Practical Approaches to Formally Verify Concurrent Software

Ganesh Gopalakrishnan

Microsoft HPC Institutes,
NSF CNS-0509379
SRC Contract TJ 1318

http://www.cs.utah.edu/formal_verification

(talk is kept at above URL under presentations/ce-junior-seminar-2008.pptx)
Multicores are the future!

Why?

(photo courtesy of Intel Corporation.)
Multicores are the future!

Why?

- That is the only way to get more compute power
- Energy per compute operation lowered thru the use of parallelism
  - e.g. Today’s DSP processors

(photo courtesy of Intel Corporation.)
How many different types of multicore CPUs?

- Quite a variety
- Use in cloud computing
- Use in hand-helds
- Use in embedded devices
- Use in Graphics / Gaming
- Use in HPC
- ...

- A plethora of SW and HW challenges

(photo courtesy of Intel Corporation.)
HW design challenges: Hierarchical Cache Coherence Protocols will play a major role in multi-core processors

Chip-level protocols

Intra-cluster protocols

Inter-cluster protocols

State Space grows multiplicatively across the hierarchy!
Verification will become harder
Other challenges: Varying grains of atomicity in modeling/verification

One step in high-level

```
home -> remote
```

Multiple steps in low-level

```
home -> router -> buf -> remote
```

→ an atomic guarded/command
Diagrams showing Xiaofang Chen’s PhD work that deals with complex cache coherence protocols: Abstraction / refinement
Step 1 of Refinement
Step 2 of Refinement
Final Step of Refinement

Inv P1  Inv P2  Inv P3

Inv P1'  Inv P2'

Inv P1  Inv P2  Inv P3'

Inv P1'  Inv P2'

Inv P1  Inv P2  Inv P3''

Inv P1'  Inv P2'
Multicore Challenges:
Need to employ / teach concurrent programming at an unprecedented scale!

Some of today’s proposals:
- Threads (various)
- Message Passing (various)
- Transactional Memory (various)
- OpenMP
- MPI
- Intel’s Ct
- Microsoft’s Parallel Fx
- Cilk Arts’s Cilk
- Intel’s TBB
- Nvidia’s Cuda
- ...

(photo courtesy of Intel Corporation.)
Sequential program verification remains hard!

```c
main(){
  int Z1, Z2, Z3;
  int x1, x2;
  int z11, z12, z13, z21, z22, z23;
  /* x1 = x2; */
  z11 = z21; z12 = z22; z13 = z23;

  if (x1 == 1) z11 = Z1; if (x1 == 2) z12 = Z2; if (x1 == 3) z13 = Z3;
  if (x2 == 1) z21 = Z1; else if (x2 == 2) z22 = Z2; else if (x2 == 3) z23 = Z3;

  assert((z11 + z12 + z13) == (z21 + z22 + z23)); }
```

How might we prove / disprove the assertion for ALL possible initial values of the variables?

(Welcome to symbolic methods!)
Concurrent program verification is harder!!

main(){
    int Z1, Z2, Z3;
    int x1, x2;
    int z11, z12, z13, z21, z22, z23;
    /* x1 = x2; */
    z11 = z21; z12 = z22; z13 = z23;

    if (x1 == 1) z11 = Z1; if (x1 == 2) z12 = Z2; if (x1 == 3) z13 = Z3;

    if (x2 == 1) z21 = Z1; else if (x2 == 2) z22 = Z2; else if (x2 == 3) z23 = Z3;
    assert((z11 + z12 + z13) == (z21 + z22 + z23)); }

(* More specifically: at each “grain boundary” of atomicity lurks a potential interleaving that was not considered...*)
Q: Why is concurrent program debugging hard?
A: Too many interleavings!!

Card Deck 0

0: __________
1: __________
2: __________
3: __________
4: __________
5: __________

Card Deck 1

0: __________
1: __________
2: __________
3: __________
4: __________
5: __________

• Suppose only the interleavings of the red cards matter

• Then don’t try all riffle-shuffles \( (12!) / ((6!) (6!)) = 924 \)

• Instead just try TWO shuffles of the decks  !!
The Growth of \((n.p)! / (n!)^p\)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th></th>
<th>Thread p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Unity / Murphi “guard / action” rules: \(n=1, p=R\) \(R!\) interleavings

- \(p = 3, n = 5\) \(\Rightarrow 10^6\) interleavings

- \(p = 3, n = 6\) \(\Rightarrow 17 \times 10^6\) interleavings

- \(p = 4, n = 5\) \(\Rightarrow 10^{10}\) interleavings
Ad-hoc Testing is INEFFECTIVE for thread verification!

