Lecture 10
Verification Models

LTL Quiz
## Recap

- **Testing:**
  - Cannot test exhaustively
  - Can show presence but not absence of bugs
  - Expensive
  - Not effective on multiple threads

- **Formal Models:**
  - Formal specifications
  - Theorem proving
  - Model checking
Models

- Abstraction of system
- Used for analysis
  - Simplified system: analysis is tractable
  - Performed in design phase: before implementation
- Not new:
  - We’ve been using models for years:
    - Prototypes

Process of Engineering

- Requirements
- Design
- Analysis
- Implementation
Process of Engineering

- Requirements: Specification: Logic
- Design: Model: Prototype
- Analysis: Computation: Model Checking
- Implementation

Relevance of Model Checking

- 1968: software crisis
  - Complexity of software grows faster than our ability to control it
  - Large programs then were 100K lines of code
Relevance of Model Checking

- **1968: software crisis**
  - Complexity of software grows faster than our ability to control it
  - Large programs then were 100K lines of code

- **Today:**
  - Large programs are 10,000 times larger
  - Memory is 100,000 times larger
  - Speeds are 1,000,000 times faster

Software development: has it changed since 1968?
Relevance of Model Checking

- 2012 software crisis: We cannot predictably produce reliable software
- This is the single most important unsolved problem in computer science

Standish Group Survey

- 1/3 of software projects are completed on time within budget
- More than 1/5 projects fail completely
Potential added value?

- Harmless and rare is not very important
- Catastrophic and rare is not adequately covered
- Model checking targets issues that have haunted developers for decades
- It is relevant, practical, and important

Feasibility of Model Checking

![Graph showing the memory needed and available over years from 1980 to 2000.](image)
Feasibility of Model Checking

Models

- Abstraction
  - Abstraction of reality
  - Less detail than the artifact being modeled
  - Details are selected on basis of relevance
  - Objective is to gain analytical power

- Purpose
  - Predict and explain
  - Too much or too little detail is not useful

- Design Aid
  - Often developed iteratively
  - Analysis often requires more than one model
How does it work?

- **System** \( L(S) \) the set of possible behaviors of \( S \)
- **Property** \( L(p) \) the set of valid behaviors
- **Prove that** \( L(S) \subseteq L(p) \) (Everything possible is valid)

Method: Prove \( L(S) \cap L(\neg p) = \emptyset \)

Possible Executions

Invalid Executions

Possible and invalid
Types of errors we’re looking for

- Deadlocks, livelocks, starvation
- Race conditions
- Locking problems, priority problems
- Resource allocation errors
- Reliance on relative speeds
- Violations of known system bounds
- Specification incompleteness
- Specification redundancy
- Logic problems

Livelock

- A condition that occurs when two or more processes continually change their state in response to changes in the other processes.
- The result is that none of the processes will complete.
- An analogy is when two people meet in a hallway and each tries to step around the other but they end up swaying from side to side getting in each other’s way as they try to get out of the way.
Deadlock

- A condition that occurs when two processes are each waiting for the other to complete before proceeding.
- The result is that both processes hang.
- Deadlocks occur most commonly in multitasking and client/server environments.
- A deadlock is also called a deadly embrace.

Race Condition

- An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.
Starvation

A multitasking-related problem, where a process is perpetually denied resources needed to complete its task.

Deadlock occurs when two programs each hold resources the other needs to finish, and neither is willing to give them up.

Starvation occurs when one program is unable to acquire the needed resource.

Starvation is illustrated by Dijkstra’s dining philosophers problem.

Properties:

- **Reachability** property: some particular situation can be reached.
- **Safety property**: Under certain conditions, an event never occurs.
- **Liveness property**: Under certain conditions, some event will occur
- **Deadlock freeness** property: a system can never be in a situation where no progress impossible
- A **fairness property** under certain conditions, an event will occur (or fail to occur) infinitely often.
Spin and SMV

- Two very popular model checkers
- SPIN
  - Software
  - Linear temporal logic
- SMV
  - Hardware
  - Computation Tree Logic
- Lots of others

Spin and Promela

- Spin is a model checker
- Models are built in Promela
- Promela is
  - A verification modeling language
  - Not a programming language
- Promela
  - Provides abstractions of essentials of processes
  - Does not provide data structures or programming details
Parts

- SPIN: Simple Promela Interpreter
- Promela: Process Meta Language
  - Behavior specification
- LTL: Linear Temporal Logic
  - Property specification

Promela

- Non deterministic, guarded command language for specifying possible system behaviors in distributed system design
  - Threads, interactions,
- Not intended to prevent specification of design errors (e.g., gotos are supported)
- Intended to support specification of real designs so they can be checked
Group Quiz 1 (10 minutes)

- What is the difference between a high level programming language like Java and a modeling language like Promela?
- Why does Promela allow only for a limited use of input statements like “scanf”?

