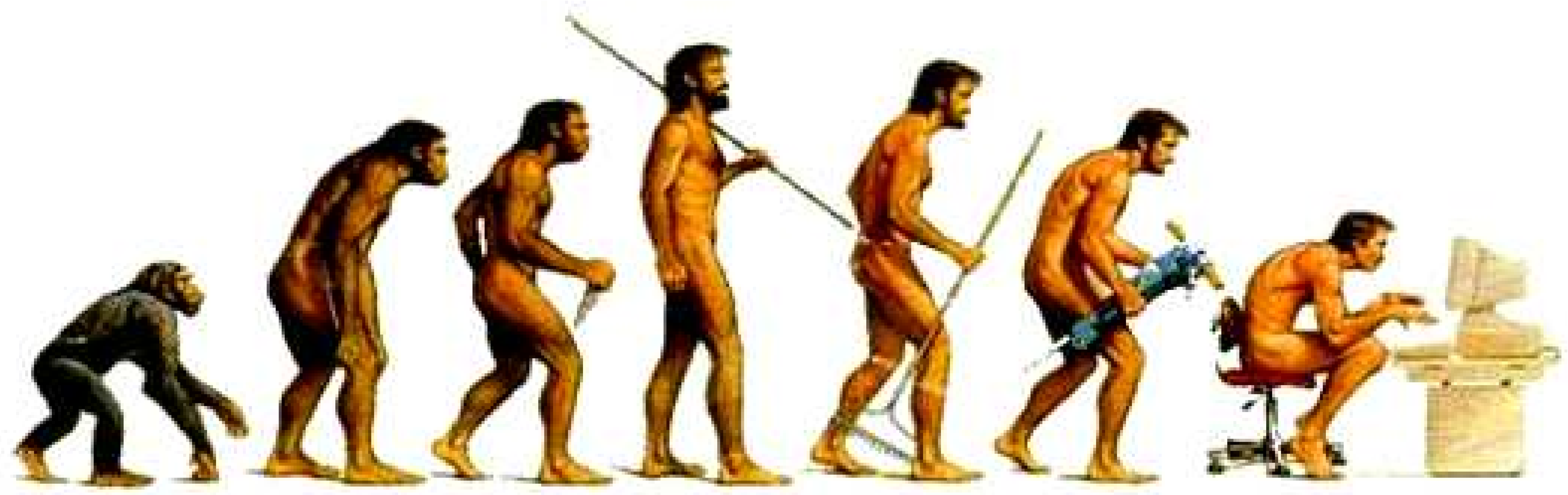
# Determinism Should Ensure Deadlock-Freedom

**Nalini Vasudevan and Stephen A. Edwards**

**Columbia University**




## Data Races

```
void f(shared int &a) {
  a = 3;
}

void g(shared int &b) {
  b = 5;
}

main() {
  shared int x = 1;
  spawn f(x);
  g(x);
  sync; /* Wait for f and g to finish */
  print x;
}
```

The above program creates two tasks *f* and *g* in parallel using the *spawn* construct. *x* is being modified concurrently by the two tasks and therefore the program is not race-free.

By determinism, we mean the output behavior of the program is independent of the scheduling choices (e.g., the operating system) and depends only on the input behavior.

## Non-Determinism

A remedy to avoid races is to introduce locks.

```
lock p;

void f(shared int &a) {
  lock (p);
  a = 3;
  unlock (p);
}

void g(shared int &b) {
  lock (p);
  b = 5;
  unlock (p);
}

main() {
  shared int x = 1;
  spawn f(x);
  g(x);
  sync; /* Wait for f and g to finish */
  print x;
}
```

Even though x is protected by a lock, the value printed by this program is either 3 or 5 depending on the schedule. Therefore, it is non-deterministic.

## Deadlocks

The problem with locks: incorrect usage may lead to deadlocks.

```
lock p, q;

void f(shared int &a) {
  lock (p);
  lock (q);
  a = 3;
  unlock (q);
  unlock (p);
}

void g(shared int &b) {
  lock (q);
  lock (p);
  b = 5;
  unlock (p);
  unlock (q);
}

main() {
  shared int x = 1;
  spawn f(x);
  g(x);
  sync; /* Wait for f and g to finish */
  print x;
}
```

## The SHIM Model

**SHIM allows single writes and in a synchronized fashion.** Tasks in SHIM run asynchronously but synchronize explicitly using rendezvous communication. There is no shared data.

SHIM is a C-like language with additional constructs for concurrency:

| | |
|---|---|
| *stmt*$_1$ *par stmt*$_2$ | Run *stmt*$_1$ and *stmt*$_2$ concurrently |
| *send var* | Send on channel *var* |
| *recv var* | Receive on channel *var* |

## An Example in SHIM

```
void f(in a) {
  a = 3;
  recv a;
  /* a is now 5 */
}

void g(out b) {
  b = 5;
  send b;
  /* b = 5 */
}

main() {
  chan int c;
  f(c) par g(c);
  /* c is 5 */
}
```

This program creates two tasks, *f* and *g*, and runs them in parallel. The *par* statement blocks until both *f* and *g* terminate. *c* is a channel and both *a* and *b* are incarnations of *c*. *g* takes *c* by *out* (reference); any modification of *b* is therefore reflected in main's *c*. *f* takes *c* by *in* (value), and hence *f* maintains a local copy of *c*. Suppose *f* wants to receive the updated value, then it explicitly calls *recv* on *a*. This statement synchronizes with *send b* of *g* to exchange values.

The SHIM model prohibits any variable from being passed by reference (*out*) to more than one task at a time and this makes it impossible for a task to modify another task's copy of a variable through a simple assignment.

## Deadlocks in SHIM

Even though SHIM is deterministic, it can introduce deadlocks. Consider a program below. Task *f*'s *send a* waits for a matching *recv a* from task *g*. Task *g*'s *send b* waits for a matching *recv b* from task *f*. The two tasks *f* and *g* wait infinitely for each other causing a deadlock.

```
void f(out a, in b) {
  /* wait for recv a from task g */
  send a = 1;
  recv b; /* unreached */
}

void g(out b, in a) {
  /* wait for recv b from task f */
  send b = 2;
  recv a; /* unreached */
}
```

## Resolving deadlocks in SHIM

We maintain a dependency graph during runtime and check for cycles. If a cycle is detected, the deadlock is broken.

SHIM's semantics makes cycle detection algorithm easy. A process can block atmost on one channel at a time, therefore allowing atmost one outgoing edge out of any node.

```
void f(out a, in b) {
  /* Wait for recv a from task g */
   send a = 1; /* Deadlocking action; deadlock broken */
   /* 1 is written to a */
   recv b; /* Deadlocking action; deadlock broken */
  /* 2 is read */
}

void g(out b, in a) {
  /* Wait for recv b from task f */
  send b = 2; /* Deadlocking action; deadlock broken */
  /* 2 is written to b */
  recv a; /* Deadlocking action; deadlock broken */
  /* 1 is read */
}
```

## Deadlock Breaking Algorithm in SHIM

f——g

[*f* blocks at *send a*]

f⇄g

[Next, *g* blocks at *send b*]

f g

[*g* detects a cycle, revives *f*, and breaks the cycle, ]

f g

[As the cycle is broken, *f* writes 1 to *a* and *g* writes 2 to *b*]

f——g

[*f* blocks at *recv b*]

f⇄g

[Next, *g* blocks at *recv a*]

f g

[*g* detects a cycle, revives *f*, and breaks the cycle ]

f g

[As the cycle is broken, *f* gets 1 from *b* and *g* gets 2 from *a*]