Methods of Assessing Model Behavior

- Testing
  - "spot checks" aspects of real system
- Simulation
  - "spot checks" aspects of abstract (model) system
- Deductive verification
  - Uses axioms and proofs on a mathematical model of system
- Model checking
  - Exhaustively checks states of a finite state model

Testing

- Requires the real system
  - Remember the "cost to repair" during testing?
- Can't test all possibilities
- Primarily an experimental approach
- For embedded systems, the same test may yield varying results depending on timing.

Simulation

- Tests a model of the real system
  - Cheaper than testing
- Many details can be abstracted away
  - Lets us concentrate of the important aspects
  - Can simulate long before we can test with code
- Works fairly well, cost is medium
- For embedded systems, often the only way for "real" execution prior to having the hardware
Deductive Verification

- Extremely powerful
- Extremely hard
- One proof can cover very large range of behaviors
- Usually requires automated theorem prover
  - These are hard to use
  - Require lots of experience
  - Remember loop check? That was easy.
  - May *Not* produce an answer (undecidable)

Model Checking

- Exhaustively checks all states of a finite state machine.
- Can be automated
- Always terminates with a yes/no answer
- Often provides counter-example of bad behavior
- Requires a model. Doesn’t work well on real code.

Unfolding a State Machine

This is what we can do with a state machine

Example path: A,B,A,B,C,C,C,…

This is an infinite tree
What is Model Checking?

Really, a Kripke Structure is used for model checking. A Kripke Structure is a graph of states with transitions where each state is labeled with properties true in that state.

Passes, or we get counter-example

What Can Model Checking Do?

- Determine if something always happens
  - Or, eventually fails to happen
- Determine if something eventually happens
  - Or, it never happens
- Determine if a state can be reached at all
- Determine if a series of states form an infinite loop
- Sometimes, run the model in simulation

How Can These Be Used?

Specifying Important Properties

- Safety properties:
  - Nothing “bad” ever happens
  - Formalized using state invariants
    - execution never reaches a “bad” state
- Liveness properties:
  - Something “good” eventually happens
  - Formalized using temporal logic
    - special logic for describing sequences
**The Model Checker “SPIN”**

Steps to follow to perform Model Check

- Code the model in the language **Promela**
- Run model through SPIN and produce C code
  - Produces "model" to execute
- Specify properties
  - "never cases"
  - reachability
  - presence of loops
- Execute Model

---

**Promela - Procedure**

- Declares a procedure

```promela
active procedure foo()
{
  int x, y, z;
  x = 1;
  y = 2;
  z = x + y;
  printf("the value of z is \%d\n", z);
}
```

More or less standard "C" syntax

---

**Promela - Guards**

- Guard blocks until it can execute.
  - Any statement can be a guard

Guards are used extensively in Promela.
By convention, the first statement is called a "guard", but
a sequence can be a guard too...

- `state = idle -> ready -> count > 16 -> state = go;`
  - tests conditions sequentially
### Promela - IF vs DO

```
do
  :: cond1 -> stat1;
  :: cond2 -> stat2;
  :: cond3 -> stat3;
  od
```

Continually loops executing the statement with true guard. If none true, waits until one is true.

```
if
  :: cond1 -> stat1;
  :: cond2 -> stat2;
  :: cond3 -> stat3;
fi
```

Waits until one of the guards is true, then executes the statement and continues. If none true, **if-** hangs.

---

### Breaking loops and non-determinism

```
init
|
  int x = 0;
  do
    :: printf("value of x is \%d\n", x) -> x++;  
    :: x == 0 -> break;
  od;
  printf("done\n");
```

Notice non-deterministic execution

```
break gets out of loop.
```

---

### Sending Messages

Declare a channel `chan <chan name> = [<size>] of (message type)`:

- Send a message: `chan/value;`
- Receive a message: `chan?value;`

*<size>* is length of queue. 0 means no queue; processes must "sync up" on the send/receive pair.
Message Example

```c
mt type
{ hello, goodbye } ;
ch event = [0] of { mt type } ;
active posttype one()
{
    printf("proc one waiting for hello \n");
    event? hello -> event? goodbye
    printf("proc one got hello and sent goodbye()\n");
}
active posttype two()
{
    printf("proc two sending hello\n");
    event+ hello;
    printf("proc two now looking for goodbye()\n");
    event? goodbye =>
        printf("proc two got goodbye()\n");
}
```

