FSM-test generation

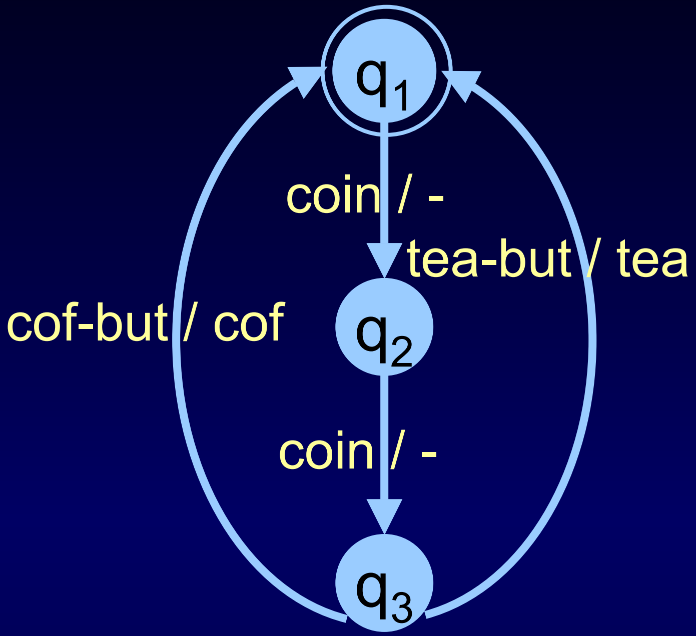
Brian Nielsen
bnielsen@cs.auc.dk
Department of Computer Science,
Aalborg University, Denmark

CSS

Menu

- Review of basic definitions and fundamental results
- Classical Deterministic Untimed (very) finite FSMs
- Conformance Testing with FSMs
  - Transition Testing
  - Synchronizing sequences
  - State identification and verification
  - State and transition covering sequences

Finite State Machine (Mealy)



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1

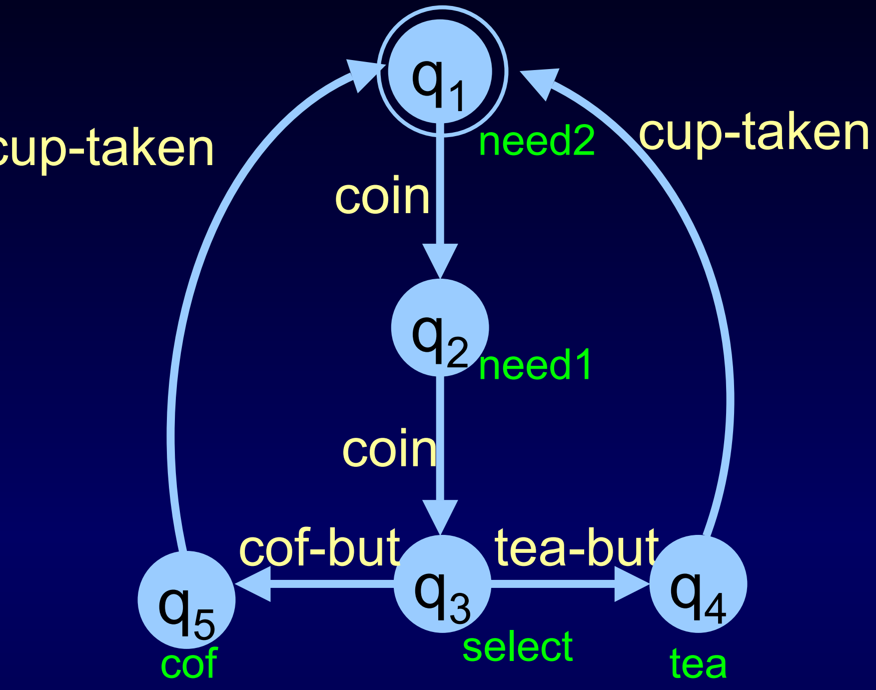
Inputs = {cof-but, tea-but, coin}
Outputs = {cof,tea}
States: $\{q_1,q_2,q_3\}$
Initial state = q_1
Transitions= {
$(q_1$, coin, -, $q_2)$,
$(q_2$, coin, -, $q_3)$,
$(q_3$, cof-but, cof, $q_1)$,
$(q_3$, tea-but, tea, $q_1)$
}

Sample run:

$$q_1 \xrightarrow{\text{coin/ -}} q_2 \xrightarrow{\text{coin/-}} q_3 \xrightarrow{\text{cof-but / cof}} q_1 \xrightarrow{\text{coin/ -}}$$

$$q_2 \xrightarrow{\text{coin/ -}} q_3 \xrightarrow{\text{cof-but / cof}} q_1$$

Finite State Machine (Moore)

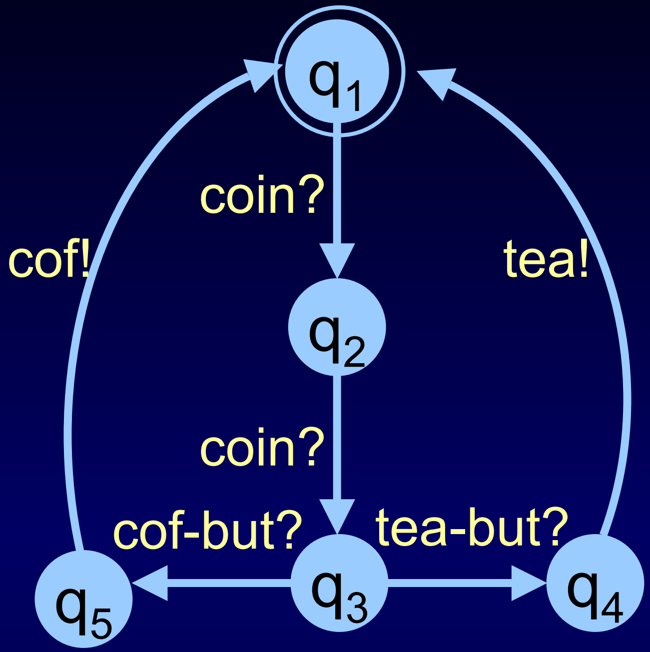


condition		effect	
current state	input	Activity	next state
q_1	coin	need2	q_2
q_2	coin	need1	q_3
q_3	cof-but	select	q_5
q_3	tea-but	select	q_4
q_5	cup-taken	cof	q_1
q_4	cup-taken	tea	q_1

Input sequence: coin.coin.cof-but.coin.coin.cof-but
Output sequence: need2.need1.select.cof. need2.need1.select.cof

need2=display shows “insert two coins”

IO-FSM



condition		effect
current state	action	next state
q_1	coin?	q_2
q_2	coin?	q_3
q_3	cof-but!	q_5
q_3	tea-but!	q_4
q_4	cof?	q_1
q_5	tea!	q_1

Inputs = {cof-but, tea-but, coin}
Outputs = {cof,tea}
States: {q_1,q_2,q_3}
Initial state = q_1
Transitions= {
(q_1, coin, q_2),
(q_2, coin, q_3),
(q_3, cof-but, q_5),
(q_3, tea-but, q_4),
(q_4, tea, q_1),
(q_5, cof, q_1)

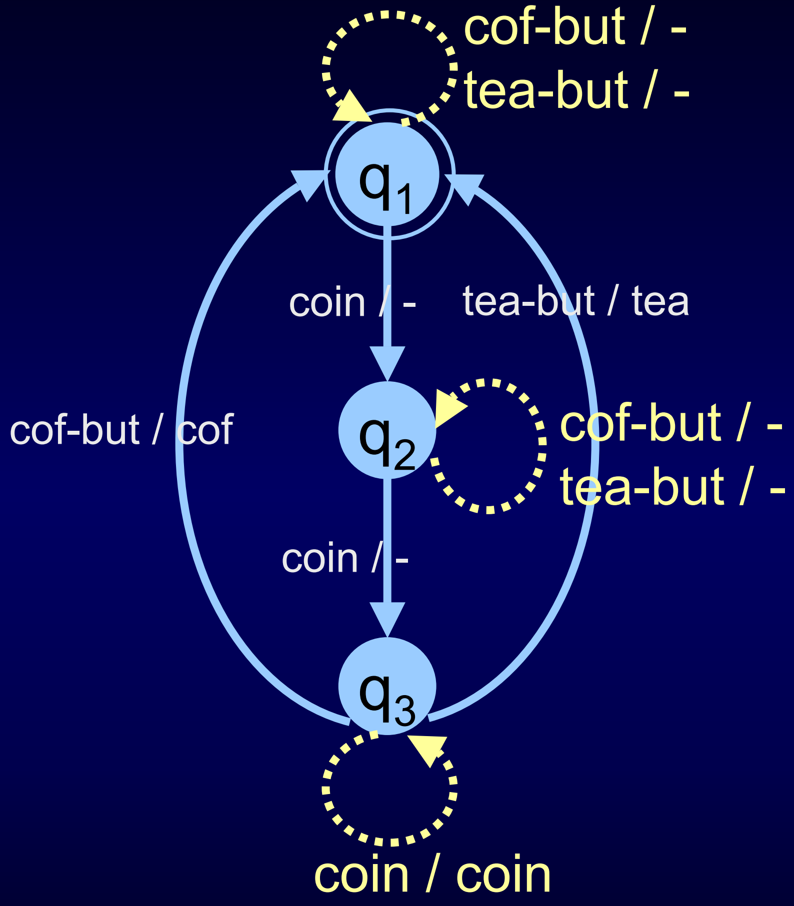
Sample run:

$$q_1 \xrightarrow{\text{coin?}} q_2 \xrightarrow{\text{coin?}} q_3 \xrightarrow{\text{cof-but?}} q_5 \xrightarrow{\text{cof}}$$

$$q_1 \xrightarrow{\text{coin?}} q_2 \xrightarrow{\text{coin}} q_3 \xrightarrow{\text{cof-but?}} q_5 \xrightarrow{\text{cof}} q$$

action trace: coin?.coin?.cof!-coin?.coin?.cof!
input sequence: coin.coin.coin.coin
Output sequence: cof.cof

Fully Specified FSM



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1
q_1	cof-but	-	q_1
q_1	tea-but	-	q_1
q_2	cof-but	-	q_2
q_2	tea-but	-	q_2
q_3	coin	coin	q_3

