



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)



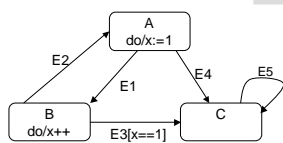
Model Checking

- 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.

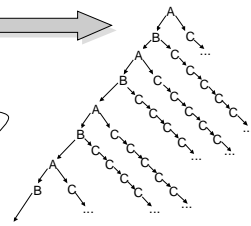


Unfolding a State Machine

This is what we can do with a state machine



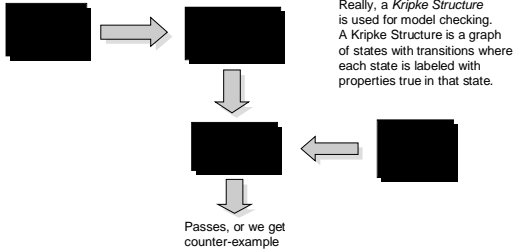
Example path: A,B,A,B,C,C,C,.....



This is an infinite tree



What is Model Checking?





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

```

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

```

Declares a procedure
 Declares variables
 Variable assignment
 More or less standard "C" syntax



Promela - Guards

These are equivalent

```

{ (state == idle) ; state = go;
  (state == idle) -> state = go;
}

```

Guard blocks until it can execute.
 Any statement can be a guard
 This is syntactic "sugar" for reading convenience

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 \Longrightarrow



Promela - IF vs DO

```
do
:: cond1 -> stmt1;
:: cond2 -> stmt2;
:: cond3 -> stmt3;
od
```

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

```
if
:: cond1 -> stmt1;
:: cond2 -> stmt2;
:: cond3 -> stmt3;
fi
```

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



Breaking loops and non-determinisim

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

```
c:\spin>SPIN349 test.pr
value of x is 0
value of x is 1
value of x is 2
value of x is 3
value of x is 4
value of x is 5
value of x is 4
value of x is 4
value of x is 5
value of x is 4
value of x is 5
value of x is 4
value of x is 3
value of x is 2
value of x is 1
value of x is 2
value of x is 1
done
1 processes created
c:\spin>
```



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.



Timeout Scenario

```

/* States */
stypetype {sidle, sstart, srun, sfail, shold};
/* events */
mtypetype {uplimit, downlimit, motortimeout, ten_sec_timeout, speed};
/* button events */
mtypetype {up,down,stop};
stypetype state;

chan event = [0] of {mtypetype};
chan button = [0] of {mtypetype};

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");
}

```

Simulates the environment



Output From Model (1)

```

active proctype main()
{
  state = sidle;
  do
  :: (state == sidle) -> printf("in state idle\n");
  if
  :: button?down -> {vdownlimit ->
    printf("#selecting down\n");
    state = sstart;
  :: 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;
  :: event?vuplimit -> state = shold;
  :: event?vdownlimit -> state = shold;
  :: event?speed -> state = srun;
  :: event?motortimeout -> state = sfail;
  fi;
}

```



Output From Model (2)

```

:: (state == srun) -> printf("in state run\n");
if
:: button?stop ->
printf("start coil off; run coil off\n");
state = sidle;
:: event?vuplimit -> state = shold;
:: event?vdownlimit -> 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;
}

```

Line 81

Line 25



How to Make a Model From a State Machine

Choice 1:
Use `do` and state variable →

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

Choice 2:
Use `goto`, labels

```
state1:                                performs transition
    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
```

← 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

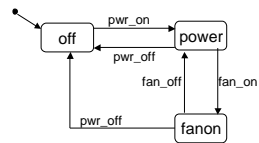
```
mtype = {pwr_on, pwr_off, fan_on, fan_off};
chan event = [0] of {mtype};

init
{
  event?pwr_on;
  event?fan_on;
  event?pwr_off;
}

active prototype fan_controller()
{
  off:
    printf("in state off\n");
    event?pwr_on -> goto power;

  power:
    printf("in state power\n");
    if
    :: event?fan_on -> goto fanon;
    :: event?pwr_off -> goto off;
    fi;

  fanon:
    printf("in state fanon\n");
    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”

```
chan wait = {5} of {int, mtype}; ← notice compound message
chan event = {5} of {mtype};
mtype = {ok, state2};

active proctype one() ← one calls and waits for two
{
  int pid;
  mtype ns;

  pid = run two();
  wait??eval(pid, ns); ← Here is the wait, and return of next state.
  printf("two has returned\n");   eval turns variable into constant.
  event?ok -> printf("got queued event 1\n");
  event?ok -> printf("got queued event 2\n");
  if :: ns == state2 -> printf("next state is state2\n") fi;
}

proctype two()
{
  int i;

  printf("now in proc two\n");
  event!ok;
  event!ok;
  printf("two sent two events\n");
  wait!_pid, state2;
}

```



Example Execution

```
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\n");
  event?ok -> printf("got queued event 1\n");
  event?ok -> printf("got queued event 2\n");
  if :: ns == state2 -> printf("next state is state2\n") fi;
}

proctype two()
{
  int i;

  printf("now in proc two\n");
  event!ok;
  event!ok;
  printf("two sent two events\n");
  wait!_pid, state2;
}

```

```
c:\spin>SPIN349 composite.pr
now in proc two
two sent two events
two has returned
got queued event 1
got queued event 2
next state is state2
2 processes created

c:\spin>

```



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 **proctype** for the class.
The Promela code will represent the top level behavior.

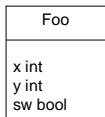
```

proctype Foo()
{
  (code for top level)
}
  
```

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



Class Instance Variables



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

```

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

Instantiated like this → `Foo_T Foo_V;`

Used like this → `Foo_V.sw->Foo_V.x = Foo_V.y + 1;`

*Instance variables have to be accessible across proctype boundaries (composite states) because each composite state can access each instance variable.
Declare the instance variables at the top of the Promela source file.*
