Introduction to FSM-test generation and reachability analysis

Brian Nielsen bnielsen@cs.auc.dk

Department of Computer Science,
Aalborg University, Denmark
Menu

• (Basic definitions and fundamental results)
• Conformance Testing with FSMs
• Model checking
• Reachability Analysis
Finite State Machine (Mealy)

Inputs = \{\text{cof-but, tea-but, coin}\}
Outputs = \{\text{cof, tea}\}
States: \{q_1, q_2, q_3\}
Initial state = q_1
Transitions = \{
(q_1, \text{coin}, -, q_2),
(q_2, \text{coin}, -, q_3),
(q_3, \text{cof-but}, \text{cof}, q_1),
(q_3, \text{tea-but}, \text{tea}, q_1)
\}

Sample run:

\begin{align*}
q_1 \xrightarrow{\text{coin}/-} & q_2 \xrightarrow{\text{coin}/-} q_3 \xrightarrow{\text{cof-but}/\text{cof}} q_1 \xrightarrow{\text{coin}/-} \\
q_2 \xrightarrow{\text{coin}/-} & q_3 \xrightarrow{\text{cof-but}/\text{cof}} q_1
\end{align*}
Concepts

• Two states $s$ and $t$ are (language) equivalent iff
  • $s$ and $t$ accepts same language
  • has same traces: $tr(s) = tr(t)$

• Two Machines $M_0$ and $M_1$ are equivalent iff initial states are equivalent

• A minimized / reduced $M$ is one that has no equivalent states
  • for no two states $s, t$, $s \neq t$, $s$ equivalent $t$
Fundamental Results

• Every FSM may be determinized accepting the same language (potential explosion in size).

• For each FSM there exist a language-equivalent minimal deterministic FSM.

• FSM’s are closed under $\cap$ and $\cup$

• FSM’s may be described as regular expressions (and vise versa)
Given a specification FSM $M_S$

a (black box) implementation FSM $M_I$

determine whether $M_I$ conforms to $M_S$.

i.e., $M_I$ behaves in accordance with $M_S$

i.e., whether outputs of $M_I$ are the same as of $M_S$

i.e., whether the reduced $M_I$ is equivalent to $M_S$
Possible Errors

- output fault
- extra or missing states
- transition fault
  - to other state
  - to new state
State Machine: FSM Model
State Machine: FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple
State Machine: FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple

\[ M = ( S, I, O, \delta, \lambda ) \]
State Machine: FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple

\[ M = (S, I, O, \delta, \lambda) \]

\( S \) finite set of states
State Machine: FSM Model

FSM - Finite State Machine - or *Mealy Machine* is 5-tuple

\[ M = (S, I, O, \delta, \lambda) \]

- **S** \quad finite set of states
- **I** \quad finite set of inputs
State Machine: FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple

\[ M = (S, I, O, \delta, \lambda) \]

- **S** finite set of states
- **I** finite set of inputs
- **O** finite set of outputs
State Machine: FSM Model

Finite State Machine - or Mealy Machine is 5-tuple

\[ M = (S, I, O, \delta, \lambda) \]

- \( S \) finite set of states
- \( I \) finite set of inputs
- \( O \) finite set of outputs
- \( \delta : S \times I \rightarrow S \) transfer function
State Machine: FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple

\[ M = \left( S, I, O, \delta, \lambda \right) \]

- \( S \) finite set of states
- \( I \) finite set of inputs
- \( O \) finite set of outputs
- \( \delta : S \times I \rightarrow S \) transfer function
- \( \lambda : S \times I \rightarrow O \) output function
State Machine : FSM Model

FSM - Finite State Machine - or Mealy Machine is 5-tuple

\[ M = ( S, I, O, \delta, \lambda ) \]

- \( S \) finite set of states
- \( I \) finite set of inputs
- \( O \) finite set of outputs
- \( \delta : S \times I \rightarrow S \) transfer function
- \( \lambda : S \times I \rightarrow O \) output function

