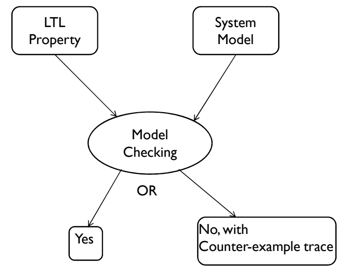
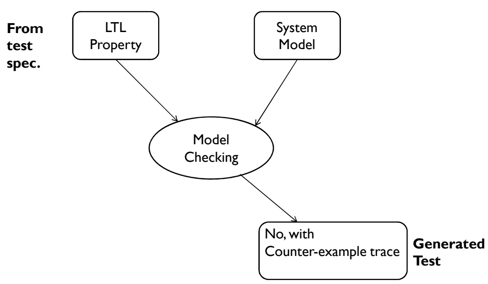
Model Checking
CS 4271

Abhik Roychoudhury
http://www.comp.nus.edu.sg/~abhik

Model Checking



Recap: Model Checking for model-based testing



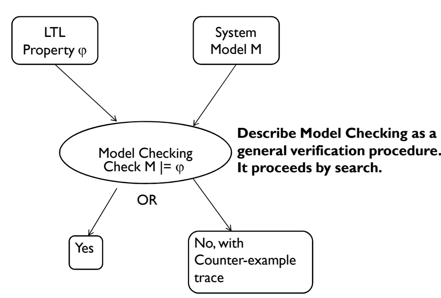
Encoding test specifications

- Def. 1
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a **contiguous** subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - Define $prop_\sigma = \mathbf{F}(e_1 \wedge \mathbf{X}(e_2 \wedge \mathbf{X}(\ldots \mathbf{X}(e_k)\ldots)))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg\, (\vee_{\sigma \in Lin(M)}\, prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Encoding test specifications

- Def. 2
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - $n_\sigma = \neg(e_1 \vee e_2 \vee \ldots \vee e_k)$
      - $prop_\sigma = (n_\sigma\ \mathbf{U}\ (e_1 \wedge \mathbf{X}(n_\sigma\ \mathbf{U}(e2 \wedge \mathbf{X}(\ldots\ \mathbf{X}(n_\sigma\ \mathbf{U}\ ek)\ldots))))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg\, (\vee_{\sigma \in Lin(M)}\, prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Model Checking



LTL Model Checking – does M |= φ

1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

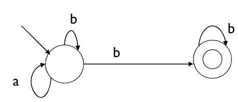
Recap: finite-state automata

- $A = (Q, \Sigma, Q_0, \rightarrow, F)$
  - Q is a finite set of states
  - Σ is a finite alphabet
  - $Q_0 \subseteq Q$ is the set of initial states
  - $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation
  - $F \subseteq Q$ is the set of final states.
- What is the Language of such an automaton?

Recap: finite-state automata

- Regular languages:
  - Accept any finite-length string $\sigma \in \Sigma^*$ which ends in a final state.
- ω-regular languages:
  - Accept any infinite-length string $\sigma \in \Sigma^\omega$ which visits a final state infinitely many times.
- Set of strings accepted = *Language* of the automata.

Finite automata



- Meaning as a regular language
  - $(a+b)^*b^+$
  - All finite length strings ending with b
- Meaning as a ω-regular language
  - All infinite length strings with finitely many a

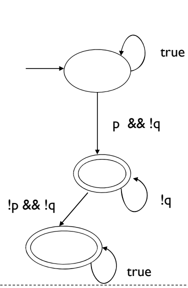
LTL properties to automata

- Given a LTL property p
  - we want to convert p to an automata A_p s.t.
  - Language(A_p) = strings / traces satisfying p
- LTL properties are checked over infinite traces.
  - Given an infinite trace σ and a LTL property p, we can check whether σ |= p
- To convert LTL properties to finite-state automata, consider automata accepting inf.-length traces.
  - Language(A_p) is ω-regular, not regular.

LTL properties to automata

- Given a LTL property φ
  - We convert it to a ω-regular automata A_φ
- Language(A_φ) = $\{\sigma \mid \sigma \in \Sigma^\omega \wedge \sigma \models \varphi\}$
  - Language(A_φ) is defined as per the ω-regular notion of string acceptance. It accepts inf. length strings.
  - All infinite length strings satisfying φ form the language of A_φ
  - Whether an infinite length string satisfies φ (or not) is defined as per LTL semantics.