Hangar Motor Controller

- Motor runs a large bi-fold aircraft door
- "up" / "down" starts motor, if not at up/down limit
- Error if not up to speed in 5 sec
  - Don't allow restart for 10 secs if error
- When a limit is reached, turn off the motor
- "stop" must stop motor immediately
  - "stop" doesn't supercede reset after error


Hangar Door Model (1st 1/2)

```c
active
    main()
    {
        state = sidle;
        do
            :: (state == sidle) -> printf("in state idle\n");
            if :: button?down -> !vdownlimit ->
                printf("selecting down\n");
                state = sstart;
            fi;
            :: button?up -> !vuplimit ->
                printf("selecting up\n");
                state = sstart;
            fi;
            :: (state == sstart) -> printf("in state start\n");
                printf("start coil on\n");
                if :: button?stop ->
                    printf("start coil off; run coil off\n");
                    state = sidle;
                fi;
                :: event?vuplimit -> state = shold
                :: event?downlimit -> state = shold;
                :: event?speed -> state = srun;
                :: event?motortimeout -> state = sfail;
            fi;
        od;
    }
```

Hangar Door Model (2nd 1/2)

```c
:: (state == srun) -> printf("in state run\n");
    if :: button?stop ->
        printf("start coil off; run coil off\n");
        state = sidle;
    :: event?uplimit -> state = shold
    :: event?downlimit -> state = shold;
    fi;
:: (state == sfail) -> printf("in state sfail\n");
    if :: event?ten_sec_timeout -> state = shold;
    fi;
:: (state == shold) -> printf("in state hold\n");
    button?stop -> state = sidle;
od;
```

Timeout Scenario

```c
/* States */
    mtype {sidle, sstart, srun, sfail, shold};
/* events */
    mtype {uplimit, downlimit, motortimeout, ten_sec_timeout, speed};
/* button events */
    mtype {up, down, stop};
    mtype state;
    chan event = [0] of {
        mtype
    };
    chan button = [0] of {
        mtype
    };
    bit vuplimit = 0;
    bit vdownlimit = 0;
init
    { button!up;
        printf("sent up button\n");
        event!motortimeout;
        printf("sent motor timeout\n");
        event!ten_sec_timeout;
        printf("sent ten sec timeout\n");
        button!stop;
        printf("sent button stop\n");
    }
```
Output From Model (1)

```c
void main() {
    state = sidle;
    do {
        if (state == sidle) {
            printf("in state idle
");
            if (button?down) {
                printf("sent down button
");
                state = sstart;
            } else if (button?up) {
                printf("sent up button
");
                state = sstart;
            }
        } else if (state == sstart) {
            printf("in state start
");
            printf("start coil on
");
            if (button?stop) {
                printf("start coil off; run coil off
");
                state = sidle;
            }
        } else if (state == srun) {
            printf("in state run
");
            if (button?stop) {
                printf("start coil off; run coil off
");
                state = sidle;
            }
        } else if (state == shold) {
            printf("in state hold
");
            button?stop {
                state = sidle;
            }
        } else if (state == sfail) {
            printf("in state fail
");
            if (event?ten_sec_timeout) {
                state = sidle;
            }
        } else if (state == sidle) {
            printf("in state idle
");
        }
    } while (state != sidle);
}
```

Output From Model (2)

```c
:: (state == sidle) -> print("Sent up button
");
:: (state == sstart) -> print("Start coil on
");
:: (state == srun) -> print("Start coil off; run coil off
");
:: (state == shold) -> print("Sent button stop
");
:: (state == sfail) -> print("Sent ten sec timeout
");
```

How to Make a Model From a State Machine

**Choice 1:** Use `do` and state variable

```c
do {
    (state == idle) -> ...
    ...
    od;
```

**Choice 2:** Use `goto`, labels

```c
state1;  // event?foo -> goto state2;
state2;  // event?bar-> goto state1
```
Transitions

Within a state, channels are not quite right, but can be useful. But, we need “choice” construct for multiple transitions:

```
state:
if
  :: event?one -> ...
  :: event?two -> ...
  :: (foo == bar) -> ...
fi
```

This picks up choices, and waits until one is ready.

```
state:
  event?one -> ...
  event?two -> ...
  (foo == bar) -> ...
```

This is wrong! Sequentially waits for each condition.

