Outline

- Verification and Validation
- History and motivation
- Spin
- Promela language
- Promela model

Verification vs. Validation

- Software verification is often confused with software validation
- Software verification is a verification of conformance to the specification
- Software validation is a validation of the compliance with the requirements
Common Design Flaws

- Deadlock
-Livellock
- Underspecification
- Overspecification
- Violations of constraints
- Assumptions about speed

Diagnosing Design Flaws

- Complexity makes design flaws difficult to uncover.
- Engineers often use simplified models (prototypes) for design verification.
- Abstract models can also be used to verify concurrent systems.

What is Model Checking?

- Use a simplified model of our system.
- Verify the system exhaustively.
- Automatically check that given properties hold in all possible states.
System State Based Analysis

- Complete state space must be represented
- State space defined by significant variables
- Each integer variable has $2^{32}$ distinct possibilities. Two such variables have $2^{64}$ possibilities.
- In concurrent protocols, the number of states usually grows exponentially with the number of processes.

State Space Capture

- System is the asynchronous composition of processes
- For each state the successor states are enumerated using the transition relation of each process
Reducing Complexity

- Problem: state space explosion!
- Automatic state space compression and reduction by SPIN.
- Manual reduction techniques by the designer.
  - We need to find the smallest sufficient model of our system.
  - Biggest challenge!

If it is so constrained, is it of any use?

- Many protocols are finite state.
- Many programs or procedure are finite state in nature. Can use abstraction techniques.
- Sometimes possible to decompose a program, and prove part of it by model checking and part by theorem proving.
- Many techniques to reduce the state space explosion (Partial Order Reduction).

Alternating Bit Protocol

```
#type = {MSG, ACK};
chan toS = [2] of {#type, bit};
chan toR = [2] of {#type, bit};
proctype sender(chan in, chan out)
{
  bit sendbit, recvbit;
  do
  :: out!MSG, sendbit ->
     in?ACK, recvbit
    if
    :: recvbit == sendbit ->
      sendbit = 1-sendbit
    :: else
      fi
  od
}
proctype receiver(chan in, chan out)
{
  bit recvbit;
  do
  :: in!MSG, recvbit ->
     out!ACK, recvbit
  od
}
init
{
  run sender(toS, toR);
  run receiver(toR, toS);
}
```
What is this all about?

- **SPIN**
  - On-the-fly verifier developed at Bell-labs by Gerard Holzmann and others

- **Promela**
  - Modeling language for SPIN
  - Targeted at asynchronous systems

“First Computer Bug”

History

- Work leading to SPIN started in 1980
  - First bug found on Nov 21, 1980 by Pan
  - One-pass verifier for safety properties

- Succeeded by
  - Pandora (82)
  - Trace (83)
  - SuperTrace (84)
  - SdValid (88)
  - SPIN (89)
SPIN

- SPIN (Simple PROMELA Interpreter)
  - Tool for analyzing the logical consistency of concurrent systems.
  - Takes a PROMELA model as input.
- Model-checker.
- Based on automata theory.
- Allows LTL or automata specification
- Efficient (on-the-fly model checking, partial order reduction).
- Developed in Bell Laboratories.

SPIN Features

- "press on the button" verification (model checker)
- Efficient implementation
- Graphical user interface (Xspin)
- Used for research and industry
- Contains more than two decades research on advanced computer aided verification (many optimization algorithms)

The language of SPIN

- Called Promela
- The expressions are from C.
- The communication is from CSP.
- The constructs are from Guarded Command.
Command Line Tools

- Spin
  - Generates the Promela code for the LTL formula
  - Generates the C source code
- Pan (Process Analyzer)
  - Performs the verification
    - Has many compile time options to enable different features
    - Optimized for performance

Simulator

- Spin can also be used as a simulator
  - Simulated the Promela program
- It is used as a simulator when a counterexample is generated
  - Steps through the trace
  - The trace itself is not “readable”
- Can be used for random and manually guided simulation as well

Xspin
XSpin Features

- Graphical front-end to the SPIN model checker.
- Features:
  - Editor
  - Syntax checking
  - Simulation
  - Verification
  - Requirements specification