Thread 1  ....  Thread p

1:  
2:  
3:  
4:  
...  
n:  
1:  
2:  
3:  
4:  
...  
n:  

- MUST reduce the number of interleavings considered
- Rigorous proof to back omitting interleavings
- DYNAMIC PARTIAL ORDER REDUCTION achieves both
#include <stdlib.h>     // Dining Philosophers with no deadlock
#include <pthread.h>    // all phils but "odd" one pickup their
#include <stdio.h>      // left fork first; odd phil picks
#include <string.h>     // up right fork first
#include <malloc.h>
#include <errno.h>
#include <sys/types.h>
#include <assert.h>
#define NUM_THREADS 3

pthread_mutex_t mutexes[NUM_THREADS];
pthread_cond_t conditionVars[NUM_THREADS];
int permits[NUM_THREADS];
pthread_t tids[NUM_THREADS];

int data = 0;

void * Philosopher(void * arg){
  int i;
  i = (int)arg;

  // pickup left fork
  pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
  while (permits[i%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, i%NUM_THREADS);
    pthread_cond_wait(&conditionVars[i%NUM_THREADS],&mutexes[i%NUM_THREADS]);
  }
  permits[i%NUM_THREADS] = 0;
  printf("P%d : get F%d\n", i, i%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

  // pickup right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  while (permits[(i+1)%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
    pthread_cond_wait(&conditionVars[(i+1)%NUM_THREADS],&mutexes[(i+1)%NUM_THREADS]);

    pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
    permits[(i+1)%NUM_THREADS] = 0;
    printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
    pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
  }
  permits[(i+1)%NUM_THREADS] = 1;
  printf("P%d : put F%d\n", i, (i+1)%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

  // data = 10 * data + i;
  printf("%d\n", i);
  fflush(stdout);

  // putdown right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  permits[(i+1)%NUM_THREADS] = 1;
  printf("P%d : put F%d\n", i, (i+1)%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
...Philosophers in PThreads

// putdown left fork
pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
permits[i%NUM_THREADS] = 1;
printf("P%d : put F%d \n", i, i%NUM_THREADS);
pthread_cond_signal(&conditionVars[i%NUM_THREADS]);
pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

// putdown right fork
pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
permits[(i+1)%NUM_THREADS] = 1;
printf("P%d : put F%d \n", i, (i+1)%NUM_THREADS);
pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

return NULL;

}

int main(){
    int i;

    for (i = 0; i < NUM_THREADS; i++)
        pthread_mutex_init(&mutexes[i], NULL);
    for (i = 0; i < NUM_THREADS; i++)
        pthread_cond_init(&conditionVars[i], NULL);
    for (i = 0; i < NUM_THREADS; i++)
        permits[i] = 1;

    for (i = 0; i < NUM_THREADS - 1; i++){
        pthread_create(&tids[i], NULL, Philosopher, (void*)(i) );
    }

    pthread_create(&tids[NUM_THREADS-1], NULL, 
          OddPhilosopher, (void*)(NUM_THREADS-1));

    for (i = 0; i < NUM_THREADS; i++){
        pthread_join(tids[i], NULL);
    }

    for (i = 0; i < NUM_THREADS; i++){
        pthread_mutex_destroy(&mutexes[i]);
    }
    for (i = 0; i < NUM_THREADS; i++){
        pthread_cond_destroy(&conditionVars[i]);
    }

    //printf(" data = %d \n", data);
    //assert( data != 201);
    return 0;
}
‘Plain run’ of Philosophers

gcc -g -O3 -o nobug examples/Dining3.c -L ./lib -lpthread -lstdc++ -lssl

% time nobug

P0: get F0
P0: get F1
0
P0: put F1
P0: put F0
P1: get F1
P1: get F2
1
P1: put F2
P1: put F1
P2: get F0
P2: get F2
2
P2: put F2
P2: put F0

real 0m0.075s
user 0m0.001s
sys 0m0.008s
...Buggy Philosophers in PThreads

// putdown left fork
pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
permits[i%NUM_THREADS] = 1;
printf("P%d : put F%d \n", i, i%NUM_THREADS);
pthread_cond_signal(&conditionVars[i%NUM_THREADS]);
pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