Example State Machine

```
state = {pwr_on, pwr_off, fan_on, fan_off};
chan event = [0] of {
  state
};
init
  event!pwr_on;
  event!fan_on;
  event!pwr_off;
} active
} proctype fan_controller()
{
  off:
    printf("in state off
");
    event?pwr_on -> goto power;
  power:
    printf("in state power
");
    if :: event?fan_on -> goto fanon;
    :: event?pwr_off -> goto off;
    fi;
  fanon:
    printf("in state fanon
");
    if :: event?fan_off -> goto power;
    :: event?pwr_off -> goto off;
    fi;
}
```

How to Make a Composite State

- Could “flatten” state machine
  - But this is not aesthetically pleasing
- Would like encapsulation
  - proctypes are only construct available
- proctypes are concurrent. How to sync?
- Need to handle transitions to more complex than simple return
  - Composite states can transition anywhere
Simulating a “call”

```haskell
chan wait = [5] of {int, mtype};
chan event = [5] of {mtype};
mtype = {ok, state2};
active proctype one()
{
  int pid;
mtype ns;
pid = run two();
wait? eval(pid), ns;
printf("two has returned
");
event?ok -> printf("got queued event 1
");
event?ok -> printf("got queued event 2
");
if :: ns == state2 -> printf("next state is state2
") fi;
}
proctype two()
{
  int i;
  printf("now in proc two
");
event!ok;
event!ok;
printf("two sent two events
");
wait!_ pid, state2;
}
```

Here is the wait, and return of next state. eval turns variable into constant.

Here is “return”, or state exit. Passes “next state”

---

Example Execution

```text
now in proc two

```

---

How to Make a Class

- Same problems as composite state
  - Must have encapsulation
    - Implies proctype again
  - Need concurrency between classes
    - proctype works for this
  - All instance variables visible to this state machine
    - If composite states in class, need to share between proctypes
  - Can use a structure for instance variables
Representing Class Structure

Use a `proc type` for the class. The Promela code will represent the top level behavior.

```
proc type Foo()
{
    // code for top level
}
```

Put instance variables in a global `typedef` so they can be accessed by each composite state.

```
typedef Foo_T {
    int x;
    int y;
    bool sw;
}
```

```
Foo_T Foo_V;
```

```
Foo_V.sw -> Foo_V.x = Foo_V.y + 1;
```

Class Instance Variables

For this class, instance variables are declared like this and made public.

```
Foo x int
  y int
  sw bool
```

Instantiated like this:

```
Foo_T Foo_V;
```

Used like this:

```
Foo_V.sw = Foo_V.x = Foo_V.y + 1;
```

Verification With SPIN

- Assert
- State Reachability
- Never claims
  - Liveness
  - Safety
How to Make a Verifier By Hand (Unix)

1. `spin -a model.pr` → Produce the C code for the model. `pan.c` is always the output file (along with several other `pan` files).
2. `gcc -DNOREDUCE pan.c pan` → Compile the code. `NOREDUCE` avoids possible verifier problems.
3. `pan -a` → Run the verifier program. Various options can be passed to the verifier.

Assertions

```c
  :: (state == sidle) -> printf("in state idle
");
  if :: button?down-> !vdownlimit ->
    printf("selecting down
");
    state = sstart;
  :: button?up -> !vuplimit ->
    printf("selecting up
");
    state = sstart;
  fi;
  :: (state == sstart) ->
    printf("in state start
");
    printf("start coil on
");

/*  ===== ASSERT  ======  */
  assert(!vuplimit && !vdownlimit);
  if :: button?stop ->
    printf("start coil off; run coil off
");
    state = sidle;
  :: event?uplimit -> state = shold
  :: event?downlimit -> state = shold
  :: event?speed -> state = srun
  :: event?motortimeout -> state = sfail
  fi;
```

Assertion applies only at this point

Any logical condition can be asserted

Assertion Trace Output

```
Full state space search for:
  - no assertions
  - assertion violations
  - assertion cycles
  - unselected

reachability 25 steps, depth reached 10: groups:
  0
  18 states, nested 12
  43 states, unreach
  0 exceptions (1 unreachunmatched)
  0 exceptions (1 unreachable)
  0 exceptions (0 unreachunmatched)
  0 exceptions (0 unreachable)
  1.493 Mbyte

No comment, so assertion succeeded.
```

12c-ModelChecking

12c-ModelChecking
**Assertion Failure**

```c
// (state == sidle) -> printf("in state idle\n");
end0:
if :: button?down -> !vdownlimit ->
printf("selecting down\n");
state = sstart;
fi;
:: (state == sstart) ->
printf("in state start\n");
printf("start coil on\n");
/*  ===== ASSERT  ======  */
assert(vuplimit && !vdownlimit);
if :: button?stop ->
printf("start coil off; run coil off\n");
state = sidle;
:: event?uplimit -> state = shold;
:: event?downlimit -> state = shold;
:: event?speed -> state = srun;
:: event?motortimeout -> state = sfail;
fi;
Remove '!' to make the assertion fail.
```

