Model Checking

CS511

Foundations of Model-Checking

Linear Temporal Logic

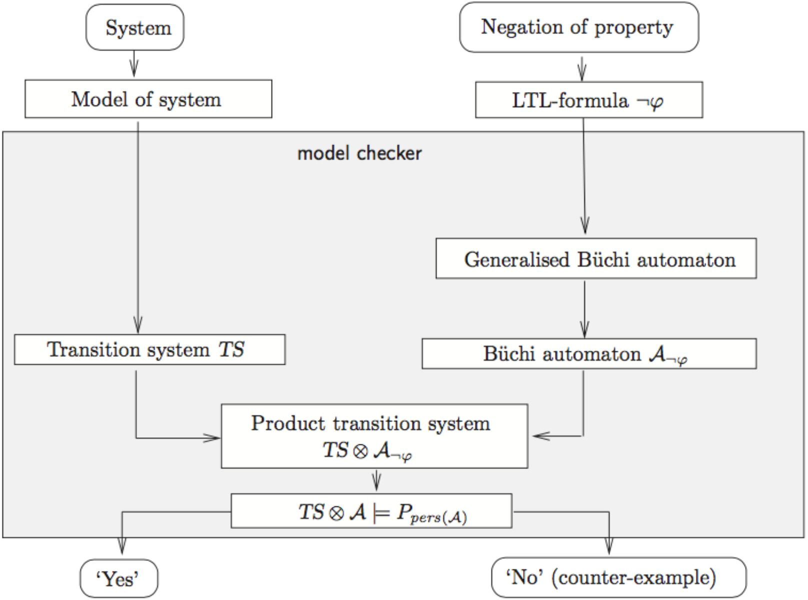
Using LTL Formula to Specify and Verify Properties in Spin

Model-Checking

An introduction to the foundations of Model-Checking

- We shall use the whiteboard
- Principles of Model Checking, Christel Baier, Joost-Pieter Katoen and Kim Guldstrand Larsen, The MIT Press, 2008.

Map


System
Negation of property
Model of system
LTL-formula $\neg\varphi$
model checker
Generalised Büchi automaton
Transition system TS
Büchi automaton $\mathcal{A}_{\neg\varphi}$
Product transition system
$TS \otimes \mathcal{A}_{\neg\varphi}$
$TS \otimes \mathcal{A} \models P_{pers(\mathcal{A})}$
'Yes'
'No' (counter-example)

Foundations of Model-Checking

Linear Temporal Logic

Using LTL Formula to Specify and Verify Properties in Spin

LTL

On the board

Foundations of Model-Checking

Linear Temporal Logic

Using LTL Formula to Specify and Verify Properties in Spin

States in Spin

- Spin uses DFS to explore the complete state space of the model
  - Old states are stored on a stack
- States are:
  - Stored in a hash table
  - Generated on-the-fly
  - Represented using state vectors (compressed)
    - global variables
    - contents of the channels
    - for each process in the system: local variables and process counter

Spin Verification Report

```
(Spin Version 6.4.6 -- 2 December 2016)
        + Partial Order Reduction
Full statespace search for:
        never claim             + (ltl_0)
        assertion violations    + (if within scope of claim)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by never claim)
State-vector 28 (size of a state) byte, depth
reached 149 (longest path), --- errors: 0 ---
      484 (total number of states) states, stored
      937 states, matched
     1421 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)
Stats on memory usage (in Megabytes):
    0.026       equivalent memory usage for states (stored*(State-
    0.286       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.107       memory used for DFS stack (-m2000)
  128.302 (memory used) total actual memory usage
```

Modalities in Spin

- $\Box\phi$ is written `[]` ϕ
- $\Diamond\phi$ is written `<>` ϕ
- $\phi_1 \mathcal{U} \phi_2$ is written ϕ_1 `U` ϕ_2 (ϕ_1 is true until ϕ_2 becomes true)
- $\neg\phi$ is written `!`ϕ
- $\phi_1 \wedge \phi_2$ is written ϕ_1 `&&` ϕ_2
- $\phi_1 \vee \phi_2$ is written ϕ_1 `||` ϕ_2

Invariance

- Properties of the form

$$\Box\phi$$

"ϕ holds at every state"

- A particular case of safety properties
- They can also be checked using assertions, like we have already done
- Example:

```
[] !(csp && csq)
```

Safety Properties

```
bool wantP = false, wantQ = false;
bool csp, csq;

active proctype P() {
  do :: wantP = true;
        !wantQ;
        csp = true; csp = false;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        csq = true; csq = false;
        wantQ = false
  od
}
```

```
spin -a -f '[]!(csp && csq)' third-safety.pml
gcc -DSAFETY -o pan pan.c
./pan
```