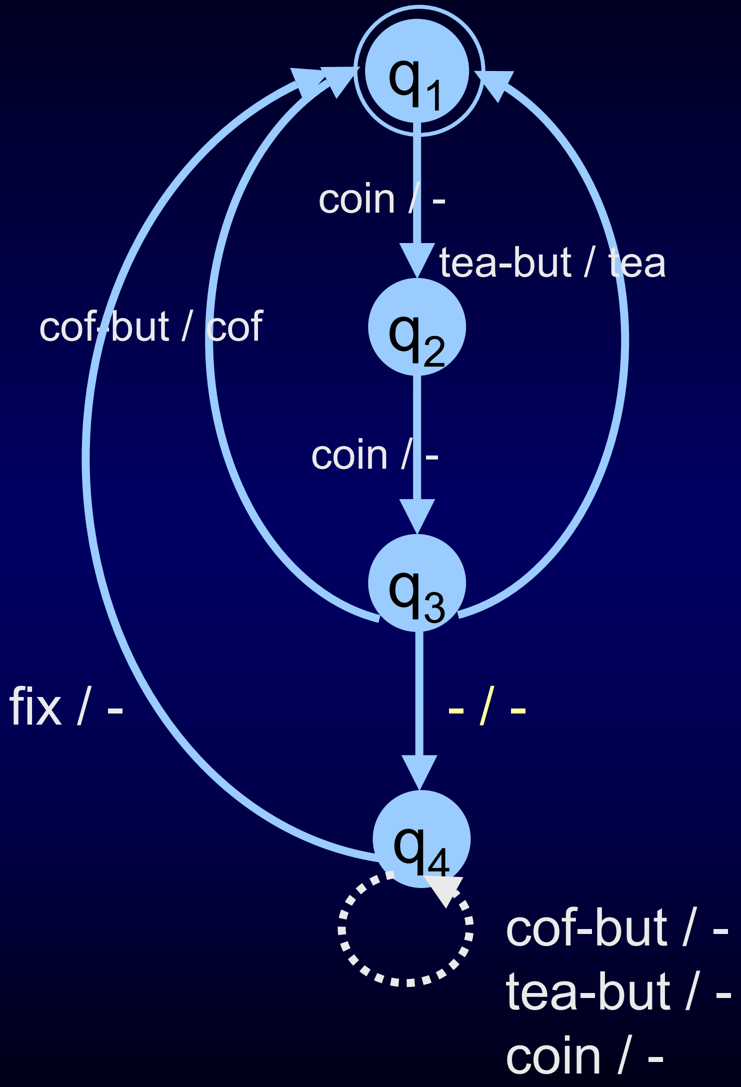
FSM as program 1

```
enum currentState {q1,q2,q3};
enum input {coin, cof_but,tea_but};
int nextStateTable[noStates][noInputs] = {
      q2,q1,q1,
      q3,q2,q2,
      q3,q1,q1 };

int outputTable[noStates][noInputs] = {
      0,0,0,
      0,0,0,
      coin,cof,tea};

While(Input=waitForInput()) {
  OUTPUT(outputTable[currentState,input])
  currentState=nextStateTable[currentState,input];

}
```

FSM as program 2

```
enum currentState {q1,q2,q3};
enum input {coin,cof,tea_but,cof_but};

While(input=waitForInput){
  Switch(currentState){
  case q1: {
      switch (input) {
        case coin: currentState=q2; break;
        case cuf_but:
        case tea_but: break;
        default: ERROR("Unexpected Input");
        }
       break;
  case q3: {
      switch(input) {
        case cof_buf: {currentState=q3;
                       OUTPUT(cof);
                       break;}
        …
```

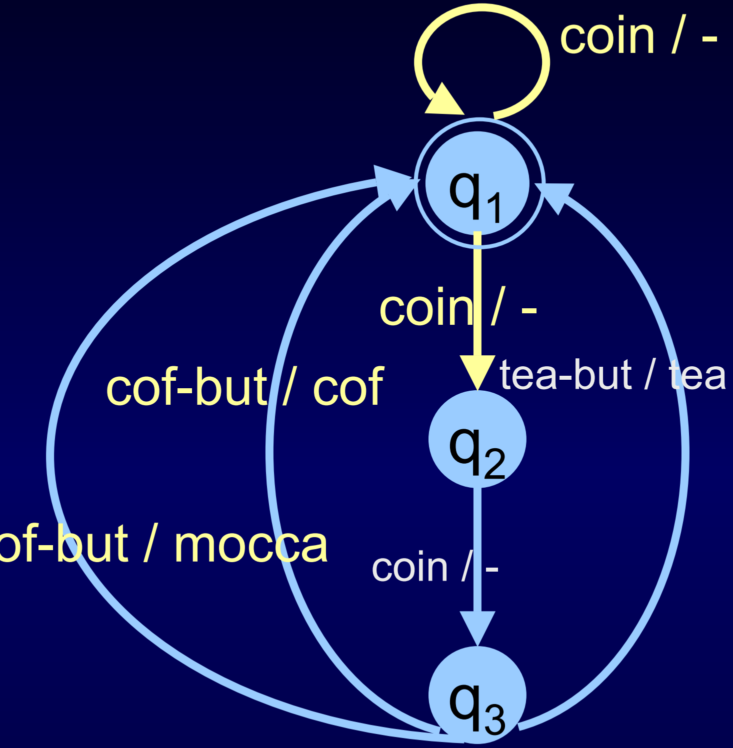
Spontaneous Transitions



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1
q_3	-	-	q_4
q_4	fix	-	q_1

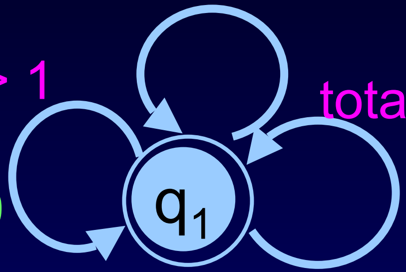
alias internal transitions alias unobservable transisions

Non-deterministic FSM



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_1	coin	-	q_1
q_2	coin	-	q_3
q_3	tea-but	tea	q_1
q_3	cof-but	cof	q_1
q_3	cof-but	mocca	q_1

Extended FSM



coin / -
total++

total>=2 and cap > 1
cof-but / cof
capacity--, total:=0

total>=1 and capacity > 1
tea-but / tea
capacity--, total:=0

- EFSM = FSMs + variables + enabling conditions + assignment
- Easier way of expressing an FSM
- Can be translated into FSM if variables have bounded domain
- State: control location+variable states: (q,total,capacity)

$$(q_1,0,10) \xrightarrow{\text{coin / -}} (q_1,1,10) \xrightarrow{\text{coin / -}} (q_1,2,10) \xrightarrow{\text{cof-but / cof}} (q_1,0,9)$$

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 M0 and M1 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*!=*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)

Conformance Testing

Reference FSM M_S

Tester

test sequence

SUT FSM M_I

Given a specification FSM M_S

an (unknown, black box) implementation FSM M_I

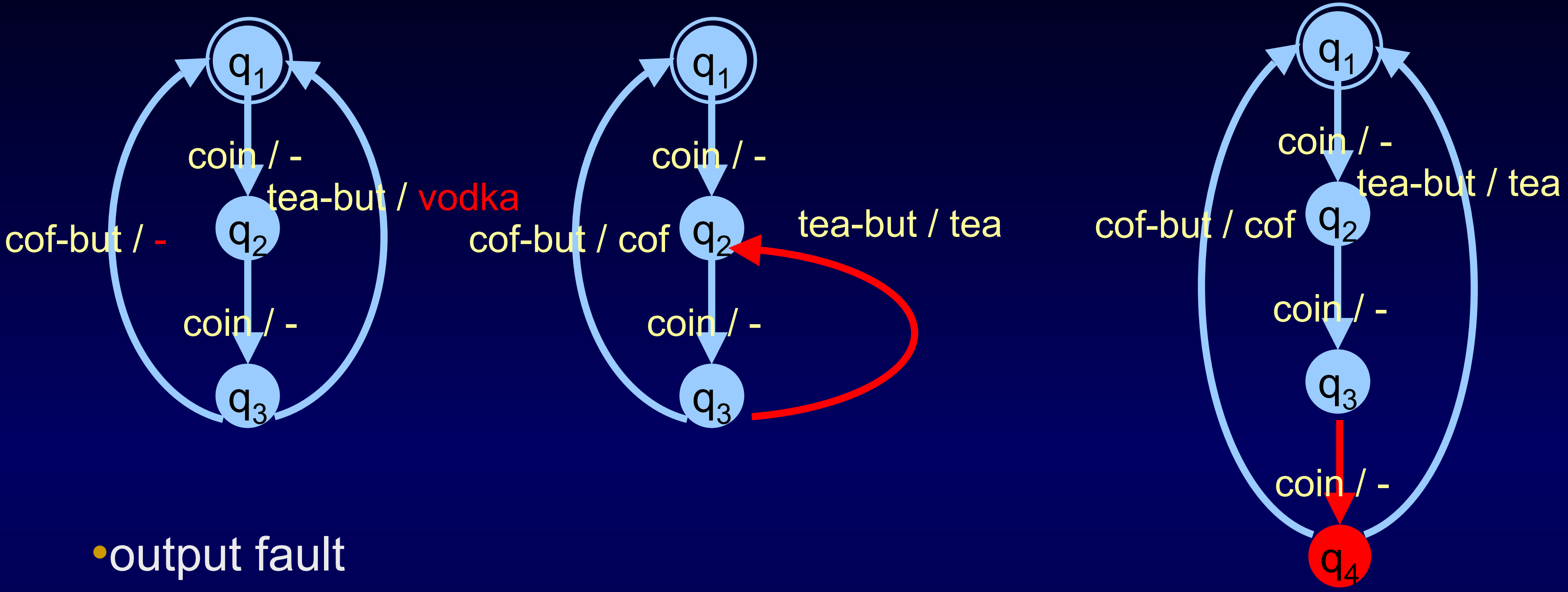
determine whether M_I conforms to M_S.

