DPHPC: SPIN Tutorial
Recitation session

Locks

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qnode *pred = new qnode;
    qn->next = NULL;
    pred->next = qn;
    if (!pred->locked) {
        pred->locked = 1;
        while (pred->next == NULL);
    }
}

void unlock(qnode *lck, qnode *qn) {
    if (qn->next == NULL) {
        if (CAS(lck, qn, NULL)) return;
        while (qn->next == NULL);
        qn->next->locked = 0; // free next waiter
    }
}
```

Over-optimization can quickly lead to incorrect locks

Complicated

Difficult to optimize

How can make sure locks (or other distributed algorithms) are correct in practice?
How to check correctness?

- Common design flaws in designing distributed systems:
  - Deadlock
  - Livelock, starvation
  - Underspecification
    - Unexpected messages
  - Overspecification
    - Dead code
  - Constraint violations
    - Buffer overruns
    - Array bound violations
Model checking

- Model checking verifies a program by using software to analyze its state space

- Alternative: mathematical deductive methods
  - Constructing a proof requires mathematical insights and tenacity
  - The complexity depends on the algorithms itself

- Deductive proofs are more elegant and powerful, but model checking can be more practical

```plaintext
integer n=0

process P
  integer regP=0
  p1: load n into regP
  p2: increment regP
  p3: store regP into n
  p4: end

process Q
  integer regQ=0
  p1: load n into regQ
  p2: increment regQ
  p3: store regQ into n
  p4: end
```

How many states?

Space Explosion Problem

- The system is described as a State Transition Graph
- We have a combinatorial explosion of systems states

- How to handle it?

- Increase the abstraction

- Not all the states are actually reachable
  - Generate the states on the fly: only the ones that can be reached are generated

- Avoid to visit the same state multiple times
  - E.g., keep a hash table to index the visited states
SPIN – Introduction

- **SPIN**: Simple Promela Interpeter
  - Goal: analyze the logical consistency of concurrent systems
  - Concurrent systems are described in the modelling language called Promela

- **Promela**: Protocol/Process Meta Language
  - Allows dynamic creation of concurrent processes
  - Communication via message passing can be
    - Synchronous (aka rendezvous)
    - Asynchronous (aka buffered)
  - C-like language
  - Enables you to model a finite-state system

- **Warning**: If that description is “too far off” from our code, we risk specifying the wrong state machine!
Promela Model

Promela model consist of:
• Type declarations
• Channel declarations
• Variable declarations
• Process declarations
• [init process]

A process type (proctype) consist of
• a name
• a list of formal parameters
• local variable declarations
• body
Promela - Processes

- Identified by the `proctype` keyword
- Can be executed concurrently
- You can create multiple processes of the same type

Each process has its own local state defined by:
- Program counter
- Local variables

Communication between processes:
- Shared variables
- Channels

Processes can be created using the `run` keyword:
- It returns the pid of the created process
- Can be called at any point
Promela: Hello World

active proctype Hello() {
  printf("Hello process, my pid is: %d\n", _pid);
}

init{
  int lastpid;
  printf("init process, my pid is: %d\n", _pid);
  lastpid = run Hello();
  printf("last pid was: %d\n", lastpid);
}

$ spin -n2 hello.pr
init process, my pid is: 1
  last pid was: 2
Hello process, my pid is: 0
  Hello process, my pid is: 2
3 processes created
Promela: Variables and Types

- **Types: 5 basic types**
  - bit, bool, byte, short, int

- **Arrays**
  - byte a[27];

- **Records (structs)**
  - typedef Record{
    short f1;
    byte f2;
  }
  Record rr;
  rr.f1 = ...

- Global and local variables are initialized to 0 by default

```c
int ii;
bit bb;
bb=1;
ii=2;
short s=-1;
typedef Foo {
    bit bb;
    int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;
ii*s+27 == 23;
printf("value: %d", s*s);
```
Promela: Statements

- The body of a process consists of a sequence of statements.
  - Executable: the statement can be executed immediately
  - Blocked: it cannot be executed

- Assignments are always executable

- An expression is executable only if it evaluates to non-zero
  - $2 < 3$ always executable
  - $x < 27$ executable only if $x < 27$
  - $3 + x$ executable only if $x \neq -3$

- The assert(<expr>) statement is always executable
  - SPIN exits with an error if an assert evaluates to 0
  - Used to check if properties hold
Semantic

- Pamela processes are executed **concurrently** and scheduled in a **non-deterministic** fashion
- Execution of statements of different processes is interleaved
  - All statements are atomic
- Each process may have multiple actions ready to be executed
  - Only one is non-deterministically chosen to be executed
Promela: Mutual Exclusion? (1)

bit flag; /* signal entering/leaving the section */
byte mutex; /* # procs in the critical section. */

proctype P(int i) {
    flag != 1;
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init { run P(0); run P(1); run monitor(); }
Promela: Mutual Exclusion? (2)