No errors reported

Avoiding Ghost Variables

```
bool wantP = false, wantQ = false;
#define mutex !(P@cs && Q@cs)

active proctype P() {
  do :: wantP = true;
        !wantQ;
cs:     wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
cs:     wantQ = false
  od
}
```

```
spin -a -f '[]mutex' third-safety-no-ghost.pml
gcc -DSAFETY -o pan pan.c
./pan
```

Never Claims

- The name given by Spin to the NBA constructed from ϕ (called $\mathcal{A}_{\neg\psi}$ in the previous slide).
- Spin transforms a formula in temporal logic into a PROMELA construct called a never claim.
- Just as a PROMELA program specifies an automaton whose state space is searched by the verifier (its TS), so a never claim specifies an automaton whose state space is searched in parallel with the one that is defined by the PROMELA program.

Never Claims

```
$ spin -f '![]mutex'
never {    /* ![]mutex */
T0_init:
        do
        :: atomic { (! ((mutex))) -> assert(!(! ((mutex)))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

```
i$ spin  -f '[]!(csp && csq)'
never {    /* []!(csp && csq) */
accept_init:
T0_init:
        do
        :: (! ((csp && csq))) -> goto T0_init
        od;
}
```

Liveness Properties

$$\Diamond\phi$$

"ϕ will eventually hold"

- Called a liveness property because it specifies that something "good" eventually happens in the computation.

Liveness Properties

- They are important
  - Safety properties are vacuously satisfied by an empty program that does nothing!
- For example, a solution to the critical section problem in which neither process tries to enter its critical section trivially fulfils the correctness properties of mutual exclusion and absence of deadlock:

```
start:
do
:: printf("Noncritical section\n");
   goto start;
   wantP = true; /* Try to enter the critical section */
   printf("Critical section\n")
od
```

Attempt IV with Back Out

```
global boolean wantP = false;
global boolean wantQ = false;
```

```
thread P: {
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
thread Q: {
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence deadlock: Yes
- Free from starvation: No

Attempt IV with Back Out

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
     wantP = false
        od
}

active proctype Q() {
   do
   :: wantQ = true;
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
   wantQ = false
   od
}
```

Starvation in Attempt IV

```
s0 = (5. wantP=1, 18. wantQ=1, 0,0)
s1 = (5. wantP=1, 20. wantP,0,1)
s2 = (5. wantP=1, 25. wantQ=0, 0,0)
s3 = (7. wantQ,   25. wantQ=0,1,1)
s4 = (8. wantP=0, 25. wantQ=0,1,1)
s5 = (9. wantP=1, 25. wantQ=0,0,1)
s6 = (9. wantP=1, 18. wantQ=1,0,0)
s7 = (9. wantP=1, 20. wantP,0,1)
s8 = (9. wantP=1, 25. wantQ=0,0,1)
s9 = (7. wantQ,   25. wantQ=0,1,1)
```

P starves

Verifying Liveness in Spin

```
bool wantP = false, wantQ = false;
bool csp;
ltl p1 {<> csp}
active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
     csp = true; csp = false;
     wantP = false
   od
}
active proctype Q() {
   do
   :: wantQ = true;
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
      wantQ = false
   od
}
```

Starvation

`<>csp`

- Expresses absence of starvation for process P.
- Verification similar to what we did before (i.e. safety properties), except that it must be performed in a mode called searching for *acceptance cycles*
- *Weak fairness*, explained soon, must also be specified when this program is verified.

Note: Must set jSpin to `Acceptance` so that it works in `Acceptance cycle` mode rather than `Safety`

```
pan:1: acceptance cycle (at depth 14)
pan: wrote attemptIV.pml.trail
(Spin Version 6.4.6 -- 2 December 2016)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             + (p1)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)
State-vector 36 byte, depth reached 51, --- errors: 1 ---
       26 states, stored (52 visited)
       18 states, matched
       70 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
Stats on memory usage (in Megabytes):
    0.002       equivalent memory usage for states (stored*(State-
    0.289       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.107       memory used for DFS stack (-m2000)
  128.302       total actual memory usage
pan: elapsed time 0 seconds
```

Command Line Alternative

Two options:

- Do not include the line `ltl p1 <> csp` and run:

```
spin -a -f '!<>csp' fourth-liveness.pml
gcc -o pan pan.c
pan -a -f
```

  - Note absence of `-DSAFETY` option

- Include the line `ltl p1 <> csp` and run:

```
spin -a fourth-liveness.pml
gcc -o pan pan.c
pan -a -f
```

Verifying Liveness in Spin

That liveness does not hold is indicated by the third line below:

```
1 warning: for p.o. reduction to be valid the never claim must be st
2 (never claims generated from LTL formulae are stutter-invariant)
3 pan:1: acceptance cycle (at depth 14)
4 pan: wrote fourth-liveness-2.pml.trail
5
6 (Spin Version 6.4.5 -- 1 January 2016)
7 Warning: Search not completed
8         + Partial Order Reduction
9
10 Full statespace search for:
11         never claim             + (never_0)
12         assertion violations    + (if within scope of claim)
13         acceptance   cycles     + (fairness enabled)
14         invalid end states      - (disabled by never claim)
15
16 State-vector 36 byte, depth reached 51, errors: 1
17 ...
```

Executing the Offending Trail

- Use "Guided" execution in jSpin

```
ltl p1: <> (csp)
starting claim 2
using statement merging
Never claim moves to line 4      [(!(csp))]
1 Q:1    1)  wantQ = 1
Process Statement              wantQ
1 Q:1    1)  else              1
1 Q:1    1)  wantQ = 0         1
0 P:1    1)  wantP = 1         0
Process Statement              wantP      wantQ
1 Q:1    1)  wantQ = 1         1          0
1 Q:1    1)  wantP             1          1
0 P:1    1)  wantQ             1          1
<<<<<START OF CYCLE>>>>>
...
```

- `START OF CYCLE` indicates that the subsequent states form a cycle that can be repeated indefinitely
- Absence of column for `csp` means `csp` has never been assigned to and hence that starvation occurs in this computation

Revisiting Attempt IV

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
     wantP = false
        od
}

active proctype Q() {
   do
   :: wantQ = true;
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
   wantQ = false
   od
}
```

Fairness

Is the following computation a counterexample for the property of absence of starvation?

```
s0 = (5. wantP=1, 18. wantQ=1, 0, 0)
s1 = (5. wantP=1, 20. wantP, 0, 1)
s2 = (5. wantP=1, 25. wantQ=0, 0, 1)
s3 = (5. wantP=1, 18. wantQ=1, 0, 0)
```

- It is a counterexample to a claim that `<>csp` is true
- But is unsatisfactory because it doesn't give process P a "fair" chance to try to enter its critical section.
  - A computation is weakly fair if and only if the following condition holds: if a statement is always executable, then it is eventually executed as part of the computation.
- The computation described above is not weakly fair: Although like all assignment statements, `5. wantP = true` is always executable, it is never executed in the computation.

Counterexample produced without Weak Fairness

```
ltl p1: <> (csp)
starting claim 2
using statement merging
<<<<<START OF CYCLE>>>>>
Never claim moves to line 4     [(!(csp))]
1 Q:1   1)  wantQ = 1
Process Statement           wantQ
1 Q:1   1)  else            1
1 Q:1   1)  wantQ = 0       1
spin: trail ends after 6 steps
#processes: 2
  6:    proc  1 (Q:1) attemptIV.pml:16 (state 11)
  6:    proc  0 (P:1) attemptIV.pml:5 (state 13)
MSC: ~G line 3
  6:    proc  - (p1:1) _spin_nvr.tmp:3 (state 3)
2 processes created
Exit-Status 0
```

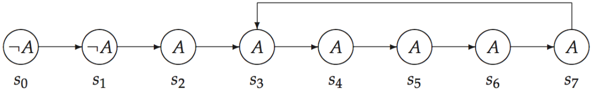
More on Fairness

```
byte n = 0;
bool flag = false;

active proctype p() {
        do
        ::  flag -> break;
        ::  else -> n = 1 - n;
        od
}

active proctype q() {
        flag = true
}
```

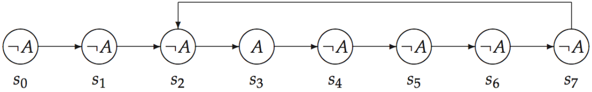
- The assignment in Q is always enabled, so in a weakly fair computation it will eventually be executed
- This causes the loop in process P to terminate.
- Thus the correctness property "the program always terminates" holds if and only if computations are required to be weakly fair.

More LTL Formulae

- $\Diamond\Box\phi$ expresses a latching property: ϕ may not be true initially in a trace, but eventually it becomes true and remains true:



- $\Box\Diamond\phi$ expresses the property that ϕ is true infinitely often: ϕ need not always be true, but at any state in the trace s, ϕ will be true in s or in some state that comes after s:



Example

```
active proctype P() {
do
  :: /* Try to enter critical section */
      csp = true;
      csp = false;
      /* Leave critical section */
od
}
```

- If an algorithm is free from starvation, we can verify the program for the temporal formula `[]<>csp`.
- This states that a process can enter its critical section repeatedly.