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	finite set of states
I	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

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

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

State Machine : FSM Model

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

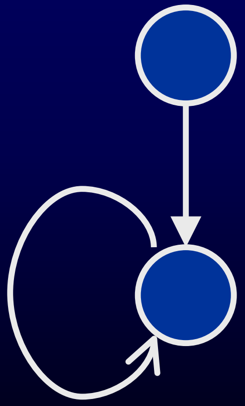
$$M = (S, I, O, \delta, \lambda)$$

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 : $\delta : S \times I^* \rightarrow S$

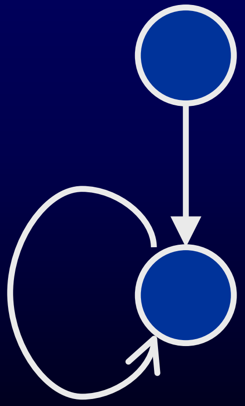
$\lambda : S \times I^* \rightarrow O^*$

Restrictions



Restrictions

FSM restrictions:

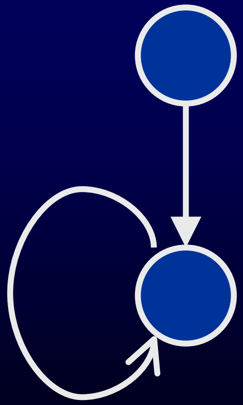


Restrictions

FSM restrictions:

- *deterministic*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions*



Restrictions

FSM restrictions:

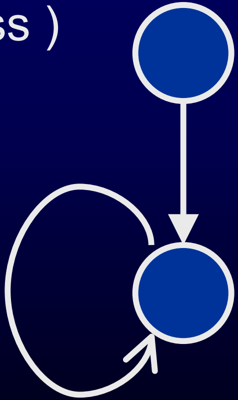
- *deterministic*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions*

- *completely specified*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *complete* functions

  (empty output is allowed; sometimes implicit completeness)



Restrictions

FSM restrictions:

- *deterministic*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions*

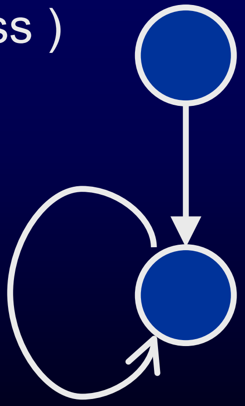
- *completely specified*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ 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*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions*

- *completely specified*

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *complete* functions

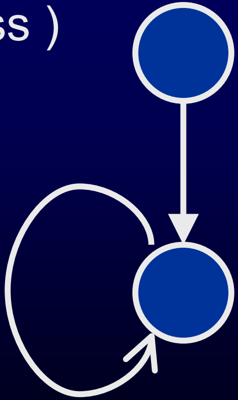
  (empty output is allowed; sometimes implicit completeness)

- *strongly connected*

  from any state any other state can be reached

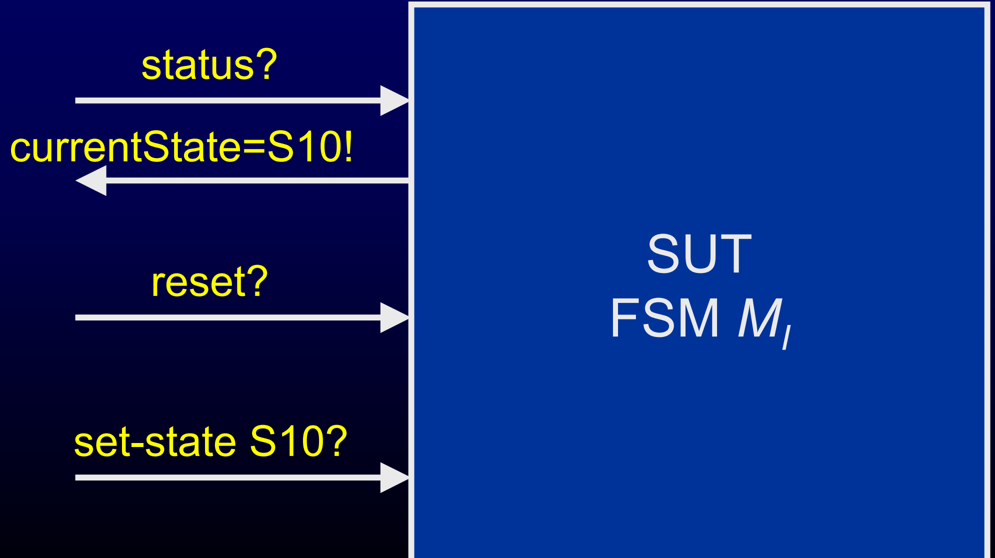
- *reduced*

  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



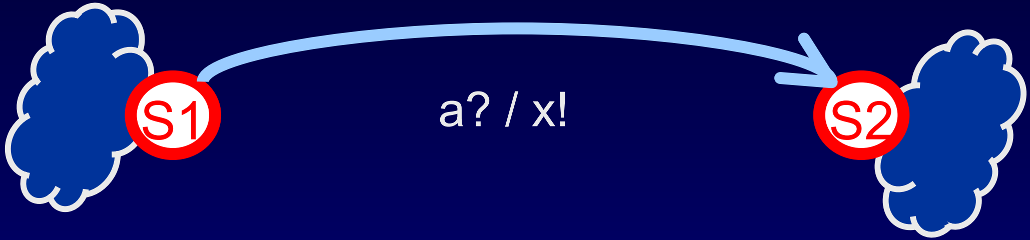
FSM Transition Testing

FSM Transition Testing

- Make a test case for every transition in spec separately:

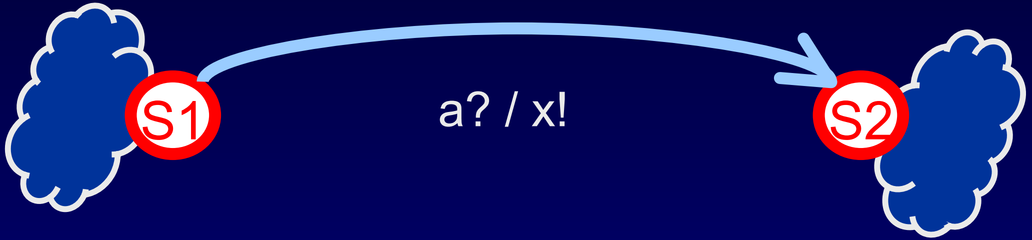
FSM Transition Testing

- Make a test case for every transition in spec separately:



FSM Transition Testing

- Make a test case for every transition in spec separately:

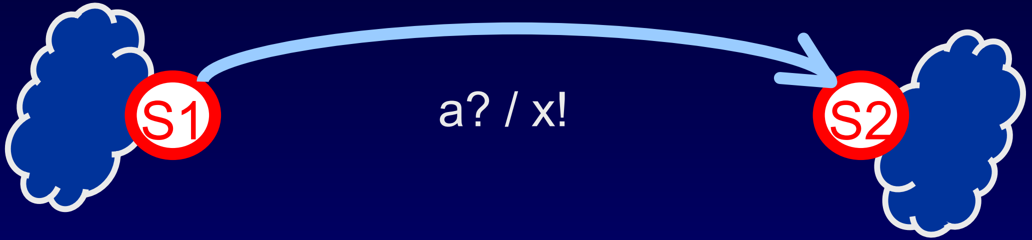


- Test transition :

1. Go to state S1
2. Apply input a?
3. Check output x!
4. Verify state S2 (optionally)

FSM Transition Testing

- Make a test case for every transition in spec separately:

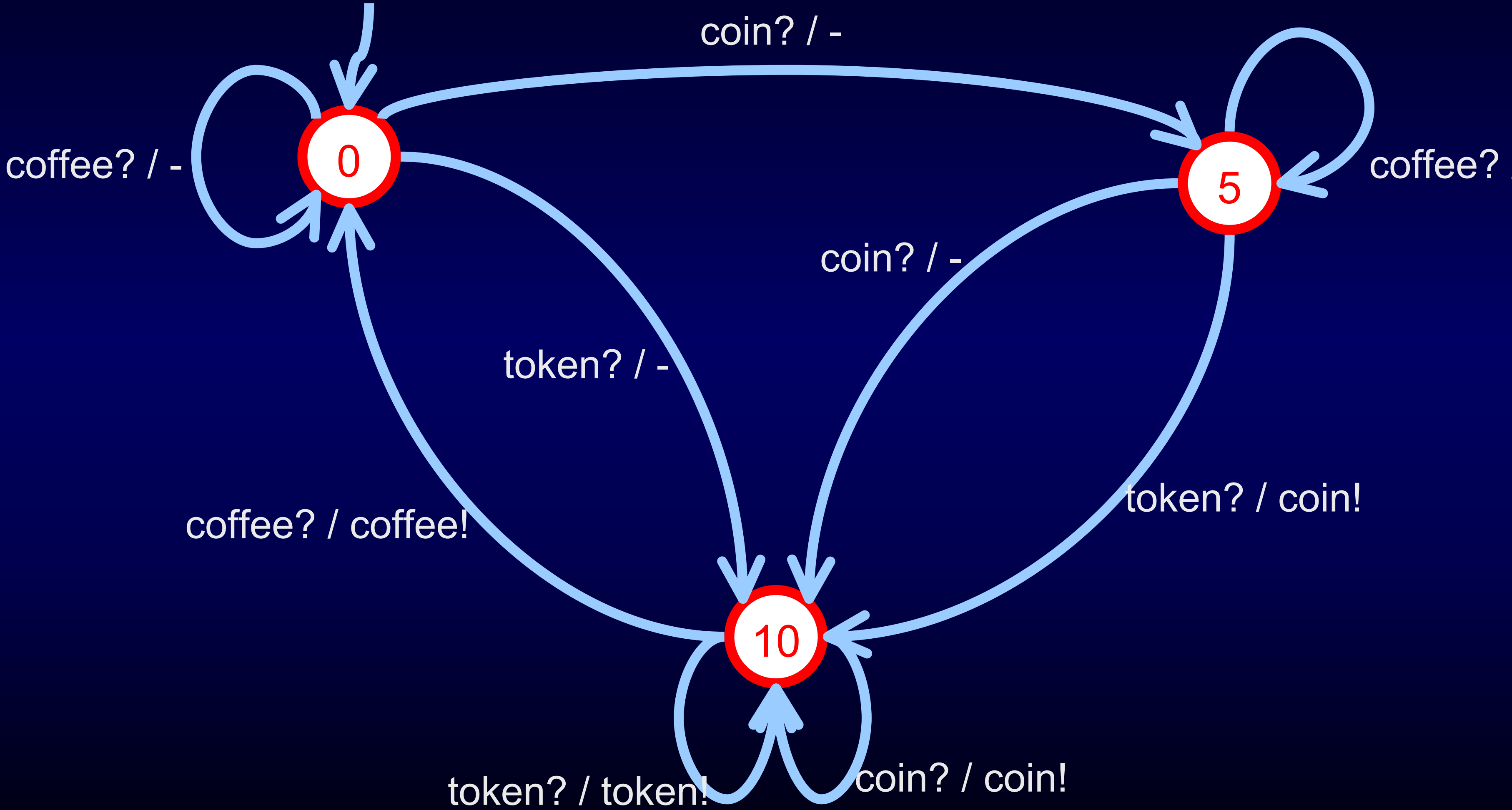


- Test transition :

1. Go to state S1
2. Apply input a?
3. Check output x!
4. Verify state S2 (optionally)

- *Test purpose: "Test whether the system, when in state S1, produces output x! on input a? and goes to state S2"*

Coffee Machine FSM Model



coffee? / -

0

coin? / -

5

coffee? /

coin? / -

token? / -

token? / coin!

coffee? / coffee!