// putdown right fork
pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
permits[(i+1)%NUM_THREADS] = 1;
printf("P%d : put F%d \n", i, (i+1)%NUM_THREADS);
pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

return NULL;

}

int main(){
  int i;

  for (i = 0; i < NUM_THREADS; i++)
    pthread_mutex_init(&mutexes[i], NULL);
  for (i = 0; i < NUM_THREADS; i++)
    pthread_cond_init(&conditionVars[i], NULL);
  for (i = 0; i < NUM_THREADS; i++)
    permits[i] = 1;
  for (i = 0; i < NUM_THREADS-1; i++)
    pthread_create(&tids[i], NULL, Philosopher, (void*)(i));

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);

  for (i = 0; i < NUM_THREADS; i++)
    pthread_mutex_destroy(&mutexes[i]);
  for (i = 0; i < NUM_THREADS; i++)
    pthread_cond_destroy(&conditionVars[i]);

  //printf(" data = %d \n", data);

  //assert( data != 201);
  return 0;
}
‘Plain run’ of buggy philosopher .. bugs missed by testing

gcc -g -O3 -o buggy examples/Dining3Buggy.c -L ./lib -lpthread -lstdc++ -lssl

% time buggy

P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P1 : put F2
P1 : put F1
P2 : get F2
P2 : get F0
2
P2 : put F0
P2 : put F2

real 0m0.084s
user 0m0.002s
sys 0m0.011s
Jiggling Schedule in Buggy Philosopher.. 

```c
#include <stdlib.h>     // Dining Philosophers with no deadlock
#include <pthread.h>    // all phils but "odd" one pickup their
#include <stdio.h>      // left fork first; odd phil picks
#include <string.h>     // up right fork first
#include <malloc.h>
#include <errno.h>
#include <sys/types.h>
#include <assert.h>

#define NUM_THREADS 3

pthread_mutex_t mutexes[NUM_THREADS];
pthread_cond_t conditionVars[NUM_THREADS];
int permits[NUM_THREADS];
pthread_t tids[NUM_THREADS];

int data = 0;

void * Philosopher(void * arg){
    int i;
    i = (int)arg;

    // pickup left fork
    pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
    while (permits[i%NUM_THREADS] == 0) {
        printf("P%d : tryget F%d\n", i, i%NUM_THREADS);
        pthread_cond_wait(&conditionVars[i%NUM_THREADS],&mutexes[i%NUM_THREADS]);
    }
    permits[i%NUM_THREADS] = 0;
    printf("P%d : get F%d\n", i, i%NUM_THREADS);
    pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

    // pickup right fork
    pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
    while (permits[(i+1)%NUM_THREADS] == 0) {
        printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
        pthread_cond_wait(&conditionVars[(i+1)%NUM_THREADS],&mutexes[(i+1)%NUM_THREADS]);
    }
    permits[(i+1)%NUM_THREADS] = 0;
    printf("P%d : get F%d\n", i, (i+1)%NUM_THREADS);
    pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

    //printf("philosopher %d thinks \n",i);
    printf("%d\n", i);

    // data = 10 * data + i;
    fflush(stdout);

    // putdown right fork
    pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
    permits[(i+1)%NUM_THREADS] = 1;
    printf("P%d : put F%d\n", i, (i+1)%NUM_THREADS);
    pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
    pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
}
```

perms[i%NUM_THREADS] = 0;
printf("P%d : get F%d\n", i, i%NUM_THREADS);
pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

