ and $h$ concurrently evaluated.
Implicit Concurrency

- An experiment tosses two dice; success if sum of throws is 7.
- \(\exp(n)\) runs \(n\) experiments and reports the number of successes.

\[
def \ toss() = \text{Random}(6) + 1 \\
\quad \text{---} \ toss \text{ returns a random number between 1 and 6.}
\]

\[
def \ exp(0) = 0
\]

\[
def \ exp(n) = \exp(n - 1) \\
\quad + (\text{if } \text{toss()} + \text{toss()} = 7 \text{ then } 1 \text{ else } 0)
\]

Number of concurrent calls to \(\text{toss}\) = ?
Synchronization: Pure Rendezvous

val $s = Semaphore(0)$  \—— $s$ is a semaphore with initial value 0.
val $t = Semaphore(0)$

def put() = $s$.acquire() \gg $t$.release()
def get() = $s$.release() \gg $t$.acquire()
Typical Iterative Process

Forever: Read $x$ from channel $c$, compute with $x$, output result on $e$:

```python
def p(c, e) = c.get()  >x>  Compute(x)  >y>  e.put(y)  > p(c, e)
```

Figure: Iterative Process
Composing Processes into a Network

Process (network) to read from both $c$ and $d$ and write on $e$:

$$\text{def } \text{net}(c, d, e) = p(c, e) \mid p(d, e)$$

Figure: Network of Iterative Processes
Workload Balancing

Read from \( c \), assign work randomly to one of the processes.

\[
\text{def } \text{bal}(c, c', d') = c\.get() > x > \text{random}(2) > t > (\text{if } t = 0 \text{ then } c'.put(x) \text{ else } d'.put(x)) \Rightarrow \text{bal}(c, c', d')
\]

\[
\text{def } \text{workbal}(c, e) = \text{val } c' = \text{Channel}() \\
\text{val } d' = \text{Channel}() \\
\text{bal}(c, c', d') \mid \text{net}(c', d', e)
\]
Recursive Pipeline network

Consider computing factorial of each input.

\[ fac(x) = \begin{cases} 
1 & \text{if } x = 0 \\
x \times fac(x - 1) & \text{if } x > 0 
\end{cases} \]

Suppose \( x \leq N \), for some given \( N \).
Real time: Metronome

External site $Rwait(t)$ returns a signal after $t$ time units. *metronome* publishes a *signal* every time unit.

\[
def \text{metronome}() = \text{signal}_{S} | (Rwait(1) \gg \text{metronome}())_{R}
\]
Parallel Auction

- A list of bidders in a single-round auction.
- Ask for bids from all bidders, then publish the highest bid.
- \( b.ask() \) requests a bid from bidder \( b \).
  All bidders called simultaneously.
- Bid is 0 if no response from a bidder within 8 seconds.

\[
\begin{align*}
\text{def } & \text{ auction}([]) = 0 \\
\text{def } & \text{ auction}(b : bs) = \\
& \quad \text{max} ( \\
& \quad \quad b.ask() \mid (Rwait(8000) \gg 0), \quad \text{both arguments run in parallel} \\
& \quad \quad \text{timeout alternative} \\
& \quad \quad \text{ auction}(bs) \\
& \quad )
\end{align*}
\]
The Subtle Commercial

• Orc web site:
  http://orc.csres.utexas.edu/

• Reference manual, user guide and research papers.

• Web site at which Orc programs can be submitted.

• Download Orc to your computer.

• Student Projects.
The Subtle Commercial; contd.

- All known synchronization communication protocols coded in Orc.

- Several student projects covering different application areas: Twitter search, Music composition, Trip manager, Workflow.

- Current work in Robotics, two robots patrolling a perimeter.
Shortest Path Algorithm with Lights and Mirrors

- Source node sends rays of light to each neighbor.

- Edge weight is the time for the ray to traverse the edge.

- When a node receives its first ray, sends rays to all neighbors. Ignores subsequent rays.

- Shortest path length = time for sink to receive its first ray. Shortest path length to node $i =$ time for $i$ to receive its first ray.
Graph structure in $\text{Succ}(u)$

Figure: Graph Structure

$\text{Succ}(u)$ publishes $(x, 2), (y, 1), (z, 5)$.
Algorithm

\[ \text{def } \text{eval}(u, t) = \begin{align*}
&\text{record value } t \text{ for } u \\
&\text{for every successor } v \text{ with } d = \text{length of } (u, v): \\
&\quad \text{wait for } d \text{ time units} \\
&\quad \text{eval}(v, t + d)
\end{align*} \]

Goal: \( \text{eval}(\text{source}, 0) \mid \)
- read the value recorded for the sink

Record path lengths for node \( u \) in FIFO channel \( u \).
Algorithm (contd.)

```python
def eval(u, t) =  
              record value t for u  
for every successor v with d = length of (u, v) :
  wait for d time units  
  eval(v, t + d)
```

Goal:
```
  eval(source, 0) |  
  read the value recorded for the sink
```

________________________________________

A cell for each node where the shortest path length is stored.

```python
def eval(u, t) =  
              u := t  
  Succ(u) > (v, d)>
  Rwait(d)  
  eval(v, t + d)
```

{- Goal :-}  
```
eval(source, 0) |  read sink
```
Algorithm (contd.)

```python
def eval(u, t) =
  u := t \gg
  Succ(u) > (v, d) >
  Rwait(d) \gg
  eval(v, t + d)
```

\{- Goal :-\} eval(source, 0) | read sink

- Any call to `eval(u, t)`: Length of a path from source to `u` is `t`.
- First call to `eval(u, t)`: Length of the shortest path from source to `u` is `t`.
- `eval` does not publish.
Drawbacks of this algorithm

- Running time proportional to shortest path length.
- Executions of \textit{Succ}, \textit{put} and \textit{get} should take no time.
Simulation: Bank

- Bank with two tellers and one queue for customers.
- Customers generated by a *source* process.
- When free, a teller serves the first customer in the queue.
- Service times vary for customers.

Determine
  - Average wait time for a customer.
  - Queue length distribution.
  - Average idle time for a teller.
Description of Bank

\[\text{def Bank}() = (\text{Customers}() \mid \text{Teller}() \mid \text{Teller}()) \gg \text{stop}\]

\[\text{def Customers}() = \text{Source}() \gg c \gg \text{enter}(c)\]

\[\text{def Teller}() = \text{next}() \gg c \gg \text{Vwait}(c.\text{ServTime}) \gg \text{Teller}()\]

\[\text{def enter}(c) = q.\text{put}(c)\]
\[\text{def next}() = q.\text{get}()\]
A Significant research area: Programming

- Obliterate the distinction between programming and concurrent programming.

- Structuring is fundamental.

- Orc suggests one mechanism, orchestration of components.

- Components should be coded in the most efficient way, perhaps in other languages.

- Intrinsic merit of concurrency as a field of study, apart from its applications in Robotics, Mobile computing, smart city management, ...
Role of universities in the age of MOOCS: Where I can get diverse expertise.

- at UT,
  - Lorenzo Alvisi
  - Don Batory
  - William Cook
  - Isil Dillig
  - Tom Dillig
  - Allen Emerson
  - Mohamed Gouda
  - Warren Hunt
  - Matt Kaufmann
  - Simon S. Lam
  - Vladimir Lifschitz
  - Calvin Lin
  - J Moore
  - Keshav Pingali
  - Peter Stone
  - David Zuckerman