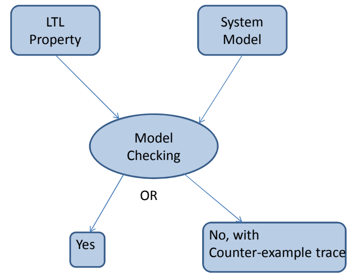
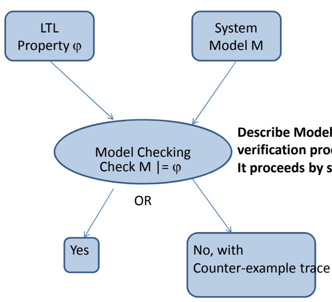
SPIN Model Checker & LTL Model Checking

Abhik Roychoudhury
CS 5219
National University of Singapore

Model Checking



Model Checking



Describe Model Checking as a general verification procedure.
It proceeds by search.

LTL Model Checking - steps

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?

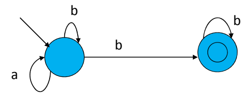
Recap: finite-state automata

- 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.

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 $\sigma \mid= p$
- 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

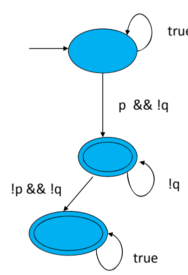


- Meaning as a regular language
  - $(a+b)^*b^+$
  - All finite length strings ending with b
- Meaning as a ω-regular language
  - All infinite length strings with finitely many a

LTL properties to automata

- Given a LTL property φ
  - We convert it to a ω-regular automata A_φ
- Language(A_φ) = $\{\sigma \mid \sigma \in \Sigma^\omega \wedge \sigma \mid= \varphi\}$
  - 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

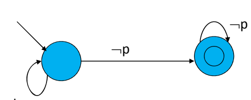
$G(p \Rightarrow (p \; U \; q))$

Recall: LTL Model Checking

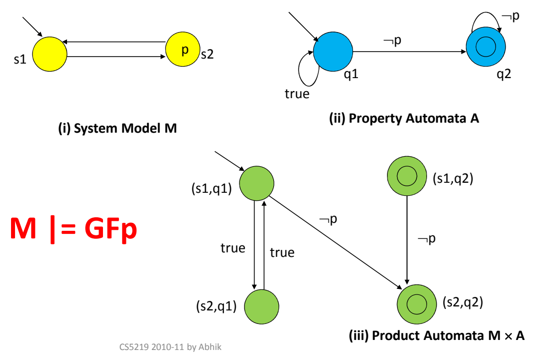
1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec 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.

Example: Verify GFp

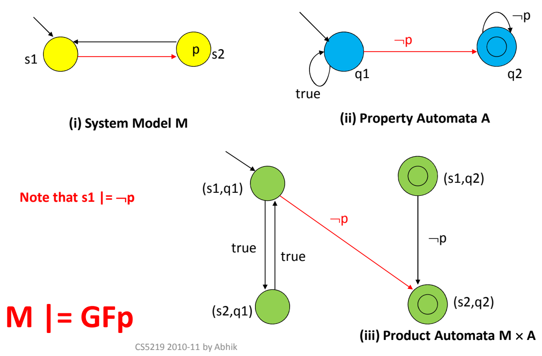
- Construct negation of the property
  - $\neg GFp \equiv FG\neg p$
- Construct automata accepting infinite length traces satisfying $FG\neg p$



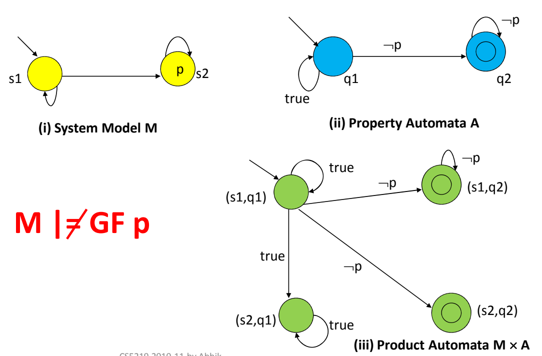
Product Automata



Product Automata Construction



Product Automata

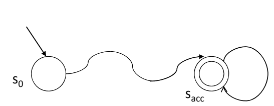


Recall: LTL Model Checking

1. Consider ¬φ. None of the exec. traces of M should satisfy ¬φ.
2. Construct a finite-state automata A $_{\neg\varphi}$ such that
   - Language(A $_{\neg\varphi}$) = Traces satisfying ¬φ
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.

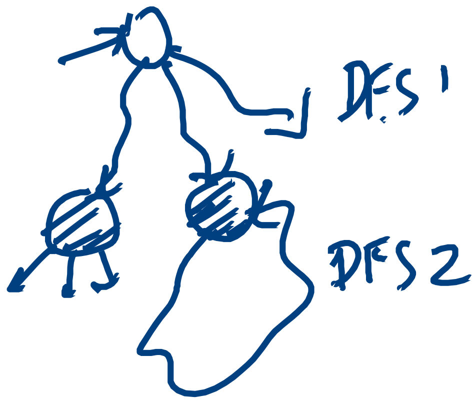
Emptiness Check

- Language(M × 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 → s' do
  - if s' ∉ 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 → s' do
  - if s' = seed then report acceptance cycle
  - else if s' ∉ States2 then df2(s')
  - endif
  - endfor
  - pop s from Stack2
- end

Organization

- So Far
  - Temporal logics
    - LTL, CTL, CTL*
  - General method for LTL Model Checking
- Now
  - Model checking in SPIN
  - SPIN's modeling language (briefly)
    - Promela

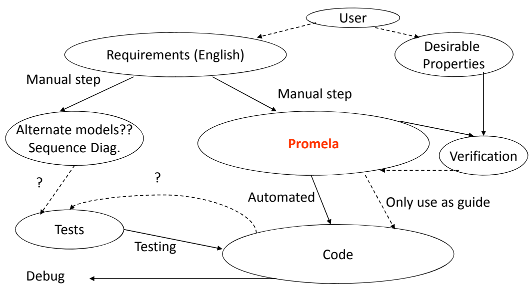
SPIN

- 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.