Natural extension to sequences:

\[ \delta : S \times I^* \rightarrow S \]
\[ \lambda : S \times I^* \rightarrow O^* \]
Restrictions
Restrictions

FSM restrictions:
FSM restrictions:

- **deterministic**

  \[ \delta : S \times I \to S \text{ and } \lambda : S \times I \to O \text{ are functions} \]
Restrictions

FSM restrictions:

• *deterministic*
  \[ \delta : S \times I \to S \] and \[ \lambda : S \times I \to O \] are *functions*

• *completely specified*
  \[ \delta : S \times I \to S \] and \[ \lambda : S \times I \to O \] are *complete* functions
  (empty output is allowed; sometimes implicit completeness)
Restrictions

FSM restrictions:

- **deterministic**
  \[ \delta : S \times I \to S \text{ and } \lambda : S \times I \to O \text{ are functions} \]

- **completely specified**
  \[ \delta : S \times I \to S \text{ and } \lambda : S \times I \to O \text{ are complete functions} \]
  ( empty output is allowed; sometimes implicit completeness )

- **strongly connected**
  from any state any other state can be reached
Restrictions

FSM restrictions:

• *deterministic*
  \[ \delta : S \times I \rightarrow S \text{ and } \lambda : S \times I \rightarrow O \text{ are functions} \]

• *completely specified*
  \[ \delta : S \times I \rightarrow S \text{ and } \lambda : S \times I \rightarrow O \text{ are complete functions} \]
  (empty output is allowed; sometimes implicit completeness)

• *strongly connected*
  from any state any other state can be reached

• *reduced*
  there are no equivalent states
Desired Properties

• Nice, but rare / problematic
  • status messages: Assume that tester can ask implementation for its current state (reliably!!) without changing state
  • reset: reliably bring SUT to initial state
  • set-state: reliably bring SUT to any given state
FSM Transition Testing
FSM Transition Testing

- Make test case for every transition in spec separately:
FSM Transition Testing

- Make test case for every transition in spec separately:

![Diagram showing transition between S1 and S2 with input a? / x!](Diagram.png)
FSM Transition Testing

• Make test case for every transition in spec separately:

  1. Go to state $S_1$
  2. Apply input $a?$
  3. Check output $x!$
  4. Verify state $S_2$ (optionally)
FSM Transition Testing

- Make test case for every transition in spec separately:
  - Test transition:
    1. Go to state $S_1$
    2. Apply input $a$?
    3. Check output $x$!
    4. Verify state $S_2$ (optionally)

- Test purpose: “Test whether the system, when in state $S_1$, produces output $x$! on input $a$? and goes to state $S_2$”
Transition Testing – 1
Transition Testing –1

- To test token? / coin!
  - go to state 5: set-state 5
  - give input token? check output coin!
  - verify state: send status? check status=10
• To test token? / coin!:
  go to state 5: set-state 5
  give input token? check output coin!
  verify state: send status? check status=10

• To test token? / coin!:
  go to state 5: set-state 5
give input token? check output coin!
verify state: send status? check status=10

4 * |S| * | I | test cases remaining
FSM Transition Tour
FSM Transition Tour

- Make Transition Tour that covers every transition (in spec)
• Make Transition Tour that covers every transition (in spec)
FSM Transition Tour

- Make Transition Tour that covers every transition (in spec)
FSM Transition Tour

- Make Transition Tour that covers every transition (in spec)

Test input sequence:
FSM Transition Tour

- Make Transition Tour that covers every transition (in spec)

Test input sequence:

+ check expected outputs and target state by status message
Transition Testing -1
Transition Testing -1

• Go to state $S_5$ :
Transition Testing -1

- Go to state S5 :
- No Set-state property???
Transition Testing -1

- Go to state S5:
- No Set-state property???
  - use reset property if available
Transition Testing -1

- Go to state S5:
- No Set-state property???
  - use reset property if available
  - go from S0 to S5
    (always possible because of determinism and completeness)
Transition Testing -1