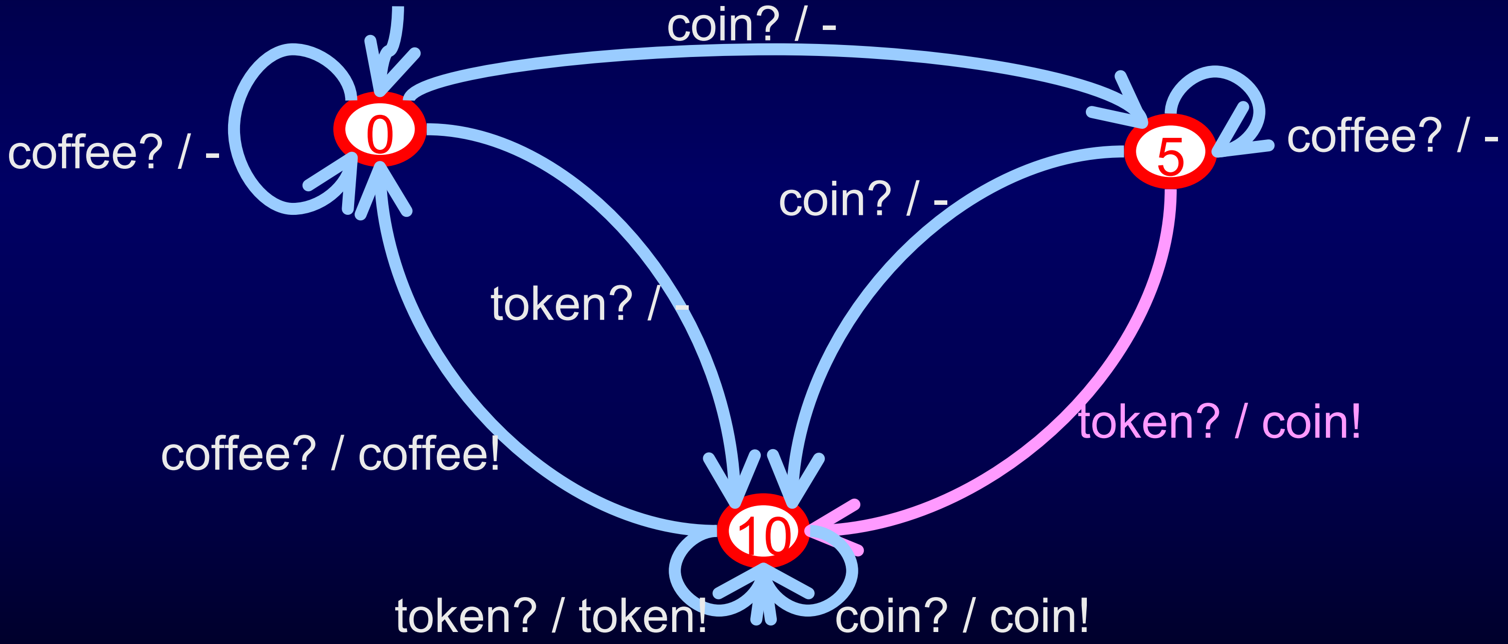
10

token? / token!

coin? / coin!

Transition Testing –1

Transition Testing –1



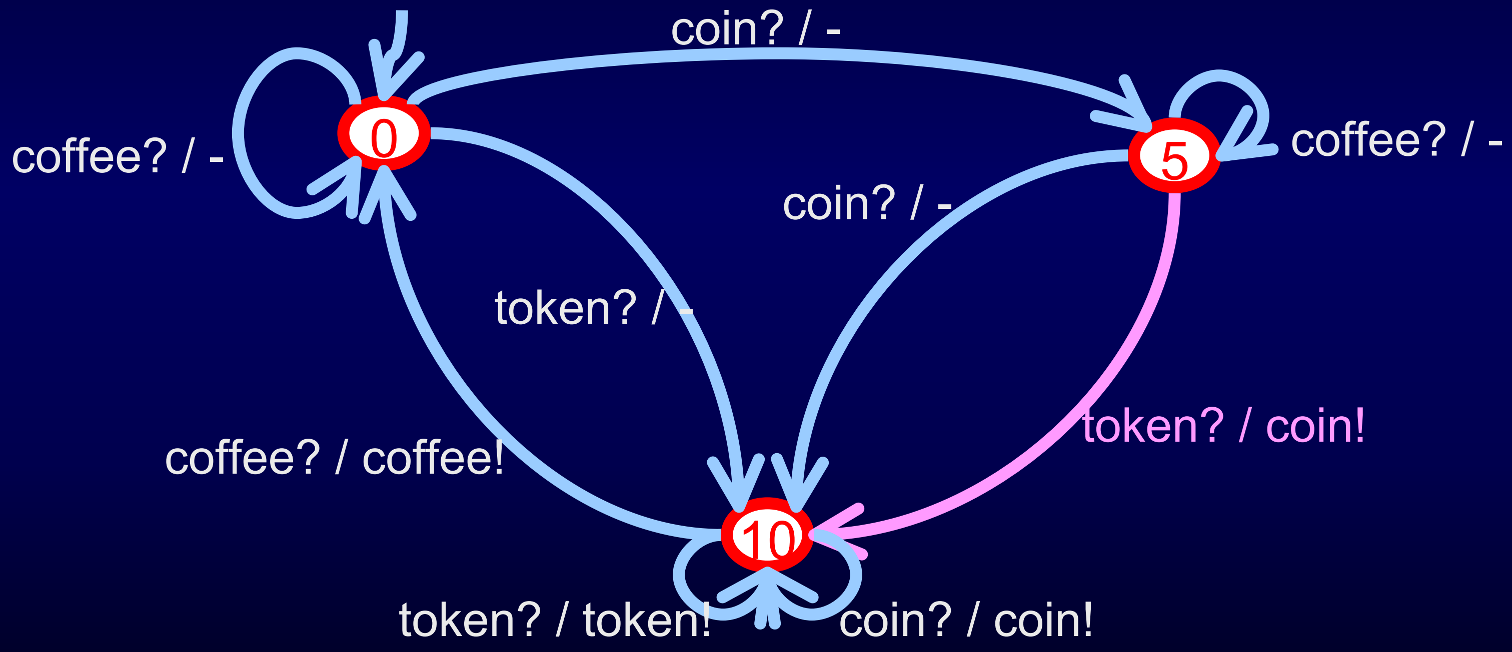
Transition Testing –1

- To test token? / coin! :

  go to state 5 : reset . set-state 5

  give input token? check output coin!

  verify state: send status? check status=10



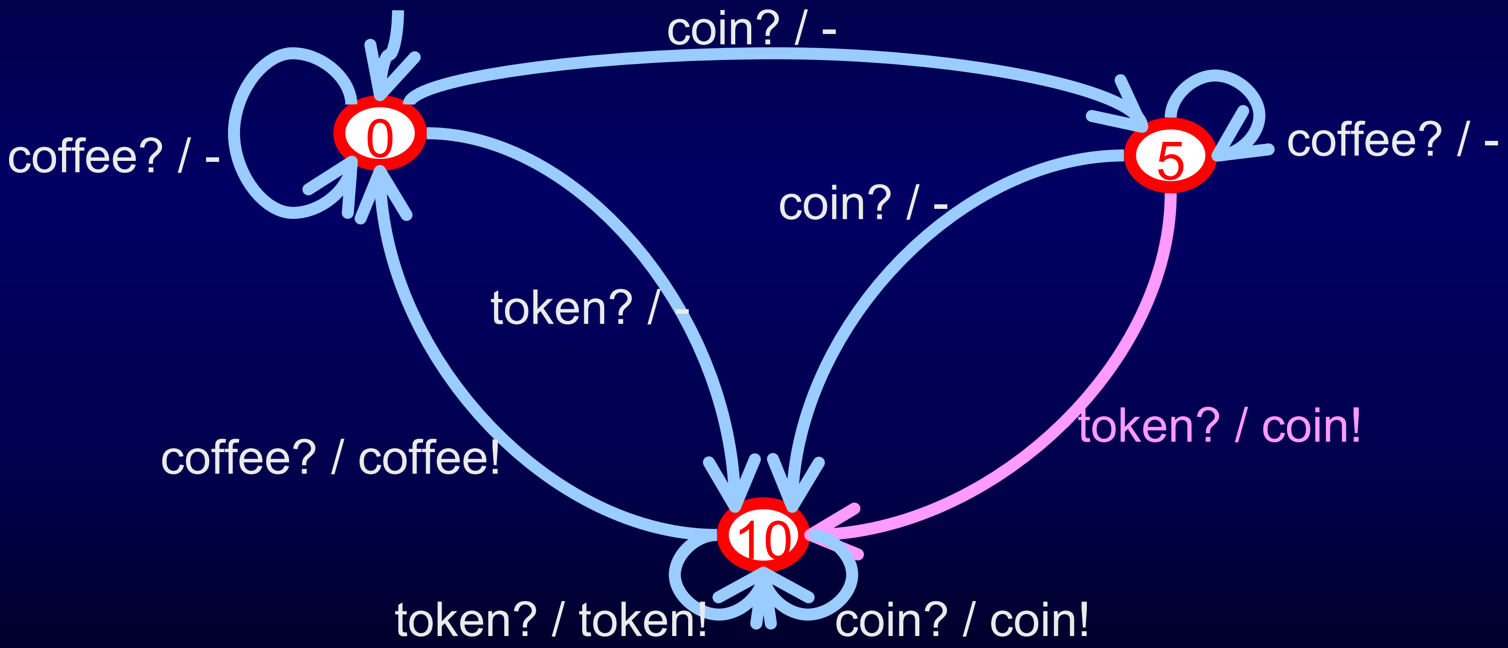
Transition Testing –1

- To test token? / coin! :

  go to state 5 : reset . set-state 5

  give input token? check output coin!

  verify state: send status? check status=10



Test case : set-state 5/ * - token? / coin! - status? / 10!