**Assertion Trace Output**

```
spin: assertion violated (vuplimit && !vdownlimit) (at depth 11)
spin: trail ends after 11 steps
#processes: 2
state = sstart
vuplimit = 0
vdownlimit = 0
spin: trail ends after 11 steps
```

**Assertion Trace**

```
9: proc 0 [init] line 62 "door.pr" (state 16)
[assertion violated]
10: proc 1 (main) line 62 "door.pr" (state 16)
[assertion violated]
11: proc 0 [init] line 63 "door.pr" (state 16)
[assertion violated]
12: proc 1 (main) line 63 "door.pr" (state 16)
[assertion violated]
13: proc 0 [init] line 64 "door.pr" (state 16)
[assertion violated]
14: proc 0 [init] line 66 "door.pr" (state 29)
15: proc 1 (main) line 66 "door.pr" (state 29)
[assertion violated]
```

It does show us the content of the failing assertion and provides a source-line number.
Reachability

unreachable in proctype init:

unreachable in proctype main:

unreachable in proctype main:

unreachable in proctype main:

Everything in "init" was executed.

13c-ModelChecking

Reachability (cont)

active proctype main();

| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |
| state = sidle; |

Not reached because this scenario didn't use the "down" button.

13c-ModelChecking

Reachability (cont)

Nor was a "stop" button invoked.

13c-ModelChecking
### Reachability (cont)

```c
1 (state == state_run) -> printf("in state run
");
2 if (button?stop) ->
3    printf("start coil off; run coil off
");
4    state = sidle;
5
6 event?uplimit -> state = shold;
7 (state == state_sfail) -> printf("in state sfail
");
8 if (event?ten_sec_timeout) -> state = shold;
9 (state == state_hubold) ->
10    printf("in state hubold
");
11    button?up -> state = sidle;
12}
```

### LTL Properties

- `[] p` means "always, p is true"
- `< > p` means "eventually, p is true"

Other operators are as in regular logic

- `[] (stop => <> idle_state)`
  - It is always the case that:
  - a stop button implies eventually idle_state will be reached

### Negating LTL Operators

Remember these?

- `![] p` means `<> !p`
- `<> p` means `[] <> p`

<table>
<thead>
<tr>
<th>P</th>
<th>Negation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[] p</code></td>
<td><code>![] !p</code></td>
<td><code>p never shows up</code></td>
</tr>
</tbody>
</table>
Subtly of <> and []

[] means “always, henceforth From Every State”

<> means “somewhere along the execution path” and can be several times, or just once.

The Controller Diagram

How to Drive the Model Through All Scenarios
The Choices this “init” Provides

Defining a Property with XSPIN

Trace of Failing Check
Trace of Failing Check (cont)

- Sends a motor timeout
- Goes to state sfail
- Gets a "stop" and can't proceed.

The Path Chosen....

- If we are here when "stop" is pushed, we won't transition to "idle".
Modified Property

Define a new variable: catches 's fail' state

Change property to exclude 's fail' state

... And this succeeds

New Property: Checking Safety

New Property: uplimit switch should mean "hold" state is entered

Same form as before: [ ] (uplimit -> <> state==hold)

Safety Property - Fails

Why did this fail?
It shouldn't have!
## A Look at the Trace

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-11</td>
<td>Model checking code showing the trace of a process.</td>
</tr>
<tr>
<td>13-20</td>
<td>Code listing showing the state transitions and actions.</td>
</tr>
</tbody>
</table>

## A Look at the Trace (cont)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-30</td>
<td>Continuation of the tracing code, showing more state transitions.</td>
</tr>
</tbody>
</table>

## Subtly of <> and []

[<>] means 'always, henceforth From Every State'

[ ] means 'somewhere along the execution path' and can be several times, or just once.

The problem is, once we are back in state 'idle', we’ll send no more signals, so ‘vuplimit’ is true, but from ‘sidle’, we’ll never enter ‘shold’ again.

The property fails.
A Technique to Get Around Tricky LTL

We can add counters and indicators to the model to pick up events

\[(\text{limit} \implies \text{holdcount} > 0)\]

Revised Property Test

- property to test
- new defines
- succeeds!!