XSpin Screenshot

Types of Properties

- Invalid end-states (deadlock)
- Assertion violations
- Unreachable code
- Liveness properties
  - Non-progress cycles
  - Acceptance cycles
- Linear Temporal Logic (LTL) formulae
Spin capabilities

• Interactive simulation
  – For a particular path
  – For a random path
• Exhaustive verification
  – Generate C code for verifier
  – Compile the verifier and execute
  – Returns counter-example
• Lots of options for fine-tuning

Spin overall structure

PROMELA

• PROMELA (Process/Protocol Meta Language)
  – Specification language to model finite-state systems.
  – Dynamic creation of concurrent processes.
  – Communication via synchronous or asynchronous message channels.
  – Non-deterministic (you’ll see!).

• Language for asynchronous programs
  – Dynamic process creation
  – Processes execute asynchronously
  – Communicate via shared variables and message channels
    • Races must be explicitly avoided
    • Channels can be queued or rendezvous
  – Very C like
Finite Systems Only!

- No unbounded data.
- No unbounded message channels.
- No unbounded processes.
- No unbounded process creation.

Variables and Types

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime
- Default initial value of basic variables (local and global) is 0.

Variables

- Variables should be declared
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing
- Variables can be used in expressions
- Most arithmetic, relational, and logical operators of C/Java are supported
Data Types

- Basic: bit/bool, byte, short, int, chan
- Arrays: fixed size
  - byte state[20];
  - state[0] = state[3 * i] + 5 * state[7/j];
- Symbolic constants
  - Usually used for message types
  - mtype = {SEND, RECV};

Basic Types (integer)

<table>
<thead>
<tr>
<th>Declarations</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit turn=1;</td>
<td>[0..1]</td>
</tr>
<tr>
<td>bool flag;</td>
<td>[0..1]</td>
</tr>
<tr>
<td>byte counter;</td>
<td>[0..255]</td>
</tr>
<tr>
<td>short s1, s2;</td>
<td>[-2^16..1; 2^16-1]</td>
</tr>
<tr>
<td>int msg;</td>
<td>[-2^32..1; 2^32-1]</td>
</tr>
</tbody>
</table>

Arrays

- `<type> <array name>[<array size]>;
- byte a[27]; // array a can hold 27 bytes
- bit flags[4]; // array flags can hold 4 bits
- Same as C/C++
- Array index starts at 0
- Array index ends at size-1
Expressions

- Arithmetic: +, -, *, /, %
- Comparison: >, >=, <, <=, ==, !=
- Boolean: &&, ||, !
- Assignment: =
- Increment/decrement: ++, --

Records

- Type definition defines records (structure)
  typedef MyRecord {
    short f1; byte f2;
  }  
- Variable declaration defines variable
  MyRecord rr;
- Values are reference field by field
  rr.f1 = …
  rr.f2 = …
  byte a = rr.f2;

Message types and channels

- Message type is enumeration declaration
  <type name>= {'<value list>'}
- mtype = {OK, READY, ACK, ERROR}
  − #define OK = 1;
  − #define READY=2;
  − #define ACK=3
  − #define ERROR=4
- mtype Mvar = ACK
Channels

- Channel defines queue that is used to pass messages between processes
  
  `chan <name>=[<size>] of {<message record>}`

- `chan Ng=[2] of {mtype, byte, byte},` 
  `Next=[0] of [byte]`

- Enqueue a message – blocks if channel has no space
  `<name>!<record fields>`

- Dequeue a message – blocks if channel is empty
  `<name>?<variables to receive fields>`

- `Ng!OK(5,4); /* send message type OK with data 5 and 4` 
  `Ng!OK,5,4;   /* same as OK(5,4)`

- `Ng?OK(byte0,byte1); /* store data if type is OK`
Executable Statements

• An assignment is always executable
• An expression is also a statement; it is executable if it evaluates to non-zero
  – $2 < 3$ always executable
  – $x < 27$ only executable if value of $x$ is smaller 27
  – $3 + x$ executable if $x$ is not equal to $-3$