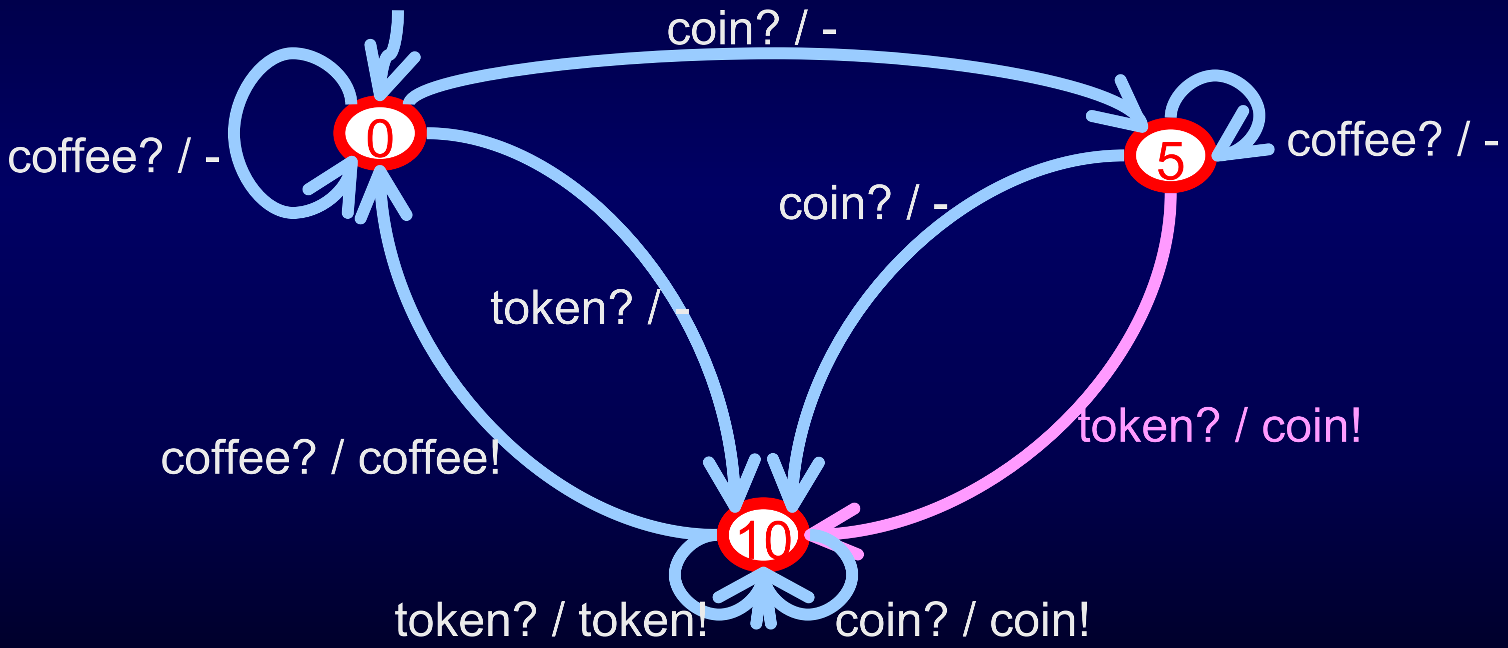
Transition Testing –1

- To test token? / coin! :

  go to state 5 : reset . set-state 5

  give input token? check output coin!

  verify state: send status? check status=10



Test case : set-state 5/ * - token? / coin! - status? / 10!

