Determinism Should Ensure Deadlock-Freedom
Nalini Vasudevan and Stephen A. Edwards
Columbia University

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: send, par stmts, Run stmts, and exit concurrently.

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);
  // Deadlocking action; deadlock broken
  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.

A remedy to avoid races is to introduce locks.

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

Non-Determinism

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

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

Deadlocks
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.

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 on one channel at a time, therefore allowing almost one outgoing edge out of any node.

Resolving deadlocks in SHIM

Even though SHIM is deterministic, it can introduce deadlocks. Consider a program below. Task f’s sends a waits for a matching recv a from task g. Task g’s sends 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 b) {
  b = 5;
  send b;
  // b = 5 waits
}

main() {
  chan int c;
  // Par g(c);
  // c = c; 5 waits
}

If a cycle is detected, the deadlock is broken.

void f(out b) {
  b = 5;
  send b;
  // b = 5 waits
}

main() {
  chan int c = 1;
  spawn f(c);
  g(c);
  // Deadlocking action; deadlock broken
  print c;
}

A remedy is to introduce locks.

lock p;
void f(out b) {
  lock (p);
  b = 5;
  unlock (p);
}

Deadlock Breaking Algorithm in SHIM

[f blocks at send a] f
[Next, g blocks at send b] g
[g detects a cycle, revives f, and breaks the cycle.]

As the cycle is broken, f writes 1 to a and g writes 2 to b]
[f blocks at recv b] f
g
[g detects a cycle, revives f, and breaks the cycle]

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