Executability

• The body of a process consists of a series of statements.
• Statements are either executable or blocked.
• No difference between conditions and statements
  – Execution of every statement is conditional on its executability
  – Executability is the basic means of synchronization
• Declarations and assignments always executable
• Conditionals are executable when they hold
• The following are the same
  while ($a != b$) skip
  ($a == b$)

Statements

• The skip statement is always executable
• A run statement is only executable if a new process can be created (the number of processes is bounded)
• A printf statement is always executable
• Statements in a sequence are separated by a semi-colon: “;”
• A given statement in a sequence isn’t executable until previous statement executed
Sample

```c
int x;
proctype A() {
    int y=1;
skip;
run N();
x=2;
x>2 && y==0;
skip;
}
```

**Executable if N can be created...**

**Can only become executable if some other process makes x greater than 2**

assert(<expression>);

- The assert statement is always executable
- If <expr> evaluates to zero, SPIN will exit with an error, as "the <expr> has been violated"
- The assert statement is often used to check whether certain properties are valid in a state

```c
proctype monitor() { assert(n <= 3); }
proctype receiver() { ...
toReceiver ? msg;
assert(msg !=ERROR);
... }
```

if-statement

```c
if :: choice1 -> stat1.1; stat1.2; stat1.3; ... :: choice2 -> stat2.1; stat2.2; stat2.3; ... :: ...
:: choice_n -> stat_n.1; stat_n.2; stat_n.3; ...
fi;
```
if-statement

- If there is at least one choice, (guard) executable, the if statement is executable and SPIN non-deterministically chooses
- If no choice is executable, the if-statement is blocked
- The operator “->” is equivalent to “;”
- The else guard is always executable
- Guard need not be exhaustive or mutually exclusive

---

do-loops

do
:: choice1 -> stat1.1; stat1.2; stat1.3; ... 
:: choice2 -> stat2.1; stat2.2; stat2.3; ... 
:: ...
:: choice_n -> stat_n.1; stat_n.2; stat_n.3; ... 
od;

---

do-loops

- With respect to the choices, a do statement behaves in the same way as an if statement
- However, instead of ending the statement at the end of the chosen list of statements, a do-statement repeats the choice selection
- The (always executable) break statement exits a do-loop statement and transfers control to the end of the loop
### goto-statement

- Transfer control to a non-sequential statement
  
  ```promela```
  ```
goto <label >;
  ```
- Transfers execution to label
- Each Promela statement might be labeled
- Quite useful in modeling communication protocols

### Interleaving Semantics

- Promela processes execute concurrently
- Non-deterministic scheduling of the processes
- Processes are interleaved
- All statements are atomic; each statement is executed without interleaving with other processes
- Each process may have several different possible actions enabled at each point of execution

### atomic-statement

- Groups statements into an atomic sequence
  
  ```promela```
  ```
atomic { st_1; st_2; ... st_n }
  ```
- all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if st_1 is executable
- if a st_i is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step
PROMELA Model Basics

- Promela model consists of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations

```
mtype = {MSG, ACK};
chan toS;
chan toR;
bool flag;
proctype sender()
{
...
}
proctype receiver()
{
...

init{
...}
creates
processes
```

Promela Code

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;
proctype Sender() {
...
}
proctype Receiver() {
...

init {
...}
creates
processes
```

Promela Model Syntax

- Type declarations
  - mtype, typedefs, constants
- Channel declarations
  - chan ch= [dim] of (type, ...)
  - asynchronous: dim=0
  - rendez-vous: dim>0
- Global variable declarations
  - can be accessed by all processes
- Process declarations
  - behaviour of the processes
  - local variables + statements
- [initprocess]
  - initializes variables and starts processes
Processes

- A process is defined by a proctype definition
- A process executes concurrently with all other processes, independent of speed of behavior
- A process communicates with other processes
  - using global (shared) variables
  - using channels
- There may be several processes of the same type
- Each process has its own local state

Process Example