Example: LTL property to automata



Represents negation of the LTL property

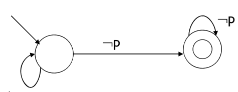
G (p ⇒ (p U q))

Recall: LTL Model Checking

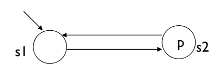
1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata A $_{\neg\varphi}$ such that
   - Language(A $_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product M × A $_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product M × A $_{\neg\varphi}$ i.e. check Language(M × A $_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

Example: Verify GFp

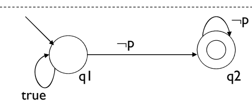
- Construct negation of the property
  - ¬GFp ≡ FG¬p
- Construct automata accepting infinite length traces satisfying FG¬p



Product Automata

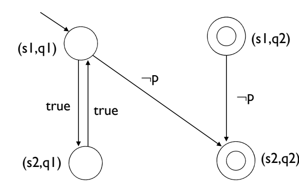


(i) System Model M



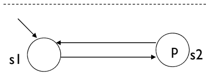
(ii) Property Automata A

M |= GFp

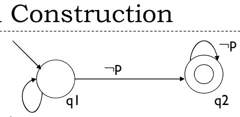


(iii) Product Automata M × A

Product Automata Construction

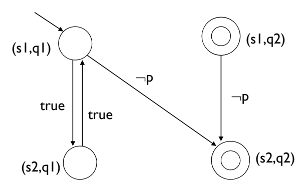


(i) System Model M



(ii) Property Automata A

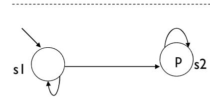
Note that s1 |= ¬p



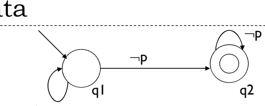
M |= GFp

(iii) Product Automata M × A

Product Automata

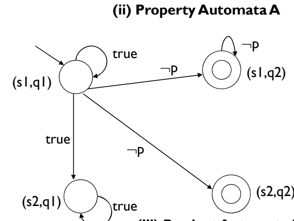


(i) System Model M



(ii) Property Automata A

M |≠ GF p



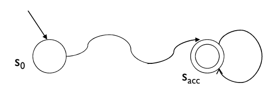
(iii) Product Automata M × A

Recall: LTL Model Checking

1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

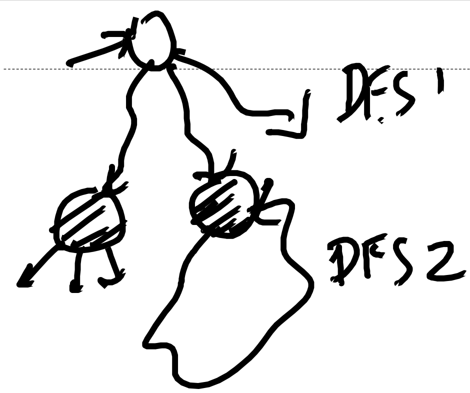
Emptiness Check

- Language($M \times A_{\neg\varphi}$) = empty-set?
  - Is there any trace which visits one of the accepting states of the product automata infinitely many times?
  - Look for accepting cycles.



Emptiness Check

- Perform DFS from initial state until you reach an accepting state s_{acc}
- When you reach s_{acc}, remember s_{acc} in a global var. and start a nested DFS from s_{acc}
  - Stop the nested DFS if you can reach s_{acc}
- If no accepting cycles are found, report yes.
- If accepting cycles are found
  - Concatenate the two DFS stacks and report it as counter-example trace of the LTL property.
- This algo. is implemented in SPIN model checker.



Nested DFS – step 1

- procedure dfs1(s)
  - push s to Stack1
  - add {s} to States1
  - if accepting(s) then
  - States2 := empty; seed := s; dfs2(s)
  - endif
  - for each transition s $\rightarrow$ s' do
  - if s' $\notin$ States1 then df1(s')
  - endfor
  - pop s from Stack1
- end

Nested DFS – step 2

- procedure dfs2(s)
  - push s to Stack2
  - add {s} to States2
  - for each transition s $\rightarrow$ s' do
  - if s' = seed then report acceptance cycle
  - else if s' $\notin$ States2 then df2(s')
  - endif
  - endfor
  - pop s from Stack2
- end

Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking
    - Model Checkers
      - SPIN

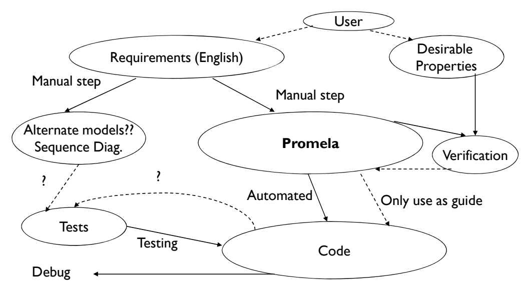
SPIN

- A tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A model checker
  - A random simulator for system simulation
  - Promela models can be automatically generated from a safe subset of C.

Our Usage

- Learn Promela, a low-level modeling language.
- Use it to model simple concurrent system protocols and interactions.
- Gain experience in verifying such concurrent software using the SPIN model checker.
- Gives a feel (at a small scale)
  - What are hard-to-find errors ?
  - How to find the bug in the code, once model checking has produced a counter-example ?

Our Usage



Features of Promela

- Concurrency
  - Multiple processes in a system description.
- Asynchronous Composition
  - At any point one of the processes active.
  - Interleaving semantics
- Communication
  - Shared variables
  - Message passing
    - Handshake (synchronous message passing)
    - Buffers (asynchronous message passing)

Features of Promela

- Within a process
  - Non-determinism : supports the situation where all details of a process may not be captured in Promela model.
  - Standard C-like syntax
    - Assignment
    - Switch statement
    - While loop
    - Guarded command
      - Guard and body may not evaluated together, that is, atomically.

Example

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
       tmp = tmp+1;
       state = tmp;
}

init  { run A() ; run A(); }
```

We need to define how processes are scheduled.

SPIN's process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the "active" processes will execute (non-deterministic choice here)
  - A process is active, if it has been created, and its "next" statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
    - We have not seen such an example yet !

Will this loop terminate?

Non-determinism within a single process.

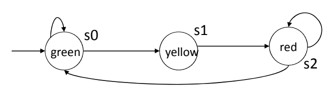
```
byte count;

proctype counter()
{
        do
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
        od;
}
```

Enumerate the reasons for non-termination in this example

This loop will not terminate

```
active proctype TrafficLightController() {
    byte color = green;
    do
     :: (color == green) -> color = yellow;
     :: (color == yellow) -> color = red;
     :: (color == red) -> color = green;
    od;
}
```



Channels

- SPIN processes can communicate by exchanging messages across channels
  - Apart from communication via shared variables.
- Channels are typed.
- Any channel is a FIFO buffer.
- Handshakes supported when buffer is null.
- chan ch = [2] of bit;
  - A buffer of length 2, each element is a bit.
- Array of channels also possible.
  - Talking to diff. processes via dedicated channels.

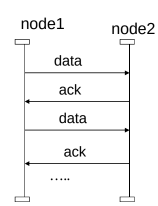
Example with channels

```
chan data, ack = [1] of bit;

proctype node1() {
do
:: data!1;
:: ack?1;
od
}

proctype node2() {
    do
    :: ack!1;
    :: data?1;
    od
}

init{ atomic{
        run node1(); run node2();
    }
}
```

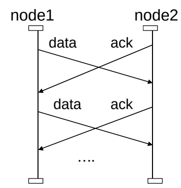


Example with channels

```
chan data, ack = [1] of bit;

        proctype node1() {            proctype node2() {
        do                                do
        :: data!1;                        :: ack!1;
        :: ack?1;                         :: data?1;
        od                                od
        }                                 }