nanosleep (0) added here

// pickup right fork
pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
while (permits[(i+1)%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
    pthread_cond_wait(&conditionVars[(i+1)%NUM_THREADS],&mutexes[(i+1)%NUM_THREADS]);
}
permits[(i+1)%NUM_THREADS] = 0;
printf("P%d : get F%d\n", i, (i+1)%NUM_THREADS);
pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
```
‘Plain runs’ of buggy philosopher – bug still very dodgy ...

gcc -g -O3 -o
buggynsleep examples/Dining3BuggyNanosleep0.c
   -L ./lib -lpthread -lstdc++ -lssl

% buggysleep

P0 : get F0
P0 : sleeping 0 ns
P1 : get F1
P1 : sleeping 0 ns
P2 : get F2
P2 : sleeping 0 ns
P0 : tryget F1
P2 : tryget F0
P1 : tryget F2

First run deadlocked – second did not ...

buggysleep

P0 : get F0
P0 : sleeping 0 ns
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : sleeping 0 ns
P2 : get F2
P2 : sleeping 0 ns
P1 : tryget F2
P2 : get F0
2
P2 : put F0
P2 : put F2
P1 : get F2
1
P1 : put F2
P1 : put F1
Dijkstra’s famous quote

• Testing only confirms the presence of errors... never its absence

• MUCH MORE TRUE for concurrent software
Some terminology and an overview

• Concurrent includes Parallel (aka shared memory) and Distributed (aka message passing)

• In our research group, we have developed tools to verify PRACTICAL Parallel and Distributed software

• Currently for Pthreads (parallel) and MPI (distributed)

• This talk mainly focusses on Parallel (Pthreads / C) software verification
An important use of message passing

The scientific community is increasingly employing expensive supercomputers that employ distributed programming libraries....

(BlueGene/L - Image courtesy of IBM / LLNL)

...to program large-scale simulations in all walks of science, engineering, math, economics, etc.

(Image courtesy of Steve Parker, CSAFE, Utah)
Verification Realities

Code written using mature libraries
(MPI, OpenMP, PThreads, ...)

27
Verification Realities

Code written using mature libraries (MPI, OpenMP, PThreads, ...)

API calls made from real programming languages (C, Fortran, C++)
Verification Realities

- Code written using mature libraries (MPI, OpenMP, PThreads, ...)
- API calls made from real programming languages (C, Fortran, C++)
- Runtime semantics determined by realistic Compilers and Runtimes
Verification Realities

- Code written using mature libraries (MPI, OpenMP, PThreads, ...)
- API calls made from real programming languages (C, Fortran, C++)
- Runtime semantics determined by realistic Compilers and Runtimes

How best to verify codes that will run on actual platforms?
Traditional verification methods don’t work well in practice... e.g.

Classical Approach to Model Checking

Finite State Model of Concurrent Program

Check Properties
Background

Classical Approach to Model Checking

Finite State Model of Concurrent Program

Check Properties

DRAWBACK:

Extraction of Finite State Models for Realistic Programs is Difficult!
Background

Dynamic Model Checking

- Actual Concurrent Program
- Check Properties
Dynamic Model Checking

Check Properties

**MAIN ADVANTAGE:**

- Avoid model extraction which can be tedious and imprecise
- Program serves as its own model
- Reduce Complexity through Reduction of Interleavings (and other methods)
Background

Dynamic Model Checking in Practice

Actual Concurrent Program

One Specific Test Harness

Check Properties
Dynamic Model Checking in Practice

- Actual Concurrent Program

  One Specific Test Harness

  Check Properties

Background

- Need test harness in order to run the code

- We will Explore ALL RELEVANT INTERLEAVING for ONE test harness

- Conventional Testing tools cannot do this !!

- E.g. 5 threads 5 instructions each  $\rightarrow 10^{10}$ interleavings !!
Dynamic Model Checking in Practice

Check Properties

Actual Concurrent Program

One Specific Test Harness

Drawback:

- Need to consider all test harnesses

(but luckily, FOR MANY PROGRAMS, this number seems small -- e.g. Hypergraph partitioner – test harness is the initial graph to be partitioned)
Workflow of Parallel Software Verification tool “Inspect”

1. **Multithreaded C/C++ program**
   - Instrumentation
   - Instrumented program

2. **Thread library wrapper**

3. **Compile**
   - Executable
     - thread 1
     - thread n

4. **Scheduler**
   - Request/permit
   - Request/permit
void * Philosopher(void * arg){
  int i;
  i = (int)arg;
  ...
pthread_mutex_lock(&mutexes[i%3]);
  ...
  while (permits[i%3] == 0) {
    printf("P%d : tryget F%d\n", i, i%3);
    pthread_cond_wait(...);
  }
  ...
  permits[i%3] = 0;
  ...
  pthread_cond_signal(&conditionVars[i%3]);
  pthread_mutex_unlock(&mutexes[i%3]);
  return NULL;
}

void *Philosopher(void *arg)
{
  int i;
  pthread_mutex_t *tmp ;
  {
    inspect_thread_start("Philosopher");
    i = (int)arg;
    tmp = &mutexes[i % 3]; ...
    inspect_mutex_lock(tmp); ...
    while (1) {
      __cil_tmp43 = read_shared_0(&permits[i % 3]);
      if (! __cil_tmp32) {
        break;
      }
      __cil_tmp33 = i % 3; ...
      tmp___0 = __cil_tmp33; ...
      inspect_cond_wait(...);
    }
    __cil_tmp33 = i % 3; ...
    tmp___0 = __cil_tmp33; ...
    inspect_cond_wait(...);
  }
  ...
  write_shared_1(&permits[i % 3], 0); ...
  inspect_cond_signal(tmp___25);
  ...}
Inspect of nonbuggy and buggy Philosophers ..