• Go to state $S_5$ :
• No Set-state property???
  • use $reset$ property if available
  • go from $S_0$ to $S_5$
    ( always possible because of determinism and completeness )
• or:
Transition Testing -1

• Go to state S5:
• No Set-state property???
  • use reset property if available
  • go from S0 to S5
    ( always possible because of determinism and completeness )
• or:
  • synchronizing sequence brings machine to particular known state, say S0, from any state
Transition Testing -1

• Go to state $S_5$ :
• No Set-state property???
  • use reset property if available
  • go from $S_0$ to $S_5$
    ( always possible because of determinism and completeness )
• or:
• $synchronizing sequence$ brings machine to particular known state,
  say $S_0$, from any state
• ( but synchronizing sequence may not exist )
Transition Testing -1

token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing -1

synchronizing sequence : token? coffee?

To test token? / coin! : go to state 5 by : token? coffee? coin?
Transition Testing -1

synchronizing sequence: token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing -1

synchronizing sequence: token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing -1

synchronizing sequence: token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing -1

synchronizing sequence: token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing -1

synchronizing sequence : token? coffee?

token? / coin!

To test token? / coin! : go to state 5 by : token? coffee? coin?
Transition Testing –2,3
Transition Testing –2,3

Diagram showing states labeled with '0', '5', and '10', with transitions labeled as 'coffee?', 'coin?', 'token?', and 'coffee!'.
Transition Testing –2,3

• To test `token? / coin!`:
  1. go to state 5 by: `token? coffee? coin?`
  2. give input `token?`
  3. check output `coin!`
  4. verify that machine is in state 10
Transition Testing-4
Transition Testing-4

• No Status Messages??
Transition Testing-4

• No Status Messages??
• State identification: What state am I in??
Transition Testing-4

• No Status Messages??
• State identification: What state am I in??
• State verification: Am I in state s?
  • Apply sequence of inputs in the current state of the FSM such that from the outputs we can
    • identify that state where we started; or
    • verify that we were in a particular start state
  • Different kinds of sequences
    • UIO sequences (Unique Input Output sequence, SIOS)
    • Distinguishing sequence (DS)
    • W-set (characterizing set of sequences)
    • UIOv
    • SUIO
    • MUIO
    • Overlapping UIO
Transition Testing-4
Transition Testing-4

State check:
Transition Testing-4

State check:

- UIO sequences (verification)
Transition Testing-4

State check:

- UIO sequences (verification)
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
Transition Testing-4

State check:

• UIO sequences (verification)
  • sequence $x_s$ that distinguishes state $s$ from all other states:
    for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  • each state has its own UIO sequence
Transition Testing-4

State check:

• UIO sequences (verification)
  • sequence \( x_s \) that distinguishes state \( s \) from all other states:
    for all \( t \neq s \):
    \[ \lambda(\ s, \ x_s) \neq \lambda(\ t, \ x_s) \]
  • each state has its own UIO sequence
  • UIO sequences may not exist
State check:

- **UIO sequences (verification)**
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    
    \[
    \lambda(s, x_s) \neq \lambda(t, x_s)
    \]
    
    for all $t \neq s$.
  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence (identification)**
Transition Testing-4

State check :

• UIO sequences (verification)
  • sequence $x_s$ that distinguishes state $s$ from all other states:
    for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  • each state has its own UIO sequence
  • UIO sequences may not exist

• Distinguishing sequence (identification)
  • sequence $x$ that produces different output for every state:
    for all pairs $t, s$ with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$
Transition Testing-4

State check:

- **UIO sequences** (verification)
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    
    $$\lambda(s, x_s) \neq \lambda(t, x_s)$$

  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence** (identification)
  - sequence $x$ that produces different output for every state:
    
    $$\lambda(s, x) \neq \lambda(t, x)$$

  - a distinguishing sequence may not exist
State check:

- **UIO sequences** *(verification)*
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    - for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence** *(identification)*
  - sequence $x$ that produces different output for every state:
    - for all pairs $t, s$ with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$
  - a distinguishing sequence may not exist

- **$W$ - set of sequences** *(identification)*
Transition Testing-4

State check :

- **UIO sequences** (verification)
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    - for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence** (identification)
  - sequence $x$ that produces different output for every state:
    - for all pairs $t, s$ with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$
  - a distinguishing sequence may not exist

- **W - set of sequences** (identification)
  - set of sequences $W$ which can distinguish any pair of states:
    - for all pairs $t \neq s$ there is $x \in W$: $\lambda(s, x) \neq \lambda(t, x)$
State check:

- **UIO sequences (verification)**
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence (identification)**
  - sequence $x$ that produces different output for every state:
    for all pairs $t, s$ with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$
  - a distinguishing sequence may not exist

- **W - set of sequences (identification)**
  - set of sequences $W$ which can distinguish any pair of states:
    for all pairs $t \neq s$ there is $x \in W$: $\lambda(s, x) \neq \lambda(t, x)$
  - $W$ - set always exists for reduced FSM
Transition Testing-4: UIO
Transition Testing-4: UIO

UIO sequences
Transition Testing-4: UIO

UIO sequences

0
- coffee? / -
- token? /
- coffee? / coffee!
10
- token? / token!
- coin? / coin!
5
- coin? / -
- coffee? / -
- token? / coin!
Transition Testing-4: UIO

UIO sequences

state 0 : coin? / -  coffee? / -
state 5 : token? / coin!
state 10 : coffee? / coffee!
Transition Testing-4: DS

DS sequence
Transition Testing-4: DS

DS sequence

0

coffee? / -

10

token? / token!

5

coffee? / -

coin? / -

token? / coin!
Transition Testing-4: DS

DS sequence:

DS sequence: token? output state 0: -
output state 5: coin!
output state 10: token!
Transition Testing – 4 done
Transition Testing – 4 done

0 -> coffee? / -
5 -> coffee? / -
10 -> token? / token!

0 -> token? / -
10 -> coin? / coin!

0 -> token? / coin!
10 -> coin? / coin!
Transition Testing –4 done

• To test token? / coin!:
  
go to state 5: token? coffee? coin?
give input token? check output coin!
Apply UIO of state 10: coffee? / coffee!
Transition Testing –4 done

• To test token? / coin!
  
  go to state 5:  token?  coffee?  coin?
  give input token?  check output coin!

  Apply UIO of state 10:  coffee? / coffee!

Transition Testing - done
Transition Testing - done
- 9 transitions / test cases for coffee machine
- if end-state of one corresponds with start-state of next then concatenate
- different ways to optimize and remove overlapping / redundant parts
- there are (academic) tools to support this
FSM  Transition Testing
FSM Transition Testing

- Test transition :
  - Go to state S1
  - Apply input a?
  - Check output x!
  - Verify state S2
FSM Transition Testing

- Test transition:
  - Go to state S1
  - Apply input a?
  - Check output x!
  - Verify state S2
- Checks every output fault and transfer fault (to existing state)
FSM Transition Testing

- Test transition:
  - Go to state $S_1$
  - Apply input $a$?
  - Check output $x$!
  - Verify state $S_2$
- Checks every output fault and transfer fault (to existing state)
- If we assume that
  
  *the number of states of the implementation machine $M_I$ is less than or equal to*
  
  *the number of states of the specification machine to $M_S$.*

  then testing all transitions in this way

  leads to equivalence of reduced machines,
i.e., complete conformance
FSM Transition Testing

- Test transition:
  - Go to state $S_1$
  - Apply input $a$?
  - Check output $x$!
  - Verify state $S_2$
- Checks every output fault and transfer fault (to existing state)
- If we assume that

  \[
  \text{the number of states of the implementation machine } M_i \\
  \text{is less than or equal to} \\
  \text{the number of states of the specification machine to } M_s.
  \]

  then testing all transitions in this way
  leads to equivalence of reduced machines,
  i.e., complete conformance
- If not: exponential growth in test length in number of extra states.
State Coverage
State Coverage

• Make *State Tour* that covers every state (in spec!)
State Coverage

- Make *State Tour* that covers every state (in spec!)
State Coverage

- Make *State Tour* that covers every state (in spec!)
State Coverage

- Make *State Tour* that covers every state (in spec!)

