Algorithmic Verification

Comp4151
Lecture 1-B
Ansgar Fehnker

Algorithmic Verification

The software crisis (and hardware as well)

- Computer become more powerful (Moore's law)
- The quality of programs cannot keep up
  - Up to 80% of all software development time is spent on locating and correcting defects
  - About 70% of all cost in hardware design go to verification and validation
  - Rework due to defects identified accounts for between 40% and 50% of total project cost

"When there were no computers programming was no problem. When we had a few weak computers, it became a mild problem. Now that we have gigantic computers, programming is a gigantic problem." (Edsger Dijkstra)

Algorithmic verification

Solution: Give Proof

Computer Aided Verification

- *Theorem proving (mostly semi-automatic)*
- Model checking (mostly automatic)

"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness." (Edsger Dijkstra)

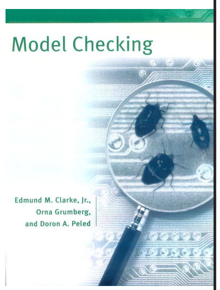
Model checking

The basic idea

- Given a model of the system
  - Kripke structre, FSM, automaton, Petri net, …
- Given a formal specification
  - LTL, CTL, mu-calculus, …
  - another simpler model
- Calculate whether model satisfies specification

No proofs. (But you need math to build a model checker)
Fast (compared to other rigorous approaches)
Gives counter-examples (help with debugging, too)

Reading



Model Checking
Edmund Clarke, Orna Grumberg and Doron Peled
MIT Press 2000
This textbook can be obtained at the UNSW bookshop

Model checking

Today

Application examples
- program analysis
- mutual exclusion
- wireless network
- scheduling

Program Analysis

Semantic analysis
- Check if the program does what it is supposed to do.
- Requires an understanding of what the program actually means

Syntactic analysis
- Look for common programming constructs that cause bugs
- Except for the CFG, the program is just syntax

What is worng with this tautology?

Program Analysis

What is wrong with this program?

```
int main()
{
  int i,x;  // Variable declaration.
  int lock1;

  lock1 = 0;// lock should be 0 at the end.
  for(i=0; i <  10; i++) {

    if(x == 12) {.
      lock1--;
    }

    if(i > 8) {
      i=i-lock1++ ;
    }
  } // End for().
}
```

Problems the compiler doesn't find.

Program Analysis

What is wrong with this program?

```
int main()
{
  int i,x;  // Variable declaration.
  int lock1;

  lock1 = 0;// lock should be 0 at the end.
  for(i=0; i <= 10; i++) {

    if(x == 12) {.
      lock1--;
    }

    if(i > 8) {
      i=i-lock1++ ;
    }
  } // End for().
}
```

Problems the compiler doesn't find.

Program Analysis

Semantic properties

- Counter `i` should exceed 10 eventually
- `lock1` should be zero at the end of the program
- After each `lock++` there should be a `lock--`

Syntactic properties

- A variable cannot be used until it is initialized.
- The loop counter should not be modified in a loop.
- After each `lock++` there should be a `lock--`

Concurrent Systems

- Concurrent systems consists of subsystems, components, modules, agents, …
- The components have to
  - interact in a correct and timely manner
  - cooperate to ensure functionality of the system
  - compete for shared resources
- Concurrency bugs are often an unintended side effect of parallelism and shared access to resources.

Concurrent Systems

Example

- 2 threads. They share variable **x** and have local variables **y** and **z**, respectively.
- **x** has initially value 2

```
P1          ||        P2
y = x;                z = x;
x = y+1;              x = z+1 ;
```

- What is the result when the processes complete?

Concurrent Systems

Example

- 2 threads. They share variable **x** and **y**
- **x** has initially value 2
- **y** has initially value 1

```
P1              ||    P2
y = x++;              y = x++;
x = x+(y++);          x = x+(y++);
```

- What is the result when the processes complete?

Non-determinism

- Most sequential programs are *deterministic*
- At any point there is exactly one possible next step
- Concurrent systems and programs are often non-deterministic
- There might be multiple components that can take the next step
- Model checking explores all possibility to interleave steps

Concurrent Systems

Example from Wikipedia

```
global integer A = 0;

task Received(){         ||  task Timeout(){
 A = A + 1;              ||   if (A is divisible by 2) {
 print "RX";             ||        print A;
}                        ||   }
                         ||  }
```

Will task Timeout ever print an odd number?

Concurrent Systems

Race Conditions

Given a system with multiple processes that share a resource.
A race condition is a situation in which the result or output depends on the relative timing of events.

Concurrent Systems

Critical Section

A piece of code that in which a process needs exclusive access to a resource.

