Resource-oriented deadlock analysis

Lee Naish
Computer Science and
Software Engineering
University of Melbourne

Outline

A logic programming pearl

The dining philosophers problem

Process-oriented deadlock analysis

Concurrent logic programming

Finding deadlocks

Discussion

Conclusion

A logic programming pearl

The logic programming paradigm often exposes the logic of a problem which is obfuscated in imperative paradigms

This can lead to new or clearer understanding of the problem and how it can be solved

Here we deal with the (undecidable) problem of determining if a system of communicating concurrent processes can *deadlock*

The dining philosophers problem

N philosophers sit around a table with a fork (or chop stick) between each one and food in the middle

Philosophers repeatedly think, get hungry, get one adjacent fork then the other, eat and release the forks

Obtaining a fork may require waiting for an adjacent philosopher to finish using it

A software simulation can deadlock if (eg) each philosopher process first obtains the fork to their right

One solution is to number the philosophers and make (only) odd philosophers left handed

Our code supports left and right handed philosophers and those who alternate between hands; it uses *locks*

Process-oriented deadlock analysis

The standard way to determine if a deadlock can occur is to start by considering all possible *interleavings* of processes

Some chunks of computation (eg, "thinking") don't access shared resources so interleaving within these can be avoided (called "coarsification")

We can generalise "independent" operations:

1. they commute, and
2. they don't affect suspension of each other

With 9 philosophers and 8 operations per lock there are around 2.17×10^{62} potential interleavings (they can be generated incrementally to reduce the search space)

Concurrent logic programming

Multiple conjuncts of the current goal are reduced concurrently

Communication is achieved by binding/matching shared variables (often lists or "streams"): `?- p(..., Xs), q(Xs, ...), ...`

Distributed backtracking is very hard to implement, so each call is only ever matched with one clause

Usually calls suspend until only one clause matches (we can also use the suspension mechanism for concurrency control)

Sometimes one of several matching clauses is picked and *commited* to (*committed choice* nondeterminism)

Parallel NU-Prolog

NU-Prolog has a suspension mechanism (when declarations)

A subset of the language can be run without backtracking and a parallel version was implemented

A pre-processor was implemented to help use this subset

LazyDet declarations force suspension until just one clause matches

EagerDet declarations are used for committed choice

Implementing locking

```
    ..., Ls = lock(42, Locked, Unlocked).Ls1,
    when(nonvar(Locked),
        (<critical section>, Unlocked = unlocked)), ...

% Start lock1 process, initially unlocked
lock(Ls) :- lock1(unlocked, Ls).

% wait until unlocked and a new lock message arrives,
% allow entry to lock, recurse with Unlock flag from
% new message
:- lock1(U, L.Ls) when U and L. % causes process to wait
lock1(unlocked, lock(_I, Locked, Unlocked).Ls) :-
    Locked = locked, % back-communication
    lock1(Unlocked, Ls).
```

A dining Philosopher

```
p_think(P, LMs, RMs) :- p_hungry(P, LMs, RMs).

p_hungry(P, LMs, RMs) :- p_get_first(P, LMs, RMs).

:- p_get_first(p(_,H,_), _, _) when H.
p_get_first(p(I,r,B), LMs, lock(I,Locked,U1).RMs) :-
    p_get_snd(p(I,r,B), Locked, U1, LMs, RMs).
p_get_first(p(I,l,B), lock(I,Locked,U1).LMs, RMs) :-
    p_get_snd(p(I,l,B), Locked, U1, LMs, RMs).

:- p_get_snd(p(_,H,_), Locked,_,_,_) when H and Locked.
p_get_snd(p(I,l,B),locked,U1,LMs,lock(I,Locked,U2).RMs):-
    p_eat(p(I, l, B), Locked, U1, U2, LMs, RMs).
p_get_snd(p(I,r,B),locked,U1,lock(I,Locked,U2).LMs,RMs):-
    p_eat(p(I, r, B), Locked, U1, U2, LMs, RMs).
```

A dining Philosopher (cont.)

```
% :- lazyDet p_eat(i, i, o, o, o, o).
:- p_eat(_, Locked, _, _, _, _) when Locked.
p_eat(P, locked, U1, U2, LMs, RMs) :-
    p_release_both(P, U1, U2, LMs, RMs).

p_release_both(P, unlocked, unlocked, LMs, RMs) :-
    next_handedness(P, P1),
    p_think(P1, LMs, RMs).

:- next_handedness(p(I, H, B), _) when H and B.
next_handedness(p(I, l, 0), p(I, l, 0)).
next_handedness(p(I, r, 0), p(I, r, 0)).
next_handedness(p(I, l, 1), p(I, r, 1)).
next_handedness(p(I, r, 1), p(I, l, 1)).
```