a test case per state per input event: total length 4 * |S| * | I |

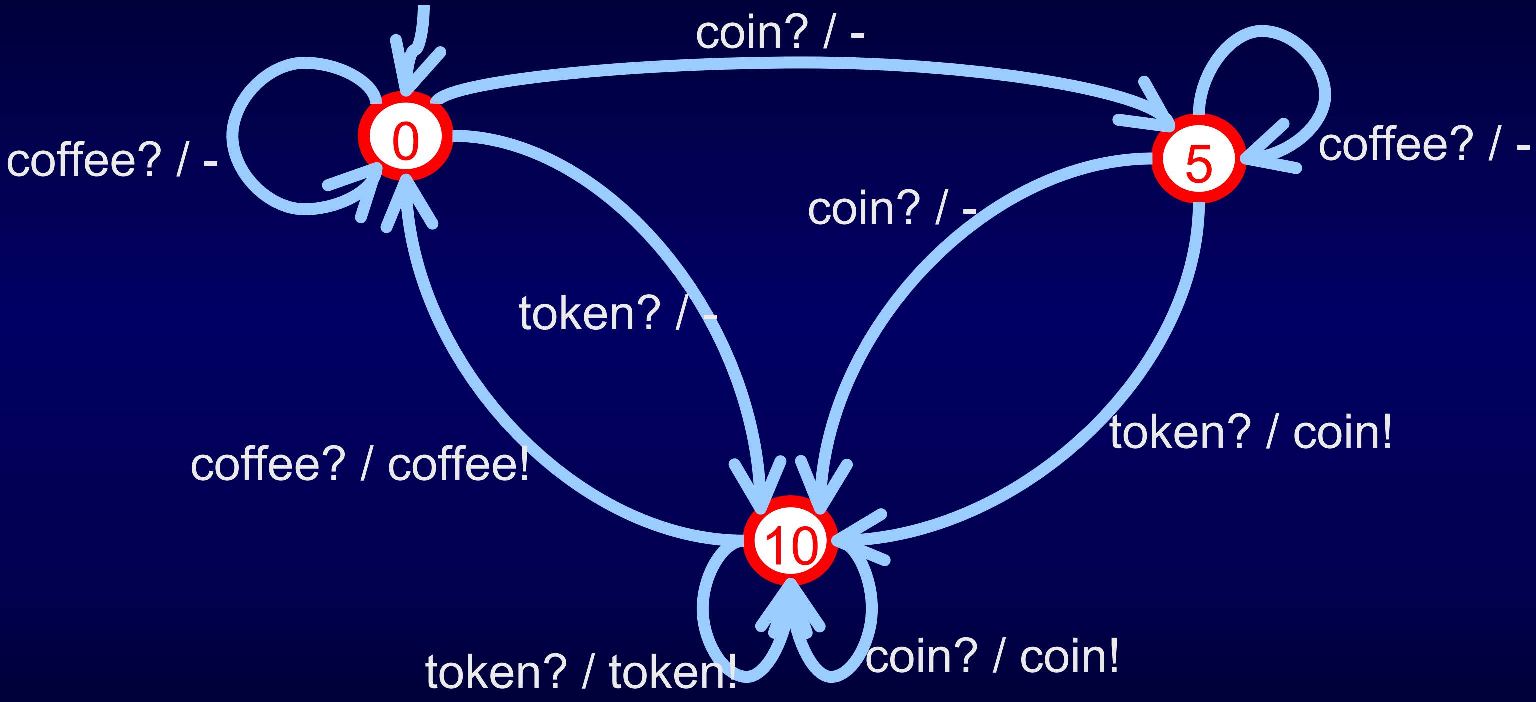
FSM Transition Tour

FSM Transition Tour

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

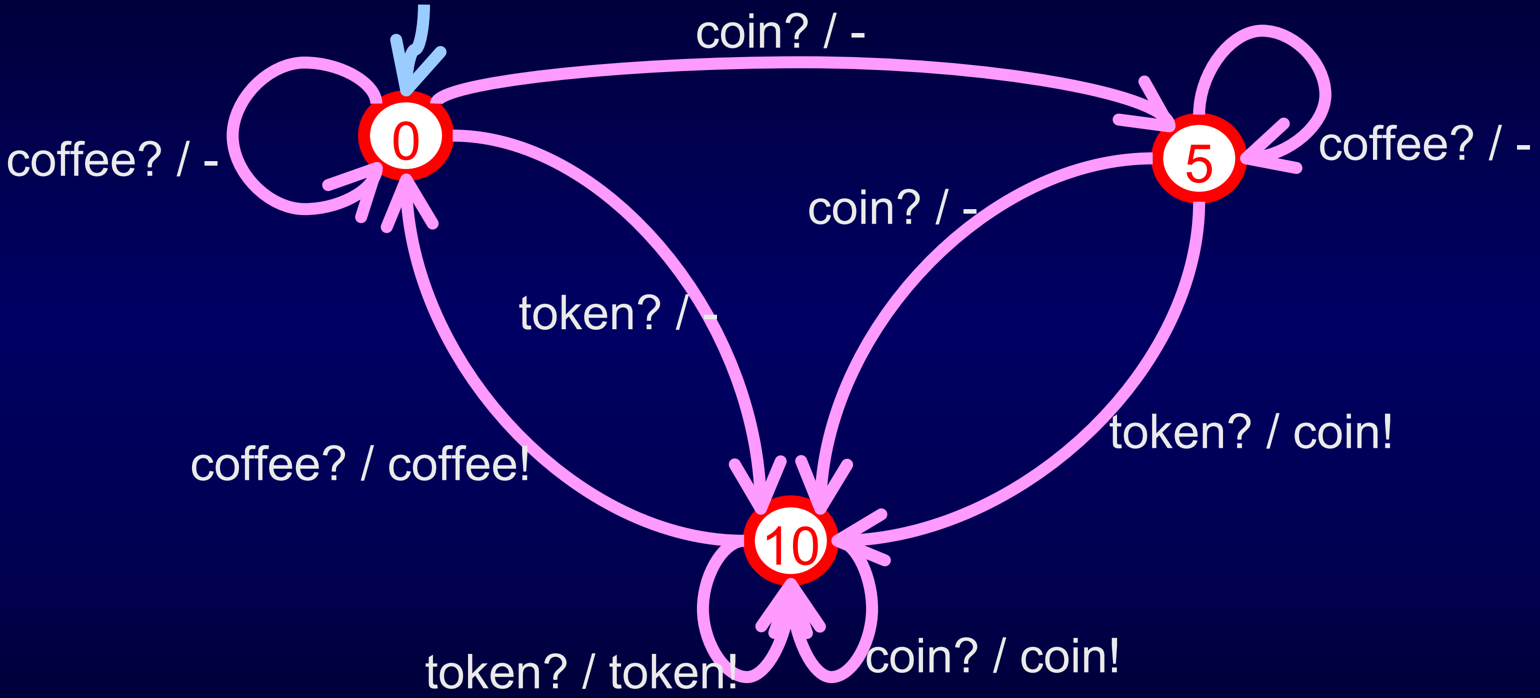
FSM Transition Tour

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



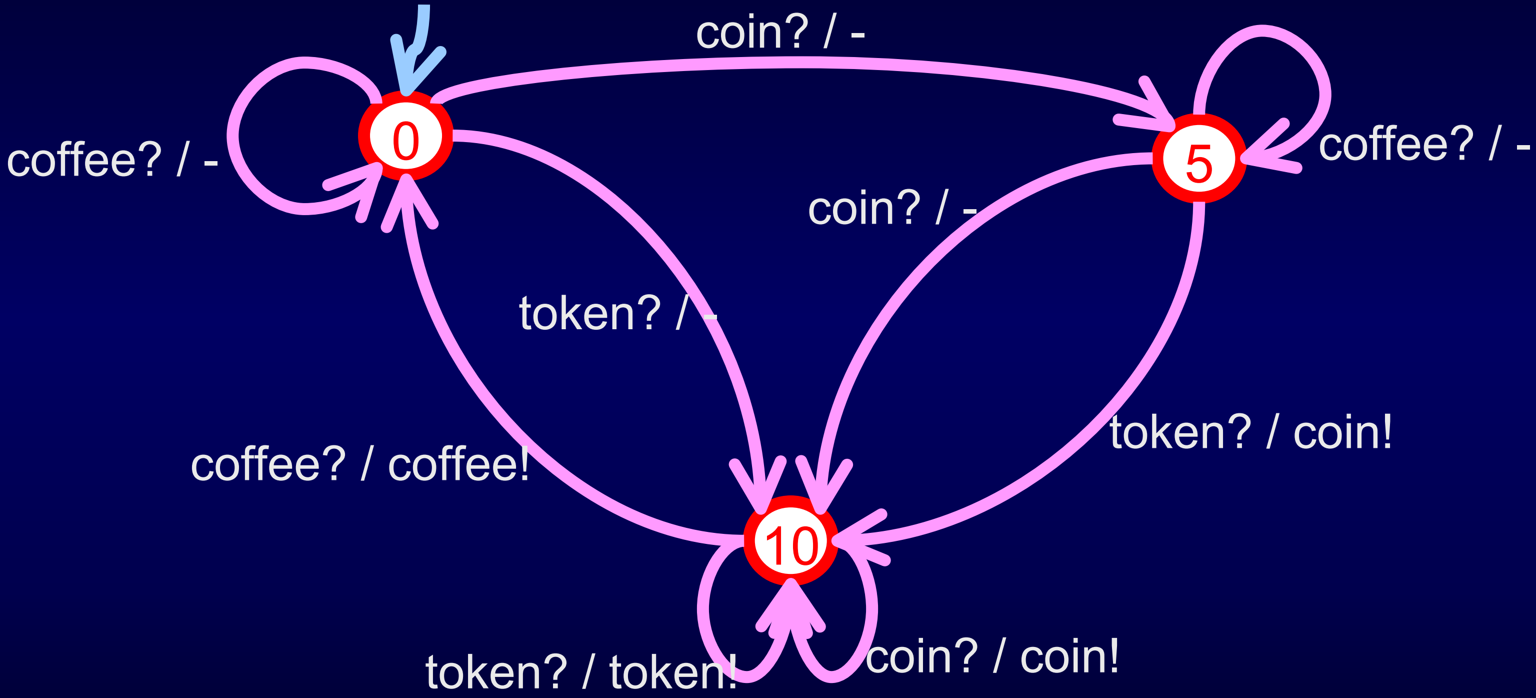
FSM Transition Tour

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



FSM Transition Tour

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

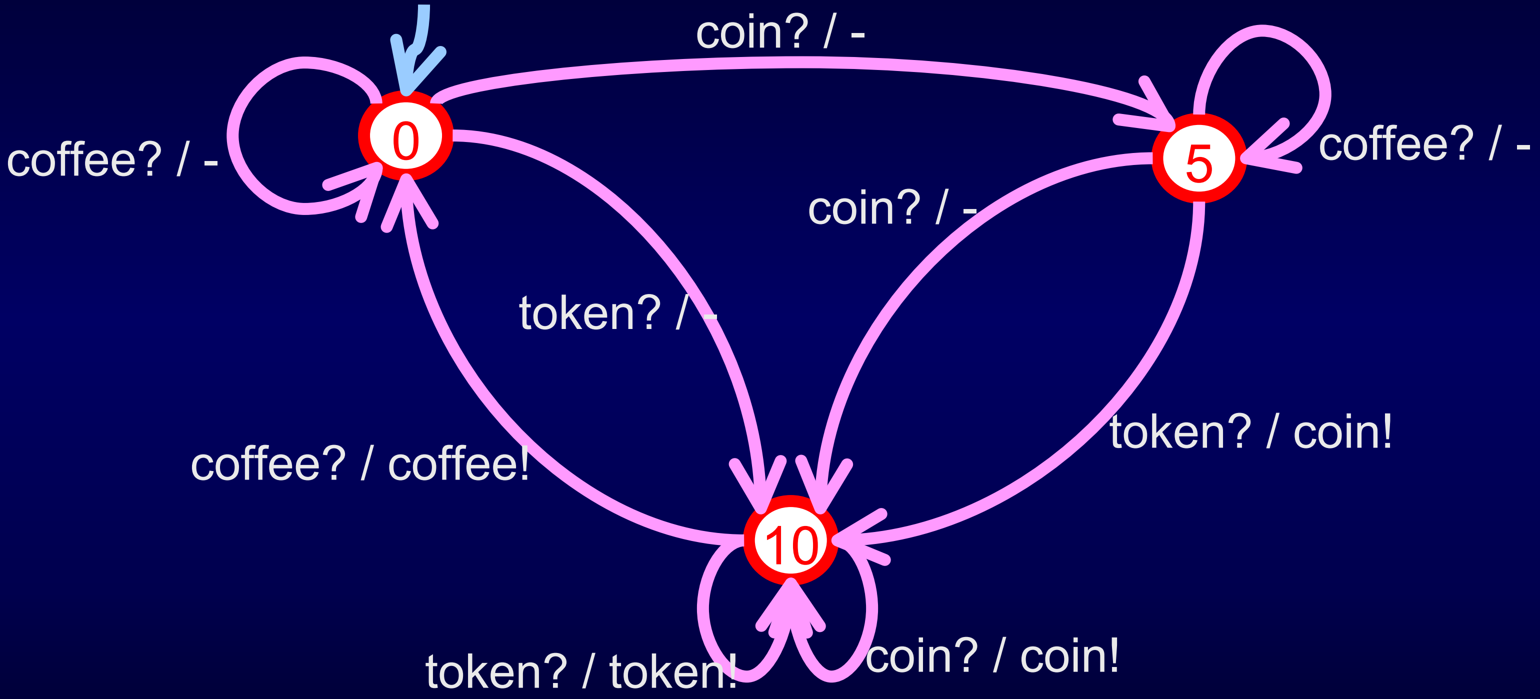


Test input sequence :

reset? coffee? coin? coffee? coin? coin? token? coffee? token? coffee? coin? token? coff

FSM Transition Tour

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



Test input sequence :

reset? coffee? coin? coffee? coin? coin? token? coffee? token? coffee? coin? token? coff

+ check expected outputs and target state by status message

Transition Testing -1

Transition Testing -1

- Go to state S5 :

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 S5 :
- No Set-state property???
  - use *reset property* if available
  - go from S0 to S5
    (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 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
  - (but synchronizing sequence may not exist)

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
  - (but synchronizing sequence may not exist)
  - A preset *homing sequence* is an input sequence x whose output on x (applied in any state) uniquely identifies the reached state after x!

Transition Testing -1

token? coffee?

To test token? / coin! : go to state 5 by : token? coffee? coin?

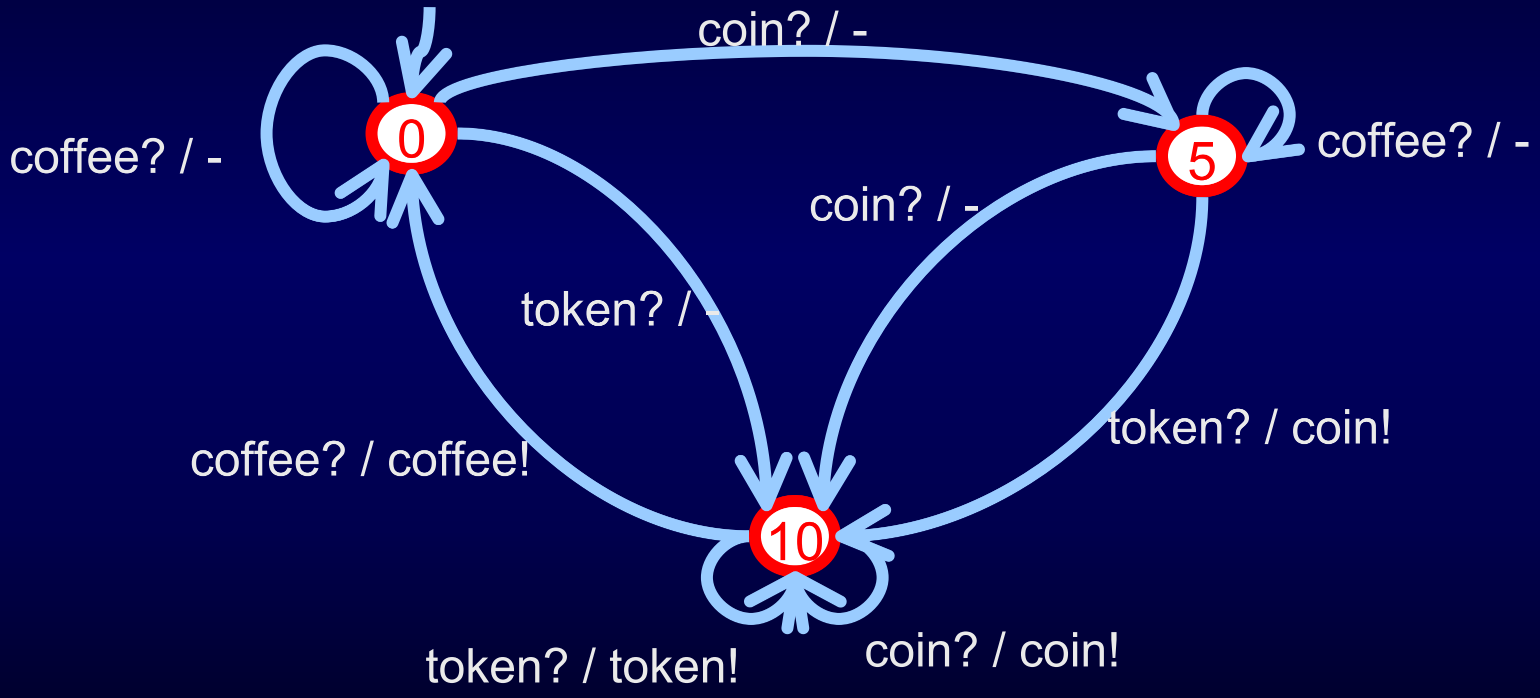
Transition Testing -1

synchronizing sequence : token? coffee?

To test token? / coin! : go to state 5 by : token? coffee? coin?