A most one process should be in the critical section at the same time.

Mutual Exclusion

A system guarantees mutual exclusion if at most one process can be in a critical section.

Important for OS. Shared memory, disc, printer, …

Mutual Exclusion

Towards a solution: Flag critical sections

```
global integer A = 0;

task Received(){      task Timeout(){
 *** begin CS ***      *** begin CS ***
 A = A + 1;            if (A is divisible by 2) {
 print "RX";                print A;
 *** end CS   ***      }
}                      *** end CS   ***
                      }
```

Mutex algorithms are arbiter to ensure exclusive access to CS

Mutual Exclusion

Potential problems for mutex algorithms

deadlock: Two or more processes wait for the another to complete a task

livelock: Two or more processes change their state in reaction to the change in states of the other processes, but none of them really progresses.

starvation: A process is ready to excute a task, but never gets the chance to take a step.

Mutual Exclusion

Dining Philosophers Problem

- **N** philosophers having spaghetti for dinner
- Only **N** forks available on the dining table
- Every philosopher needs two forks for dining
- Give a solution that avoids
  - Deadlock
  - Livelock, and
  - Starvation

Mutual Exclusion

A solution

Introduce a shared variable to tell whose turn it is

`var turn : 0..1;`

P0	P1
`while turn ≠ 0`	`while turn ≠ 1`
`do {nothing};`	`do {nothing};`
`"critical section";`	`"critical section";`
`turn:=1;`	`turn:=0;`

Ensures mutex, but processes have to alternate

Mutual Exclusion

A solution

Introduce a flag for each process

`var flag : array[0..1] of bool;`

P0	P1
`while flag[1]`	`while flag[0]`
`do {nothing};`	`do {nothing};`
`flag[0]:=true;`	`flag[1]:=true;`
`"critical section";`	`"critical section";`
`flag[0]:=false;`	`flag[1]:=false;`

Does not ensure mutex

Mutual Exclusion

A solution

Introduce a flag for each process

`var flag : array[0..1] of bool;`

P0	P1
`flag[0]:=true;`	`flag[1]:=true;`
`while flag[1]`	`while flag[0]`
`do {nothing};`	`do {nothing};`
`"critical section";`	`"critical section";`
`flag[0]:=false;`	`flag[1]:=false;`

Does ensure mutex, but may lead to deadlock

Mutual Exclusion

A solution: Peterson Algorithm (1981)

Use flag and turn variable

`var flag : array[0..1] of bool;`

P0	P1
`flag[0]:=true;`	`flag[1]:=true;`
`turn:=1;`	`turn:=0;`
`while flag[1]&turn=1`	`while flag[0]&turn=0`
`do {nothing};`	`do {nothing};`
`"critical section";`	`"critical section";`
`flag[0]:=false;`	`flag[1]:=false;`

Ensures mutex and no deadlock, livelock or starvation

Fairness

- Many algorithms are only correct given additional *fairness assumptions.*
- A fairness assumption means that each process will get a chance to take part in the protocol
- This rules out the following situation
  - A process starves because it never takes a step, although the step is continuously/infinitely often enabled.

Fairness

Weak fairness

An infinite execution is weak fair if a transition occurs infinitely often if is only finitely often disabled.

Strong fairness

An infinite execution is strongly fair if a transition occurs infinitely often if is infinitely enabled.

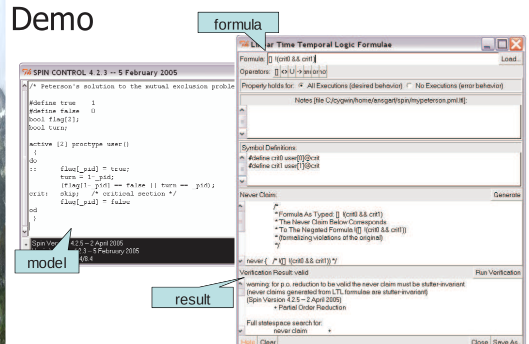
Mutual Exclusion

Observation

- Many flawed solutions to the mutual exclusion problem
- Reasoning over mutual exclusion is difficult
- You want to have proof that it works.
- Model checking can show correctness.