        init{ atomic{
              run node1(); run node2();
           }
        }
```



SPIN Execution Semantics

- Select an enabled transition of any thread, and execute it.
- A transition corresponds to one statement in a thread.
  - Handshakes must be executed together.
    - *chan x = [0] of {…};*
    - *x!1* || *x?data*

SPIN Execution Engine

- while ((E = executable(s)) != {})
- for some (p,t) ∈ E
- { s' = apply(t.effect, s); /* execute the chosen statement */
- if (handshake == 0)
- { s = s' ;
- p.curstate = t.target
- }
- else{ …

SPIN Execution Engine

- /* try to complete the handshake */
  - E' = executable(s'); /* E' ={} ⇒ s unchanged */
  - for some (p', t') ∈ E'
  - { s = apply(t'.effect, s');
  - p.curstate = t.target;
  - p'.curstate = t'.target;
  - }
  - handshake = 0
  - } /* else */
  - } /* for some (p, t) ∈ E */
  - } /* while ((E = executable(s)) … */
  - while (stutter) { s = s }

Model Checking in SPIN

- (P1 || P2 || P3) |= φ
  - P1, P2, P3 are Promela processes
  - φ is a LTL formula
- Construct a state machine via
  - M, asynchronous composition of processes P1, P2, P3
  - $A_{\neg\varphi}$, representing $\neg\varphi$
- Show that "language" of $M \times A_{\neg\varphi}$ is empty
  - No accepting cycles.
- All these steps have been studied by us !!

Specifying properties in SPIN

- Invariants
  - Local: via assert statement insertion
  - Global: assert statement in a monitor process
- Deadlocks
- Arbitrary Temporal Properties (entered by user)
  - SPIN is a LTL model checker.
  - LTL properties can be entered as input to the checker!
    - Shown in the lab hour of the last lecture!

Connect system & property in SPIN

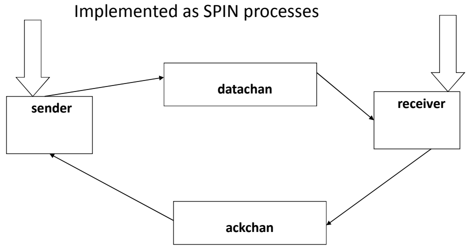
- **System model**
  - int x = 100;
  - active proctype A()
  - { do
  - :: x %2 -> x = 3*x+1
  - od
  - }
  - active proctype B()
  - { do
  - :: !(x%2) -> x = x/2
  - od
  - }
- **Property**
  - GF (x = 1)
- Insert into code
  - #define q (x == 1)
- Now try to verify GF q

More Involved Example

- Alternating Bit Protocol
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.

ABP Architecture

Implemented as SPIN processes



Sender & Receiver code

- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

```
active proctype Sender()
{   bit out, in;
    do
    :: datachan!out ->
            ackchan?in;
            if
            :: in == out -> out = 1- out;
            :: else fi
    od
}
```

```
active proctype Receiver()
{   bit in ;
    do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
    od
}
```


Timeouts

- Special feature of the language
  - Time independent feature.
    - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Model Checking in SPIN

- SPIN performs model checking by Nested DFS
  - Discussed in the past lecture !!
- Find acceptance states reachable from initial states (DFS).
- Find all such acceptance states which are reachable from itself (DFS).
- Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.

More readings on SPIN

- http://spinroot.com/spin/Man/Manual.html
  - SPIN manual
- The model checker SPIN (Holzmann)
  - IEEE transactions on software engineering, 23(5), 1997.
- http://spinroot.com/spin/Doc/SpinTutorial.pdf
  - SPIN beginner's tutorial (Theo Ruys)
- ``The SPIN model checker: primer and reference manual", by Holzmann (mostly chapters 2,3,7,8)
  - *This one is optional reading.*

Exercise – Model Checking – (1)

Two Computer Engineering students are taking the CS4271 exam. We must ensure that they cannot leave the exam hall at the same time. To prevent this, each student reads a shared token *n* before leaving the hall. The shared token is an arbitrary natural number. The global state of the system is given by *s1, s2, n where s1 and s2 are the local states of students 1 and 2 respectively.* Note that $s1 \in \{in, out\}$, $s2 \in \{in, out\}$. *The pseudo-code executed by the two students are:*

```
do forever{
if s1 = in and n is odd
        { s1 := out }
else if s1 = out
    { s1 := in; n := 3*n+1}
else {do nothing }
}
```

```
do forever{
   if s2 = in and n is even
                 {s2 := out}
   else if s2 = out and n is even
            { s2 := in ; n := n/2 }
   else { do nothing }
}
```

The two student processes are executed asynchronously. Every time one process is scheduled, it *atomically executes one iteration of its loop. The above system is an infinite state system. Design a* finite state abstraction and draw the global automata for the abstracted system. Your abstraction should be refined enough to prove mutual exclusion. Initially s1 = in and s2 = in.

Exercise – Model checking – (2)

- Consider the mutual exclusion property.
- Using the LTL model checking algorithm discussed in class, follow a step by step process to check the correctness of the property on the example given in the previous slide.