Transition Testing -1

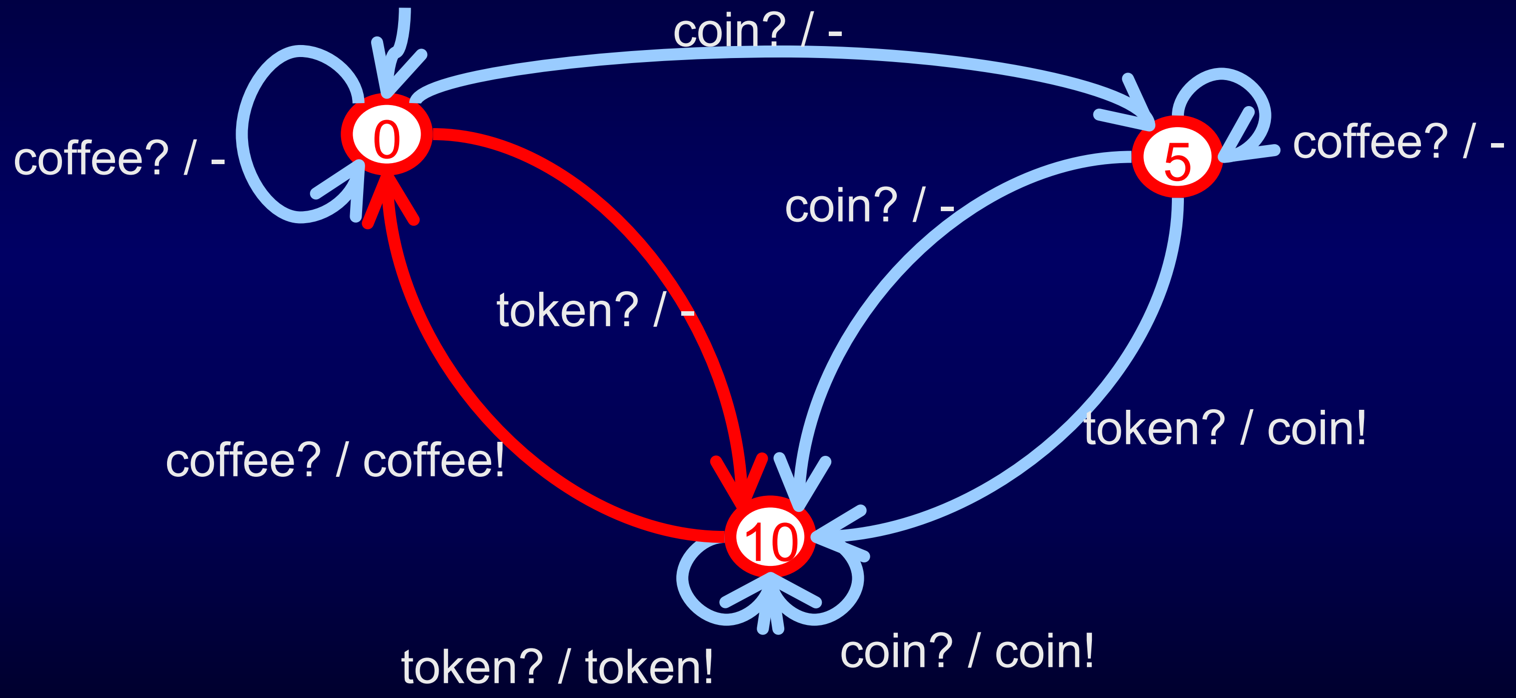
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

Transition Testing -1

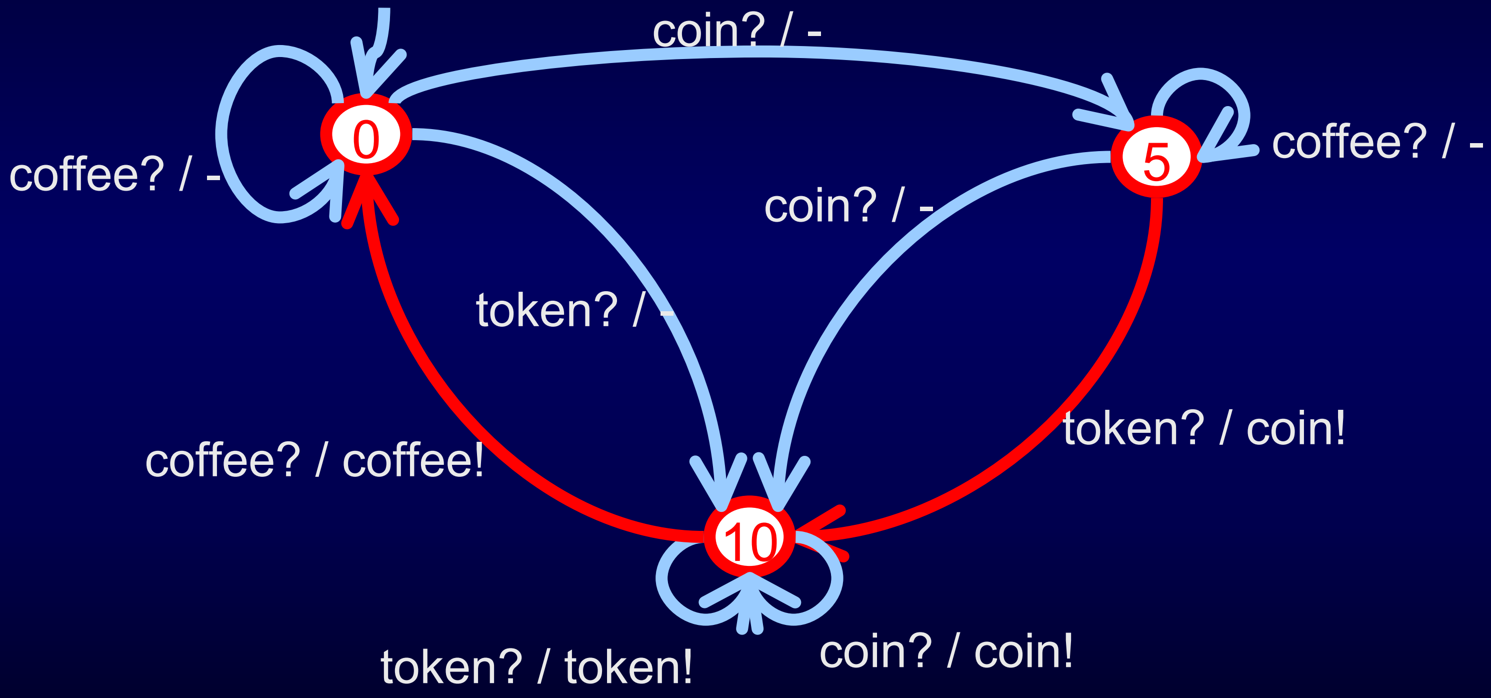
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

Transition Testing -1

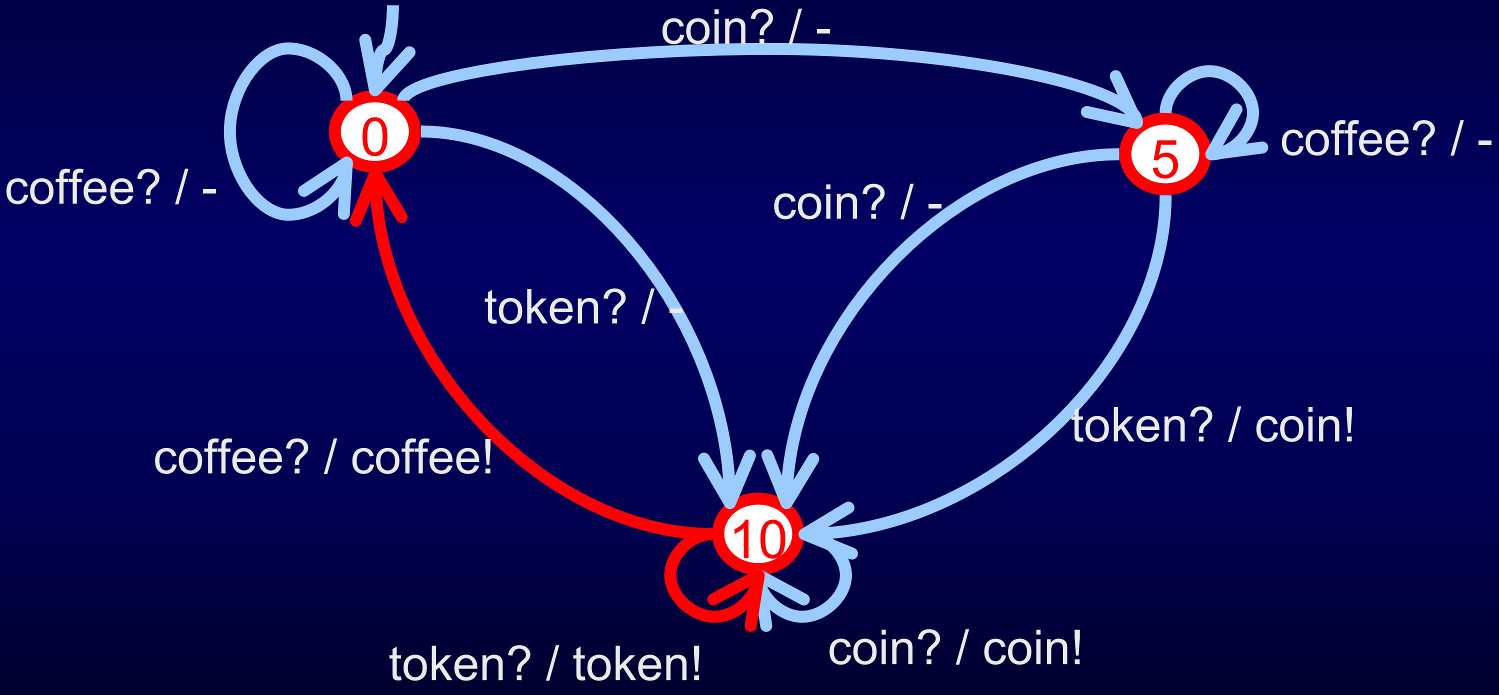
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