Test sequence: coin? token? coffee?
Transition Coverage
Transition Coverage

- Make *Transition Tour* that covers every transition (in spec)
Transition Coverage

- Make *Transition Tour* that covers every transition (in spec)
Transition Coverage

- Make *Transition Tour* that covers every transition (in spec)
Transition Coverage

• Make *Transition Tour* that covers every transition (in spec)

Test input sequence:
UppAal

Modelling and certifying untimed systems
Tool Support (model checking)

System Description

Requirement

Tools: UPPAAL, visualSTATE, ESTEREL, SPIN, Statemate, FormalCheck, VeriSoft, Java Pathfinder, Telelogic…

Yes, Protypes Executable Code Test sequences

No! Debugging Information
Semaphore Solution?
Semaphore Solution?

1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?
Semaphore Solution?

1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?

1. $A[] \implies (\text{mc1.finished and mc2.finished}) \implies (\text{accountA} + \text{accountB} == 200)$
2. $E <> \text{mc1.critical_section and mc2.critical_section}$
3. $A[] \not\implies (\text{mc1.finished and mc2.finished}) \implies \text{not deadlock}$
Semaphore Solution?

1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?

1. A[\{mc1\textunderscore finished \text{ and } mc2\textunderscore finished\}] imply (accountA+accountB==200) ✓
2. E<> \text{mc1\textunderscore critical\_section and mc2\textunderscore critical\_section} ✓
3. A[\} not (mc1\textunderscore finished \text{ and } mc2\textunderscore finished) imply not deadlock  ÷
**Global shared variables:**

`int accountA, accountB;`

**Local control:**

- `system state = snapshot of each machines control location + local variables + global variables`

- `mc1.control=requestB, mc1.a=0, mc1.b=0, accountA=100`  
  - `mc2.control=requestB, mc2.a=0, mc2.b=0, accountB=100`
Process Interaction

- ! = Output, ? = Input
- Handshake communication
- Two-way

Coffee Machine

Lecturer

4 states

synchronization results in internal actions

4 states: Interaction constrain overall behavior

University = Coffee Machine || Lecturer

- LTS?
- How many states?
- Traces?
Broad-casts

- Two way handshake
  - `chan coin, cof, cofBut;`
- One to many
  - `broadcast chan join;`
  - sending: output join!
  - every automaton that listens to “join” moves
  - ie. every automaton with enabled “join?” transition moves in one step
  - may be zero!
Committed Locations

- Locations marked C
  - *No delay* in committed location.
  - Next transition must involve automata in *committed location*.

- Handy to model atomic sequences
- The use of committed locations reduces the number of states in a model, and allows for more space and time efficient analysis.

- S0 to s5 executed atomically
UppAal Navigator
Reachability Analysis

• Compute all possible execution sequences
• And consequently all states of the system
• Exhaustive search => proof
• Check if each state encountered has the (un)-desired property
• Each trace = a program execution
• Uppaal checks all traces

• Is count possibly 3 ?  E<> count==3
• Is count always 1 ?  A[] count==1

Int count:=1
Reachability Analysis

Passed:=Ø //already seen states
Waiting:={S_0} //states not examined yet

While(waiting!=Ø) {
    Waiting:=Waiting\{s_i}\n    if s_i \notin Passed
        whenever (s_j \rightarrow s_j) then
            waiting:=waiting \cup s_j
    }