```plaintext
byte state = 2;

proctype A() { (state == 1) -> state = 3 }

proctype B() { state = state - 1 }
```
Process Instantiation

byte state = 2;
proctype A() { (state == 1) -> state = 3 }
proctype B() { state = state – 1 }
init { run A(); run B(); }

• run can be used anywhere

Parameter passing

proctype A(byte x; short foo) {
  (state == 1) -> state = foo
}
init { run A(1,3); }

• Data arrays or processes cannot be passed

Variable scoping

• Global scope variables are known throughout the model
• Process local scope variables are only known within the process
• Parameters are only known within the process and initialized to passed value

• byte foo, bar, baz;
  proctype A(byte foo) {
    byte bar;
    baz = foo + bar;
  }
Races and deadlock

byte state = 1;
proctype A() {
    (state == 1) -> state = state + 1
}
proctype B() {
    (state == 1) -> state = state - 1
}
init { run A(); run B() }

Atomic Sequence

byte state = 1;
proctype A() { atomic {
    (state == 1) -> state = state + 1
} }
proctype B() { atomic {
    (state == 1) -> state = state - 1
} }
init() { run A(); run B() }

Message Passing

- Convention: first message field often specifies message type (constant)
  - Alternatively send message type followed by list of message fields in braces
    - qname/exp1(exp2, expr3)
    - qname?var1(var2, var3)
- Channel declaration
  - chan qname = [16] of {short}
  - chan qname = [5] of {byte, int, chan, short}
- Sending messages
  - qname/exp
  - qname/exp1, expr2, expr3
- Receiving messages
  - qname?var
  - qname?var1, var2, var3
Message Passing Mismatch

- More parameters sent
  - Extra parameters dropped
- More parameters received
  - Extra parameters undefined
- Fewer parameters sent
  - Extra parameters undefined
- Fewer parameters received
  - Extra parameters dropped

Message Passing Example

```pemela```
chan x = [1] of {bool, bool};
chan y = [1] of {bool};

proctype A(bool p, bool q) { x!p,q ; y?p }
proctype B(bool p, bool q) { x?p,q ; y?q }

init { run A(1,2); run B(3,4) }
```

Executability

- Send is executable only when the channel is not full
- Receive is executable only when the channel is not empty
- A channel size reflects the ability of a channel to "store" a message for a future consumer
- len(qname) returns the number of messages currently stored in qname
- If used as a statement it will be unexecutable if the channel is empty
Rendezvous

- Channel of size 0 defines a rendezvous port
- Can be used by two processes for a synchronous handshake
- No queueing
- The first process blocks
- Handshake occurs after the second process arrives

Procedures and Recursion

- Procedures can be modeled as processes
- Even recursive ones
- Return values can be passed back to the calling process via a global variable or a message

Timeouts

Proctype watchdog() {  
do  
:: timeout -> guard!reset  
od
}  
- timeout is a predefined global boolean that is set true when the entire system is deadlocked
- No absolute timing considerations
Processes

- Process are created using the run statement (which returns the process id)
- Processes can be created at any point in the execution (within any process)
- Processes start executing after the run statement.
- Processes can also be created by adding active in front of the proctype declaration
  - Parameters will be initialized to 0

Processes

- There may be several processes of the same proctype
- Each process has its own local state:
  - process counter (location within the proctype)
  - contents of the local variables

Hello World!

```c
active proctype Hello( ) {
    printf("Hello process, my pid is: %d\n", _pid);
}

init{
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid= run Hello();
    printf("last pid was: %d\n", lastpid);
}
```
Concurrent Processes

```c
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
    beginning:
    noncritical:
        state[id] = NONCRITICAL;
        if :: goto noncritical;
        fi;
    trying:
        state[id] = TRYING;
        if :: goto trying;
        fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        fi;
    goto beginning;
} init { run process(0); run process(1); }
```

Message Type Definition

```c
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
    beginning:
    noncritical:
        state[id] = NONCRITICAL;
        if :: goto noncritical;
        fi;
    trying:
        state[id] = TRYING;
        if :: goto trying;
        fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        fi;
    goto beginning;
} init { run process(0); run process(1); }
```

XSpin Directive (show)