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 !

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 × $A_{\neg\varphi}$ 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.
- Why Verify, not Test?
  - "I have been fishing all day, I have found a number of fish since the morning, I cannot find any more now, I am pretty sure, there aren't any left!"
  - Bug finding techniques will ensure worse coverage than fishing in a small pond.

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

Model Checking in SPIN

- SPIN does not use SCC detection for detecting acceptance cycles (and hence model checking)
- The nested DFS algorithm used in SPIN is more space efficient in practice.
  - SCC detection maintains two integer numbers per node. (*dfs* and *lowlink* numbers)
  - Nested DFS maintains only one integer.
    - This optimization is important due to the huge size of the product graph being traversed on-the-fly by model checker.
- 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.

Organization

- So Far
  - Temporal logics
    - LTL, CTL, CTL*
  - General method for LTL Model Checking
- Now
  - Model checking in SPIN
  - SPIN's modeling language
    - Promela manual ☺
    - Go through this material a bit on your own ☺

Example 0

```
byte state = 0;

proctype A()
{  byte tmp;

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

init { run A() ; }
```

state : Global Variable

tmp : Local Variable

(state==0) -> tmp = state is a guarded command (blocked if the guard is false).

Only one process created.

Final value of **state** is 1

But SPIN allows multiple processes to be created.

Example 1

```
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.

Example 2

```
bit flag;
byte sem;
proctype myprocess(bit i)
{  (flag != 1) -> flag = 1;
     sem = sem + 1;
     sem = sem – 1;
     flag =  0;
}
proctype observer() {
       assert( sem != 2 );
}
```

```
init {
   atomic{
         run myprocess(0));
         run myprocess(1));
         run observer();
   }
}
```

All three processes Instantiated together

Issues

- Initial values of sem, flag not given
  - All possible init. values used for model checking.
- The system being verified is the asynchronous composition
  - myprocess(0) || myprocess(1)
- The property is the invariant
  - G sem ≠ 2
- Local & global invariants can be specified inside code via assert statements.

assert

- Of the form assert B
  - B is a boolean expression
  - If B then no-op else abort (with error).
- Can be used inside a process (local invariants)
  - proctype P(…) { x = … ; assert(x != 2); …. }
- Or as a separate observer process (global invariants)
  - proctype observer(){ assert(x != 2); }

Example 3

```
bit flags[2];
byte sem, turn;
proctype myprocess(bit id) {
    flags[id] = 1;
    turn = 1 – id;
    flags[1-id] == 0 || turn == id;
    sem++;
    sem--;
    flags[id] = 0;
}
```

```
init() {
   atomic{
       run myprocess(0);
       run myprocess(1);
       run observer(); }
}
proctype observer() {
       assert( sem != 2 );
}
```

Issues

- Can you use SPIN to prove mutual exclusion ?
  - What purpose does turn serve ?
- Arrays have been used in this example.
  - Flags is global, but each element is updated by only one process in the protocol
  - Not enforced by the language features.
- Processes could alternatively be started as:
  - active proctype myprocess(…) {
  - Alternative to dynamic creation via run statement

CS5219 2010-11 by Abhik

So far, in SPIN

- Process creation and interleaving.
- Process communication via shared variables.
- Standard data structures within a process.
- Assignment, Assert, Guards.
- NOW …
  - Guarded IF and DO statements
  - Channel Communication between processes
  - Model checking of LTL properties

CS5219 2010-11 by Abhik

Will this loop terminate?

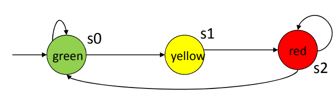
```
byte count;

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

Enumerate the reasons for non-termination in this example

CS5219 2010-11 by Abhik

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;
}
```



CS5219 2010-11 by Abhik

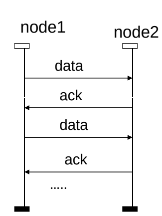
Channels

- SPIN processes can communicate by exchanging messages across channels
- 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.

CS5219 2010-11 by Abhik

Example with channels

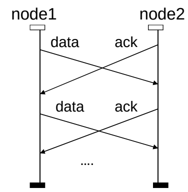
```
chan data, ack = [1] of bit;

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

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

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



CS5219 2010-11 by Abhik

Example with channels

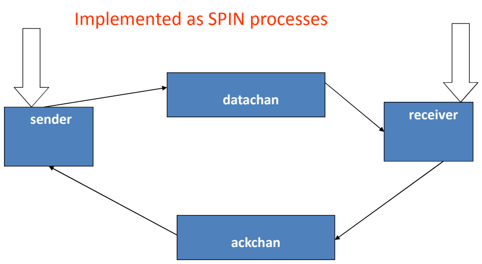
```
chan data, ack = [1] of bit;

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

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

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



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



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).

Readings (many sources)

- http://spinroot.com/spin/Man/Manual.html
  - SPIN manual (start with this !!)
- 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)
- Summer school Lecture notes on Software MC
  - (See Section 2 only),
  - http://spinroot.com/gerard/pdf/marktoberdorf.pdf