Context

- Promela Behavior model
- Correctness Properties LTL
- Random and interactive Model simulation
- Guided simulation
- Error-trails Counter examples to Correctness properties
Concepts

- **Finite state models only**
  - Promela models are always bounded
  - Boundedness guarantees decidability
  - Finite models can allow infinite executions

- **Asynchronous behavior**
  - No hidden global system clock
  - No implied synchronization between processes

Concepts

- **Non-deterministic control structures**
  - Support (or inspire) abstraction of implementation level detail

- **Executability as a core part of semantics**
  - Every basic and compound statement is defined by precondition and effect
  - Statement can be executed producing the effect only when its precondition is satisfied, otherwise it is blocked
  - Example: q?m when channel q is non-empty, retrieve message m
Group Quiz 2 (5 minutes)

What are the three types of objects in Promela?

Promela Programs

- **Processes**
  - Global
  - As many as needed
  - Asynchronous

- **Channels**
  - For communication between processes

- **Variables**
  - Global or local
Executability

- A condition (bool exp) can only be passed if the condition is true.
- If it is not true, execution blocks until it is true.
- (a == b) has the meaning “while (a ! = b) skip;

Data Types

- bit 0 or 1
- bool false/0, true/10
- byte 0..255
- pid 0..255 (process id number)
- short -2^{15} .. 2^{15}-1
- int -2^{31} .. 2^{31}-1
- unsigned 0 .. 2^n-1 (1 <= n <= 32)
- mtype 1..255
- chan 1..255
- user defined

- Values are typical ranges: actual is implementation dependent, generally matching the C compiler
Declarations

- Objects must be declared before used
- Two types of scope:
  - Local
    - Local to a process (a thread)
    - Cannot be restricted to blocks inside a process
  - Global
    - Cannot be restricted to certain processes
- Examples
  
  ```
  bit x, y;
  byte a[12]; /* array of 12 elements, zero-base, all zero */
  int cnt = 67; /* initialize to 67 */
  short b[4] = 17; /* initialize all elements to 17 */
  ```

Symbolic names

- mtype = {apple, pear, orange, banana}; /* symbolic names */
- Names in the list are assigned integer values from 1 to 255
- Multiple mtype statements can appear, but only one list is formed
- Print the name using printm routine
- No reserved words can appear
Basic statements

- Basic statements define the primitive state transformers
- They end up labeling the edges in the underlying finite state automata
- Statements are either
  - Executable
  - Blocked

Statements

- Declarations
- Assignment
- Control (If, Do, GOTO, …)
- expression
- Atomic
- Print
- Assert
- Send/receive
Assignments

- Assignments: unconditional, changes exactly one variable
  - `b = 10;`
  - `c++; /* same as c = c + 1; */`
- `<var> = <expression>`
- `(expr1 -> expr2 : expr3)`
  - In C: if (expr1) then expr2 else expr3;
  - In C: expr1 ? expr2 : expr3;

Operators

- `, ~, ++, --`, `%/`, `*/+-`, `<< >>`, `<, <=, ==, !=, >=, >`, `&& ||`, `->`, `=`
  - negation, compliment, increment, decrement
  - modulo
  - arithmetic
  - left and right shift
  - relational operators
  - logical and or
  - conditional expression operator
  - assignment operator
Expressions

- Expression: executable only if transition evaluates to nonzero
  
  \[(a == b); \quad //\text{same as in C: while (a!=b) skip;}\]

  - Used to control execution of other statements
  - Expressions must be side effect free
  - Can’t have \(b = c++;\)

Compound Statements

- Atomic
- Deterministic steps
- Selections
- Iterations
- Escapes
Atomic/Deterministic

- atomic \{ temp = b; b = c; c = temp \}
  - Uninterruptible

- d_step \{ temp = b; b = c; c = temp \}
  - Like atomic except that no non-deterministic code can reside inside dstep and no statement can block inside dstep

Control

```latex
If
::(cond) -> option
::(cond) -> option
:: else
fi
```

- First statement is guard (cond)
- Option is executed only if guard is true
- Only one statement is executed
- If more than one guard is enabled, one is selected non-deterministically
- Else part is only executable if all of the conditions are blocked
Control

\[
do
:: \text{count} = \text{count} + 1; \\
:: \text{count} = \text{count} - 1; \\
:: (\text{count} == 0) \text{ break}; \\
:: \text{else} -> \text{break}
od
\]