```c
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
    beginning:
    noncritical:
        state[id] = NONCRITICAL;
        if :: goto noncritical;
        fi;
    trying:
        state[id] = TRYING;
        if :: goto trying;
        fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        fi;
    goto beginning;
} init { run process(0); run process(1); }
```
Process Definition

```promela
mtype = { NONCRITICAL, TRYING, CRITICAL};
shyx mtype state[2];
proctype process(int id) {
beginning:
    noncritical:
        state[id] = NONCRITICAL;
        if :: goto noncritical;
        :: true;
    fi;
    trying:
        state[id] = TRYING;
        if :: goto trying;
        :: true;
    fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        :: true;
    fi;
    goto beginning;
init { run process(0); run process(1); }
}
```

Start Execution

```promela
mtype = { NONCRITICAL, TRYING, CRITICAL};
shyx mtype state[2];
proctype process(int id) {
beginning:
    noncritical:
        state[id] = NONCRITICAL;
        if :: goto noncritical;
        :: true;
    fi;
    trying:
        state[id] = TRYING;
        if :: goto trying;
        :: true;
    fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        :: true;
    fi;
    goto beginning;
init { run process(0); run process(1); }
```

Execution

```promela
mtype = { NONCRITICAL, TRYING, CRITICAL};
shyx mtype state[2];
proctype process(int id) {
beginning:
    noncritical:
        if :: goto noncritical;
        :: true;
    ::
    trying:
        state[id] = TRYING;
        if :: goto trying;
        :: true;
    fi;
    critical:
        state[id] = CRITICAL;
        if :: goto critical;
        :: true;
    fi;
    goto beginning;
init { run process(0); run process(1); }
```
**Enabled Statements**

- A statement needs to be enabled for the process to be scheduled.

```pseudocode
bool a, b;
proctype p1()
{
 a = true;
 a & b;
 a = false;
}
proctype p2()
{
 b = false;
 a & b;
 b = true;
}
init { a = false; b = false; run p1(); run p2(); }
```

These statements are enabled only if both `a` and `b` are true.

This statement can never be enabled since `b` is always false.

**Alternating Bit Protocol**

Sender

Receiver

- `msg0`
- `ack0`
- `msg1`
- `ack1`
- `msg0`
- `ack0`
- `msg1`

**Message Error**

Sender

Receiver

- `msg0`
- `ack1`
- `msg0`
- `ack0`
Retransmission

Sender

msg0

msg0

ack0

msg1

Receiver

msg0

ack1

Message and Channel Definitions

\[ mtype = \{ \text{msg0, msg1, ack0, ack1} \} \]

\[ \text{chan sender} = \{1\} \text{ of } \{ \text{mtype} \}; \]

\[ \text{chan receiver} = \{1\} \text{ of } \{ \text{mtype} \}; \]

Sender Process

\[ \text{active proctype Sender()} { \}
\]

\[ \text{do} \]

\[ \text{if } \]

\[ \text{if receiver?msg0; skip fi; do} \]

\[ \text{sender?ack0 -> break} \]

\[ \text{sender?ack1 \text{timeout -> } if} \]

\[ \text{receiver?msg0; skip fi; do; break} \]

\[ \text{sender?ack1 -> break} \]

\[ \text{sender?ack0 \text{timeout -> } if} \]

\[ \text{receiver?msg1; skip fi; do; break} \]
Receiver Process

```plaintext
receiver proctype Receiver()
{ do :: do :: receiver?msg0 -> sender!ack0; break;
:: receiver?msg1 -> sender!ack1
od
:: receiver?msg1 -> sender!ack1; break;
:: receiver?msg0 -> sender!ack0
od
}
```

Lynch's Protocol

- … a reasonable looking but inadequate scheme …
- Full duplex operation on two channels
- If previous reception was error-free, the next message on reverse channel contains ACK, otherwise NAK
- If previous reception carried NAK or was in error, retransmit, otherwise send new message

Lynch's Protocol Problems

- Cannot send ACK/NAK without data – need to send fill
- Startup not defined – send error to start process
- Receiver cannot tell whether transmission is retransmission of properly received data or new data