Spin demo

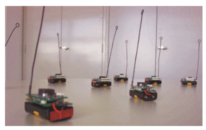
Demo



Protocols

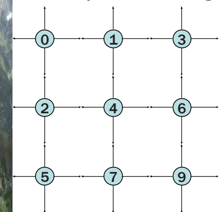
Example: Wireless sensor networks

Aggregate of small, portable devices
- *battery-operated* computing power
- distributed *gathering of sensor information*
- *wireless* communications
- *multi-hop* communication



Example

Example: Flooding and Gossiping Protocols

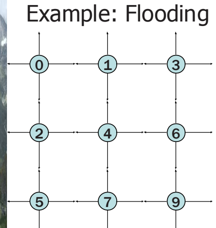


Flooding protocol
- listen to medium
- if you receive a message
  - send message
- go to sleep

Properties of flooding
- simple
- used for routing
- redundant
- prone to collisions
- inefficient

Example

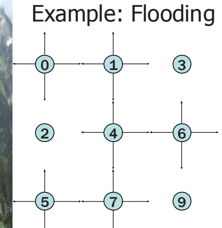
Example: Flooding and Gossiping Protocols



Gossiping protocol
- listen to medium
- if you receive a message
  - send message with probability p
- go to sleep

Example

Example: Flooding and Gossiping Protocols



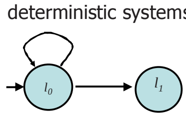
Gossiping protocol
- listen to medium
- if you receive a message
  - send message with probability p
- go to sleep

Properties of gossiping
- still simple
- reduces redundancy
- reduces collisions
- improved efficiency

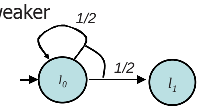
Probabilistic Choice

Probability and Non-determinism

- Many probabilistic models allow for both probabilistic choice and non-deterministic choice.
- Probabilism introduces a notion of fairness absent in non-deterministic systems.
- Fairness restrictions/assumption on non-deterministic systems are weaker



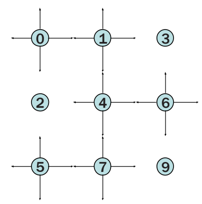
The non-deterministic system can stay in l_0 forever



The probabilistic systems stays in l_0 forever with probability 0

Example

Example: Flooding and Gossiping Protocols

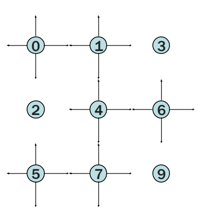


The model includes probabilistic choice

- A node sends with probability p
- A node sends not with probability 1-p
- An implementation of a node that always sends is incorrect
- An implementation of a node that never sends is incorrect
- An implementation of a node that alternates sending and "not-sending" is incorrect

Example

Example: Flooding and Gossiping Protocols



The network includes also non-deterministic choice

- If node 1 and 2 receive a message then
- either node 1 takes the next step
- or node 2 takes the next step
- An implementation in which node 1 always takes priority is correct
- An implementation in which node 2 always takes priority is correct
- An implementation in which node 1 and 2 alternate is correct

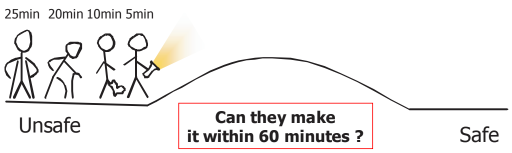
Example

- Prism demo

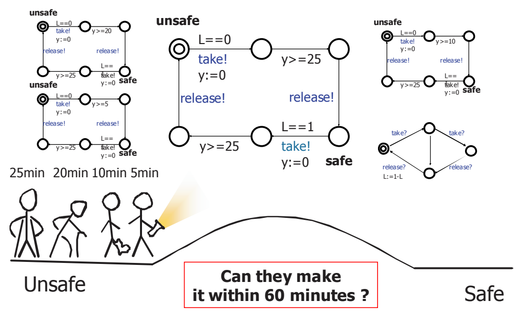
No time for this one ☺

Scheduling

Model checking can be used to solve scheduling problems

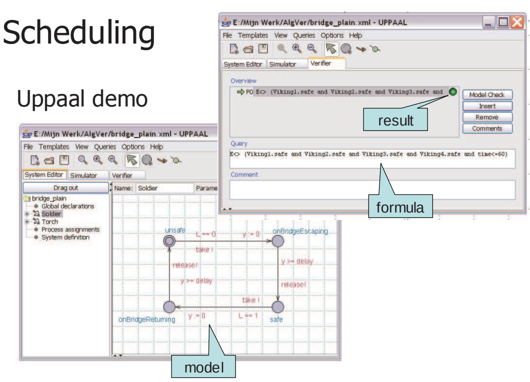


Scheduling



Scheduling

Uppaal demo



Summary

Model checking can be used to tackle a varienty of problems

- Program verification
- Concurrent systems
- Protocols
- Scheduling
- ....

Summary

Different models checkers differ in

- Modelling language
- Specification logic
- Model checking algorithm

Next week

- The fundamentals of modelling systems