Sharing resources

Each philosopher has two streams of lock messages

Each fork has two streams of of lock messages which must be (nondeterministically) merged

```
% Just 3 philosophers...
main([P1,P2,P3], [Ms1,Ms2,Ms3]) :-
                          lock(Ms1),
          lock(Ms2),                    lock(Ms3),

      p_think(P1,Ms1a,Ms2b),   p_think(P3,Ms3a,Ms1b),
                  p_think(P2,Ms2a,Ms3b),

                   merge(Ms1a,Ms1b,Ms1),
       merge(Ms2a,Ms2b,Ms2),   merge(Ms3a,Ms3b,Ms3).
```

Committed choice merge

```
% committed choice merge of two (infinite) streams
% :- eagerDet merge(i, i, o). % specifies modes, CC
% merge(A.As, Bs, A.Cs) :- !, merge(As, Bs, Cs).
% merge(As, B.Bs, B.Cs) :- !, merge(As, Bs, Cs).
% The preprocessor translates this to...
:- merge(A, B, C) when A or B.
merge(A,B,C) :- nonvar(A),A=[D|E],!, C=[D|F],merge(E,B,F).
merge(A,B,C) :- nonvar(B),B=[D|E],!, C=[D|F],merge(A,E,F).
```

The control (mode) information is used to generate the (disjunctive) delay condition *and* the `nonvar` tests

Finding Deadlocks

We use a three-stage transformation:

1. Committed choice → multiple solutions
   (so backtracking search can be used to find deadlocks)
2. Remove spurious derivations
   (some are introduced by stage 1)
3. Implement fair search
   (since the search space is often very large or infinite)

Then run (or analyse) the resulting code

Transformation 1

```
:- merge(A, B, C) when A or B. % as before
merge(A, B, C) :- m1(A, B, C). % new proc for clause 1
merge(A, B, C) :- m2(A, B, C). % new proc for clause 2

:- m1(A, B, C) when A. % cut, nonvar now redundant
m1(A,B,C) :- nonvar(A), A=[D|E], C=[D|F], merge(E,B,F).

:- m2(A, B, C) when B. % cut, nonvar now redundant
m2(A,B,C) :- nonvar(B), B=[D|E], C=[D|F], merge(A,E,F).
```

Control now used in *three* places

Deadlocked/floundered derivations preserved if we ignore derivations with `m1` or `m2` floundered

Transformation 2

An extra argument `W` is added to the new procedures and all that (recursively) call them

The new procedures resume when `W` is bound, which is done after the rest of the computation flounders (so the calls fail)

```
:- merge(A, B, C, W) when A or B.
merge(A, B, C, W) :- m1(A, B, C, W).
merge(A, B, C, W) :- m2(A, B, C, W).

:- m1(A, B, C, W) when A or W.
m1(A,B,C,W) :- nonvar(A), A=[D|E],C=[D|F], merge(E,B,F,W).

:- m2(A, B, C, W) when B or W.
m2(A,B,C,W) :- nonvar(B), B=[D|E],C=[D|F], merge(A,E,F,W).
```

Transformation 3

We use iterative deepening for the recursion depth of *nondeterministic* procedures by adding another argument

```
:- merge(A, B, C, W, S) when A or B.
merge(A, B, C, W, s(S)) :- m1(A, B, C, W, S).
merge(A, B, C, W, s(S)) :- m2(A, B, C, W, S).

:- m1(A, B, C, W, S) when A or W.
m1(A,B,C,W,S):-nonvar(A),A=[D|E],C=[D|F],merge(E,B,F,W,S).

:- m2(A, B, C, W, S) when B or W.
m2(A,B,C,W,S):-nonvar(B),B=[D|E],C=[D|F],merge(A,E,F,W,S).
```

Top level code

```
main(S, [P1,P2,P3], [Ms1,Ms2,Ms3]) :-
    lock(Ms1), lock(Ms2), lock(Ms3),
    p_think(P1,Ms1a,Ms2b), p_think(P2,Ms2a,Ms3b),
    p_think(P3,Ms3a,Ms1b), merge(Ms1a,Ms1b,Ms1,W,S),
    merge(Ms2a,Ms2b,Ms2,W,S), merge(Ms3a,Ms3b,Ms3,W,S),
    W = wake. % resume+fail spurious deadlocks with m1/m2

depth(0).
depth(s(S)) :- depth(S).

d(S) :- depth(S), write('Depth limit: '), writeln(S).

?-d(S),main(S,[p(1,l,1),p(2,r,1),p(3,l,1)],[Ms1,Ms2,Ms3]).
```

Discussion

For 9 philosophers a complete search to depth 4 (≤ 8 lock operations each) takes 100–300s running NU-Prolog on a 1GHz Intel Pentium III

Interleaving of processes doesn't affect the search space at all — the result of a derivation doesn't depend on the execution order!

"Independence" is the normal state of affairs in logic programming — calls are generally commutative, including "unlock" operations, assuming callability is "monotonic"

Committed choice procedures are an exception

The search space depends on contention for access to shared resources (the clause selected in each `merge` call)

Conclusion

The logic programming paradigm leads us to a different way of thinking about deadlock analysis: *resource*-oriented rather than *processes*-oriented

The definition of "independence" could possibly be weakened

The use of different concurrency primitives (for example, monitors) may help simplify deadlock analysis