```
bit x, y;         /* signal entering/leaving the section */
byte mutex;  /* # of procs in the critical section. */

active proctype A() {
    x = 1;
    y == 0;
    mutex++; mutex--;
    x = 0;
}

active proctype B() {
    y = 1;
    x == 0;
    mutex++; mutex--;
    y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
```

Both processes can pass execute x=1 and y=1 “at the same time”...
### PROMELA Semantics: if

```promela
if
:: choice_1 -> stat1.1; stat1.2; stat1.3; ...
:: choice_2 -> stat2.1; stat2.2; stat2.3; ...
:: ...
:: choice_n -> statn.1; statn.2; statn.3; ...
:: else -> skip
fi;
```

- If there is at least one choice (guard) executable, the if statement is executable and SPIN non-deterministically chooses one of the executable choices.
- The “else” choice is executable iff no other choices are.
- If no choice is executable, the if-statement is blocked.
With respect to the choices, a do-statement behaves in the same way as an if-statement.

However, instead of ending the statement at the end of the chosen list of statements, a do-statement repeats the choice selection.

The (always executable) break statement exits a do-loop statement and transfers control to the end of the loop.
PROMELA Semantics: Communication

- Communication between processes is via typed channels

```
chan <name> = [<dim>] of {<t1>,<t2>,...,<tn>};
```

- A channel can be synchronous (dim=0) or asynchronous (dim>0)
  - In the first case, synchronization is needed
  - In the second, the channels act like a FIFO-buffer

Example:
```
mtype = {MSG, ACK};
chan toS = [2] of {mtype, bit};
```
PROMELA Semantics: Communication

- Communication between processes is via typed channels

```
chan <name> = [<dim>] of {<t1>,<t2>,...,<tn>};
```

- Sending:
  - `ch ! <expr1>, <expr2>, ..., <exprn>;`
    
    The values of `<expri>` should correspond with the types of the channel declaration.
    
    A send statement is executable if the channel is not full.

- Receiving
  - `ch ? <var1>, <var2>, ..., <varn>;`
    
    If the channel is not empty, the message is fetched from the channel.
  
  - `ch ? <const1>, <const2>, ..., <constn>;`
    
    If the channel is not empty and the message at the front of the channel evaluates to the individual `<consti>`, the statement is executable and the message is removed from the channel.

Rendezvous communication (dim==0):

A send `ch!` is executable only if there is a corresponding receive `ch?` that can be executed simultaneously.

Message Matching

- `<vars>` and `<consts>` can be mixed.
Using spin

- Random simulation mode: debugging/testing. Randomly resolves non-determinism
  \[\text{spin } -n\text{<SEED> model.pr #fix the seed to reproduce scenarios}\]

- Guided simulation mode (-i): non-determinism solved by the user

- Verification mode: analyze all the reachable states
  \[\text{spin } -a \text{ model.pr}\]
  \[\text{gcc } -O2 -o \text{ pan pan.c}\]
  \[./\text{pan #generates trail file if things go wrong}\]
  \[\text{spin } -t -p \text{ model.pr}\]

  Generates a verifier in C code, so that compiler can optimize it, then exhaustively searches all possible states. It can still be slow/eat all your memory.
Demo: Dining philosophers

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, **a philosopher can only eat spaghetti when they have both left and right forks.** Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

*Source: https://en.wikipedia.org/wiki/Dining_philosophers_problem*
Assignment

Hippie problem:
4 Hippies want to cross a bridge. The bridge is fragile, it can only crossed by <= 2 people at a time with a torchlight. The hippies have one torchlight and want to reach the other side within one hour. Due to different degrees of intoxication they require different amounts of time to cross the bridge: 5, 10, 20 and 25 minutes. If a pair crosses the bridge, they can only move at the speed of the slower partner.