- the assignments are always enabled.
- either of the first two may get executed,
- when count is 0, the third statement may get executed (ending the loop) (It might not get executed, too).
- If none of the lines is enabled, the statement blocks
- Break exits the loop

loops

\[
do
:: (\text{count} != 0) -> //Condition to enter the loop \\
    \text{if}
    :: \text{count}++; //one of these two statements will \\
    :: \text{count}--; //non-deterministically get executed \\
    \text{fi}
:: \text{else} -> \text{break} // get out of the loop
od
\]
Group Quiz 3: (10 minutes)

do
:: (count != 0) -> //Condition to enter the loop
    if
:: count++; //one of these two statements will
:: count--;  //non-deterministically get executed
    fi
:: else -> break // get out of the loop
od

What does this piece of code do?
Demo: spin_Tutorial_Loop.pml

Atomic, Print, Assert

Print unconditional, no effect on state
assert (<cond>) simulation stops if cond is false
timeout gets executed only if no other statement is enabled
skip always enabled, does nothing
ture always executable
(1) Just like true
run
Processes

- Processes are defined using
  - proctype declaration
    - active proctype name '(' [decl list] ')' 
      
    ```
    {<sequence>}
    ```

- The active prefix creates an instance of the process on startup
- Multiple instance can be created
  - active [2] proctype hello()
    - { printf("pid %d\n", _pid) }

Group Quiz 4: (10 minutes)

- active [2] proctype you_run()
  
  ```
  { printf("you_run: %d\n", _pid) }
  ```

- active [2] proctype hello()
  
  ```
  { printf("hello %d\n", _pid) }
  ```

What are the possible outputs of a random simulation run of this program?
Demo: spin_Tutorial_Process.pml
More processes

- init is a special process
  
  ```
  init { run hello(); run hello() }
  $spin hello.pml
    pid 1
    pid 2
  ```

- Note that init is a process (3 created here)
  - Init is always process 0

- Note that SPIN indents the output

- Process may but do not need to start immediately after creation

- A process terminates once it completes its tasks
  - At which point the same process might be created again to handle the next work assignment

Group Quiz 5: (10 minutes)

```proctype hello()
{
    printf("hello %d\n", _pid)
}
init
{
    run hello();
    run hello()
}
```

Which of the following are possible outputs from running a random simulation of the code above? Justify your answer

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>--</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>hello 1</td>
<td>hello 1</td>
<td>hello 2</td>
<td>hello 2</td>
<td>hello 0</td>
<td>hello 0</td>
<td>hello 1</td>
<td>hello 0</td>
<td>hello 0</td>
</tr>
<tr>
<td>hello 1</td>
<td>hello 2</td>
<td>hello 1</td>
<td>hello 2</td>
<td>hello 1</td>
<td>hello 0</td>
<td>hello 0</td>
<td>hello 1</td>
<td>hello 2</td>
</tr>
</tbody>
</table>
Process State

- Each process has its own local state
  - Program counter (control flow point)
  - Values of local variables
- The model as a whole has global state
  - Value of all globally declared variables
  - Contents of all message channels
  - Set of all currently active processes

Dynamic process creation

- State of system maintained in global state vector
- State of vector contains entries for
  - Value of global vars and message channels
  - All active processes
    - Local vars for each
Process stack

- Processes are added and deleted in stack order
- Processes can start and stop at any time
- Processes can only be removed from the state vector in LIFO order
- pid is recycled only when the process dies
- Deletion: 2 steps
  - Termination
  - Death: only after all children have died
  - Pid of init cannot be recycled

How is finiteness preserved?

- Promela models are finite state
  - Only finitely many active processes
  - Only finitely many statements in a proctype
  - No data types with unbounded range
    - bool: 0, 1
    - pid: 0-255
    - ...
    - Channels are bounded
Parameters

```c
proctype junk (int x)
{
    printf("%d \n", x)
}
init { run junk (4); run junk (9) }
```

- run is an expression, not a statement
- run is the only expression with a side effect
- run returns the pid of the new process
- init process does not accept parameters
- active processes have parameters initialized to zero

Statement interleaving

- Processes execute concurrently and asynchronously
- Process scheduling is non-deterministic
- Statement executions from different processes are interleaved arbitrarily
- Local choices can also be non-deterministic
- Processes only synchronize behavior
  - Through use of global variables
  - Vie message passing
  - No global clock to synchronize
Message Channels

- Used to model data exchange between processes
- Can be declared globally or locally
- Declared as
  - chan fred = [16] of {short, short, byte};
- This declares a channel named “fred”
- The channel holds up to 16 messages
- Each message has three fields: two shorts and a byte
- Each field must be a user-defined or predefined type
  - can’t use an array
  - can embed the array in a user-defined type

Messages: sending

- Sending
  - fred(expr1, expr2, expr3)
  - fred(expr1(expr2, expr3))
  - This sends a message to the channel “fred” that should be expecting three fields
  - By default blocks if channel is full.
  - It’s an error to send/receive less/more parameters
  - Can be set to lose the message if full (spin option –m)
Messages: Receiving

- Receiving
  - fred?var1, var2, var3
  - fred?(var1)var2, var3
  - This reads a message from the channel “fred” and assigns field values to var1, var2, and var3
  - Blocks if channel is empty
  - fred?cons1, var2, var3 (for constant cons1) blocks until message with first argument matching cons1 is in queue

Group Quiz 6 (10 Minutes)

What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What output would the first 10 simulation steps for this model produce?