Depth First: maintain waiting as a stack
Order: 0 1 3 6 7 4 8 2 5 9

Breadth First: maintain waiting as a queue
(shortest counter example)
Order: 0 1 2 3 4 5 6 7 8 9
Properties

- **Safety**
  - Nothing bad happens during execution
  - System never enters a bad state
  - Eg. mutual exclusion on shared resource

- **Liveness**
  - Something good eventually happens
  - Eventually reaching a good state
  - Eg. a process’ request for a shared resource is eventually granted
UPPAAL Property Specification Language

- $A[]\ p$
- $A<>\ p$
- $E<>\ p$
- $E[]\ p$
- $P \rightarrow q$

process location  data guards  clock guards

$p ::= a.l | gd | gc | p \land p |$
$p \lor p | \neg p | p \implies p |$
$(p) | \text{deadlock} \text{ (only for } A[], E<>)$

$A[] \ (mc1\text{.finished and mc2\text{.finished}) imply (accountA+accountB==200)}$

$E<> \ (mc1\text{.finished and mc2\text{.finished}) and (not (accountA+accountB==200))}$
Uppaal “Computation Tree Logic”

E<> p

Possible
Uppaal “Computation Tree Logic”

E<> p  Possible

A[] p  always
Uppaal “Computation Tree Logic”

- **E<> p** (Possible)
- **A[] p** (always)
- **E[] p** (potentially always)
Uppaal “Computation Tree Logic”

- **E<> p**
  - Possible

- **A[] p**
  - Always

- **E[] p**
  - Potentially always

- **A<> p**
  - Inevitable
Uppaal “Computation Tree Logic”

- **E<> p**  
  - Possible

- **E[] p**  
  - Potentially always

- **A[] p**  
  - Always

- **A<> p**  
  - Inevitable

- **p --> q**  
  - Leads-to
‘State Explosion’

problem

All combinations = exponential in no. of components

Provably theoretical intractable
Limitations to Reachability Analysis
Limitations to Reachability Analysis

- $n$ parallel FSMs
- $k$ states each
- $k=\wedge n$ states in parallel composition
- EXPONENTIAL GROWTH
- $10^2 = 100$
- $10^3 = 1000$
- $10^4 = 10000$
- $10^{10} = 10000000000$
Limitations to Reachability Analysis

- \( n \) parallel FSMs
- \( k \) states each
- \( k=^n \) states in parallel composition
- EXPONENTIAL GROWTH
  - \( 10^2 = 100 \)
  - \( 10^3 = 1000 \)
  - \( 10^4 = 10000 \)
  - \( 10^{10} = 10000000000 \)

State Space / Time Usage / Memory Usage

- Exhaustive feasible
- Controlled partial
- Random - low coverage

system size (#parallel processes)
What Influences System Size?

- Number of parallel processes
- Amount of non-determinism
- Queue Sizes
- Range of discrete data values
- Environment Assumptions
  - Speed
  - Kinds of messages that can be sent in what states
  - Data values
Counter Measures

• Use abstraction, simplification
  • Only model the aspects relevant for the property in question
• Economize with (loosely synch’ed) parallel processes
• Make precise assumptions and restrictions
• Range of data values
  • Use bounded data values: integer (0:4);
  • reset variables to initial value whenever possible
  • Avoid complex data structures
• Partial (controlled) search heuristics
  • Bit-State hashing
  • Limit search depth
  • Restrict scheduling
    • Priority to internal transitions over env input
    • Schedule process FIFO style rather than ALL interleavings
Does verification guarantee correctness?

- Only models verified, not (physical) implementations
- Made the right model?
- Properties correctly formulated
- The right properties?
- Enough properties?
- System size too large for exhaustive check

- Modelling effort itself revealing
- Increased confidence earlier
- Cheaper
- Even partial and random search increases confidence.
The Cruise Controller

User:
- engineOff, engineOn, acc, brake
- on, off, resume

Controller:
- enableControl,
- disableControl, recordSpeed

CruiseControl

SpeedControl:
- setThrottle
- speed

Engine
END