```
...=== run 48 ===
P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : get F0
P0 : get F1
0
P1 : tryget F1
<-
Total number of runs: 48,
Transitions explored: 1814
Used time (seconds): 7.999327

=== run 1 ===
P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P0 : get F1
1
P1 : put F2
P2 : get F2
P2 : get F0
2
P2 : put F2
P2 : put F0
P0 : get F0
P0 : get F1
0
P1 : tryget F1
1
P1 : put F2
P1 : put F1
P2 : get F2
P2 : get F0
2
P2 : put F0
P2 : put F2

=== run 28 ===
P0 : get F0
P0 : get F1
0
P0 : tryget F1
P2 : get F2
P1 : tryget F2
P2 : tryget F0
1
P1 : put F2
P1 : put F1
P2 : get F2
P2 : get F0
2
P2 : put F0
P2 : put F2

--- Found a deadlock!!
(0, thread_start)
(0, mutex_init, 5)
(0, mutex_init, 6)
(0, mutex_init, 7)
(0, cond_init, 8)
(0, cond_init, 9)
(0, cond_init, 10)
(0, obj_write, 2)
(0, obj_write, 3)
(0, obj_write, 4)
(0, thread_create, 1)
(0, thread_create, 2)
(0, thread_create, 3)
(1, mutex_lock, 5)
(1, obj_read, 2)
(1, obj_write, 2)
(1, mutex_unlock, 5)
(2, mutex_lock, 6)
(2, obj_read, 3)
(2, mutex_lock, 7)
(3, obj_read, 4)
(3, obj_write, 4)
(3, mutex_lock, 5)
(3, mutex_lock, 7)
(2, mutex_lock, 7)
(2, obj_read, 4)
(2, mutex_unlock, 7)
(3, obj_read, 2)
(3, obj_write, 2)
(-1, unknown)

Total number of runs: 29,
killed-in-the-middle runs: 4
Transitions explored: 1193
Used time (seconds): 5.990523
```
The Growth of \((n.p)! / (n!)^p\) for Diningp.c illustrating the MAIN technology behind Inspect

- Diningp.c has \(n = 4\) (roughly)

- \(p = 3\) : We get 34,650 (loose upper-bound) versus 48 with DPOR

- \(p = 5\) : We get 305,540,235,000 versus 2,375 with DPOR

- DPOR really works well in reducing the number of interleavings !!

- Testing will have to exhibit its cleverness among \(3 \times 10^{11}\) interleavings
Obtaining and Running Inspect (Linux)

- [http://www.cs.utah.edu/~yuyang/inspect](http://www.cs.utah.edu/~yuyang/inspect)
- May need to obtain libssl-dev
- Need Ocaml-3.10.2 or higher
- Remove the contents of the “cache directory” autom4te.cache in case “make” loops

- `bin/instrument file.c`
- `bin/compile file.instr.c`
- `inspect –help`
- `inspect target`
- `inspect –s target`
DPOR produces alternate interleaving ONLY when there are “contentions”

First HAPPENS-BEFORE:

```c
pthread_mutex_lock(&mutex);
    A_count++;
pthread_mutex_unlock(&mutex);

pthread_mutex_lock(&lock);
    B_count++;
pthread_mutex_unlock(&lock);

pthread_mutex_lock(&mutex);
    A_count--;
pthread_mutex_unlock(&mutex);
```

Another “HAPPENS-BEFORE”

```c
pthread_mutex_lock(&mutex);
    A_count--;
pthread_mutex_unlock(&mutex);

pthread_mutex_lock(&lock);
    B_count++;
pthread_mutex_unlock(&lock);

pthread_mutex_lock(&mutex);
    A_count++;
pthread_mutex_unlock(&mutex);
```
The demos to follow will show that these ideas do work!!

DEMO 1: Seq C program verification

DEMO 2: Inspect

DEMO 3: Classical model checking