**Channel** | **Processor** | **Message**
---|---|---
Proc 0 = Sender | | 
Proc 1 = Receiver | | 
1 | 0 | to_rcvr!msg1
What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proc 0 = Sender</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_rcvr?msg1</td>
</tr>
<tr>
<td></td>
<td>Proc 1 = Receiver</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_rcvr?msg1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
</tbody>
</table>
What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proc 0 = Sender</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td></td>
<td>Proc 1 = Receiver</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>to_sndr?msg1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_sndr?msg1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
</tbody>
</table>

What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proc 0 = Sender</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td></td>
<td>Proc 1 = Receiver</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>to_sndr?msg1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_sndr?msg1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
</tbody>
</table>

mtype = {msg0, msg1, ack0, ack1};

active proctype Sender()
{
    again: to_rcvr!msg1;
    to_sndr!ack1;
    to_rcvr!msg0;
    to_sndr?ack0;
    goto again
}

active proctype Receiver()
{
    again: to_rcvr?msg1;
    to_sndr!ack1;
    to_rcvr?msg0;
    to_sndr!ack0;
    goto again
}
mtype = {msg0, msg1, ack0, ack1};
chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };

active proctype Sender()
{
    again: to_rcvr!msg1;
    to_sndr?ack1;
    to_rcvr!msg0;
    to_sndr?ack0;
    goto again
}

active proctype Receiver()
{
    again: to_rcvr?msg1;
    to_sndr!ack1;
    to_rcvr?msg0;
    to_sndr!ack0;
    goto again
}

What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proc 0 = Sender</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1 0</td>
<td>Proc 1 = Receiver</td>
<td>to_rcvr?msg1</td>
</tr>
<tr>
<td>2 1</td>
<td></td>
<td>to_sndr!ack1</td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td>to_rcvr!msg0</td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td>to_rcvr?msg0</td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td>to_sndr!ack0</td>
</tr>
<tr>
<td>2 1</td>
<td></td>
<td>to_sndr!ack1</td>
</tr>
</tbody>
</table>
Group Quiz 6 (10 Minutes)

```plaintext
mtype = {msg0, msg1, ack0, ack1};
chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };

active proctype Sender()
{
    again:
        to_rcvr!msg1;
        to_sndr!ack1;
        to_rcvr!msg0;
        to_sndr?ack0;
        goto again
}

active proctype Receiver()
{
    again:
        to_rcvr?msg1;
        to_sndr!ack1;
        to_rcvr?msg0;
        to_sndr!ack0;
        goto again
}
```

What output would the first 10 simulation steps for this model produce?

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc 0</td>
<td>= Sender</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>to_rcvr!msg1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_rcvr?msg1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>to_sndr!ack1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>to_rcvr!msg0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>to_rcvr?msg0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>to_sndr!ack0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>to_sndr?ack0</td>
</tr>
</tbody>
</table>

Elevator example

- Small elevator services three floors
- At most one door should be open at a time
- The door at the floor where the elevator is the only one that can be open
- The elevator needs to service every floor
Promela Code

```
bit doorisopen[3];
chan openclosedoor=[0] of {byte,bit};

proctype door(byte i)
{ do :: openclosedoor?eval(i),1;
   doorisopen[i-1]=1;
   doorisopen[i-1]=0;
   openclosedoor!i,0
   od
  }
}
```

Promela: elevator

```
proctype elevator ()
{ show byte floor = 1;
  do :: floor !=3 -> floor++;
     :: floor !=1 -> floor --;
     :: openclosedoor!floor,1;
     openclosedoor?eval(fLOOR),0
     od
  }
init { atomic { run door(1); run door(2); run door(3); run elevator(); } }
```
demo

- Code Elevator.pml
- Run spin simulation spin Elevator.pml
- Assert
  - Run verification spin –a elevator.pml
  - gcc –DBFS –o pan pan.c
  - spin –v –t elevator.pml

Assert

assert (doorisopen[0] &&
    doorisopen [1] &&
    doorisopen [2]);