Transition Testing -1

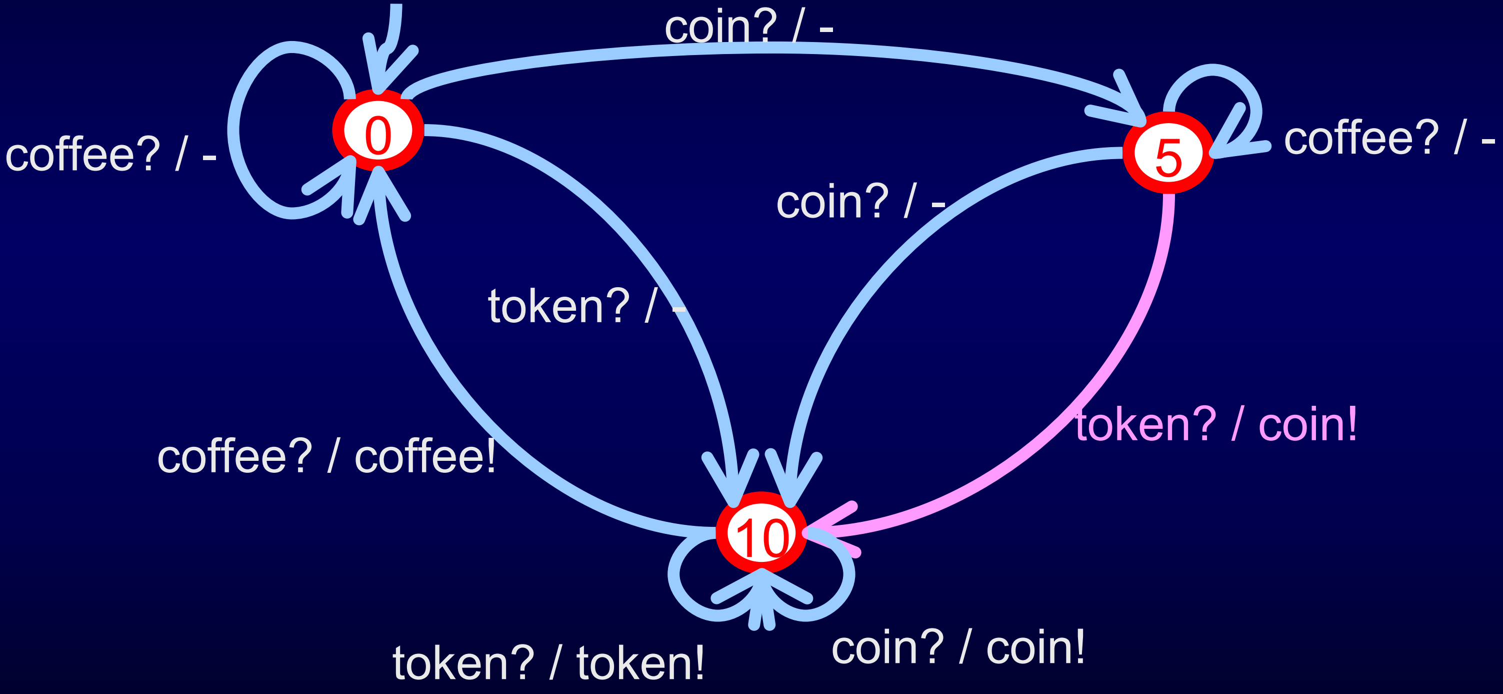
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

Transition Testing -1

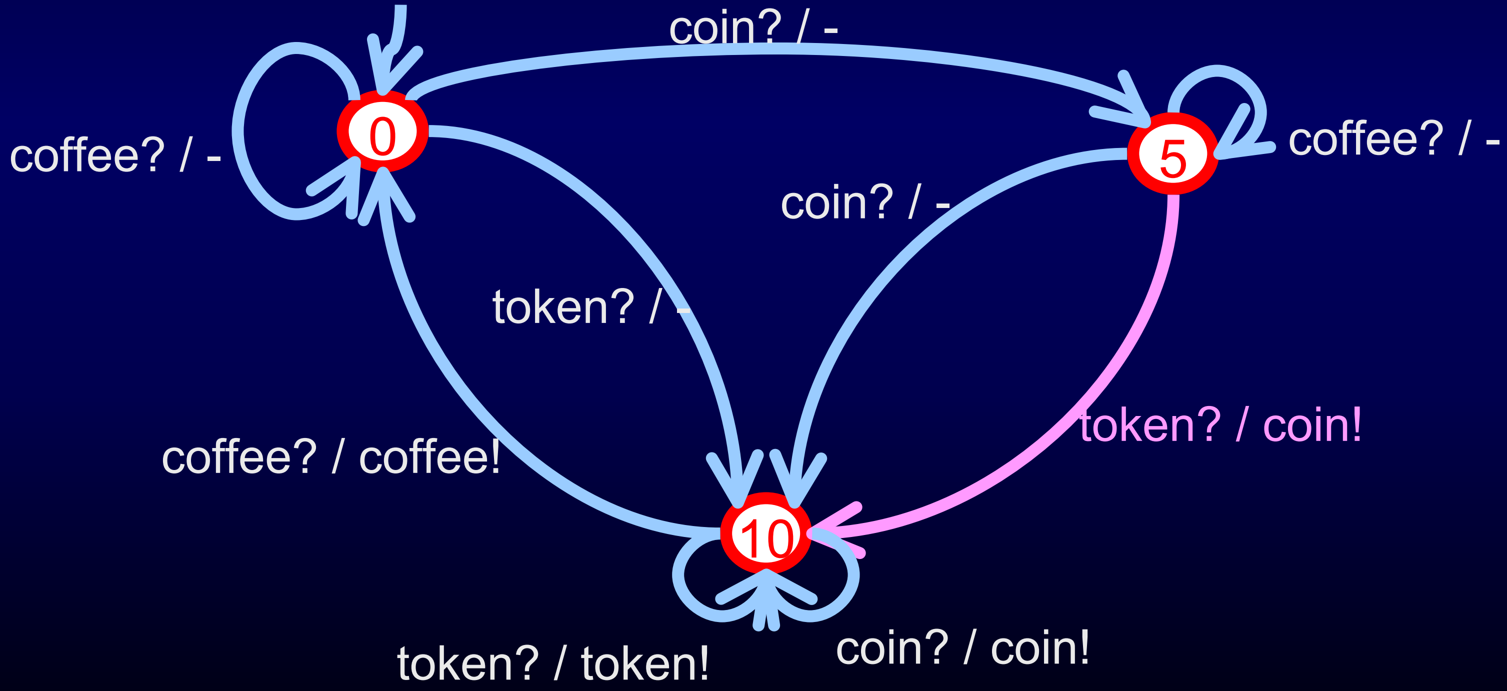
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

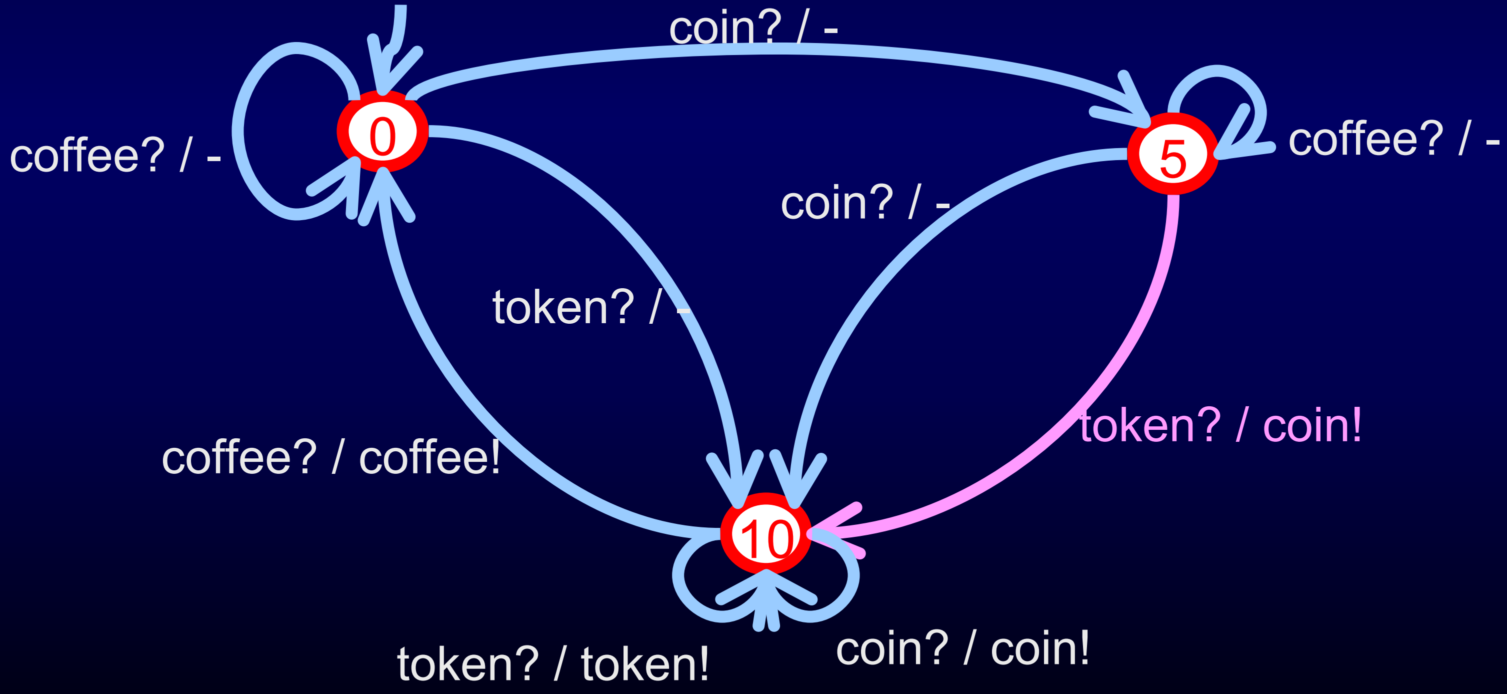
Transition Testing –2,3

Transition Testing –2,3



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

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, is P-SPACE complete

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, is P-SPACE complete
- 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, is P-SPACE complete
- 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 : 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, is P-SPACE complete
- 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

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, is P-SPACE complete
- 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
- Characterizing Sequences (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, is P-SPACE complete
- 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
- Characterizing Sequences (W - set of sequences) (identification)
  - *set of* sequences W which can distinguish any pair of states : for all pairs $t \neq s$ there is $x_{s,t} \in W$: $\lambda (s, x_{s,t}) \neq \lambda (t, x_{s,t})$

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, is P-SPACE complete
- 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
- Characterizing Sequences (W - set of sequences) (identification)
  - *set of* sequences W which can distinguish any pair of states : for all pairs $t \neq s$ there is $x_{s,t} \in W$: $\lambda (s, x_{s,t}) \neq \lambda (t, x_{s,t})$
  - W - set always exists for reduced FSM

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, is P-SPACE complete
- 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
- Characterizing Sequences (W - set of sequences) (identification)
  - *set of* sequences W which can distinguish any pair of states : for all pairs $t \neq s$ there is $x_{s,t} \in W$: $\lambda (s, x_{s,t}) \neq \lambda (t, x_{s,t})$
  - W - set always exists for reduced FSM
  - Length $O(VS^3)$

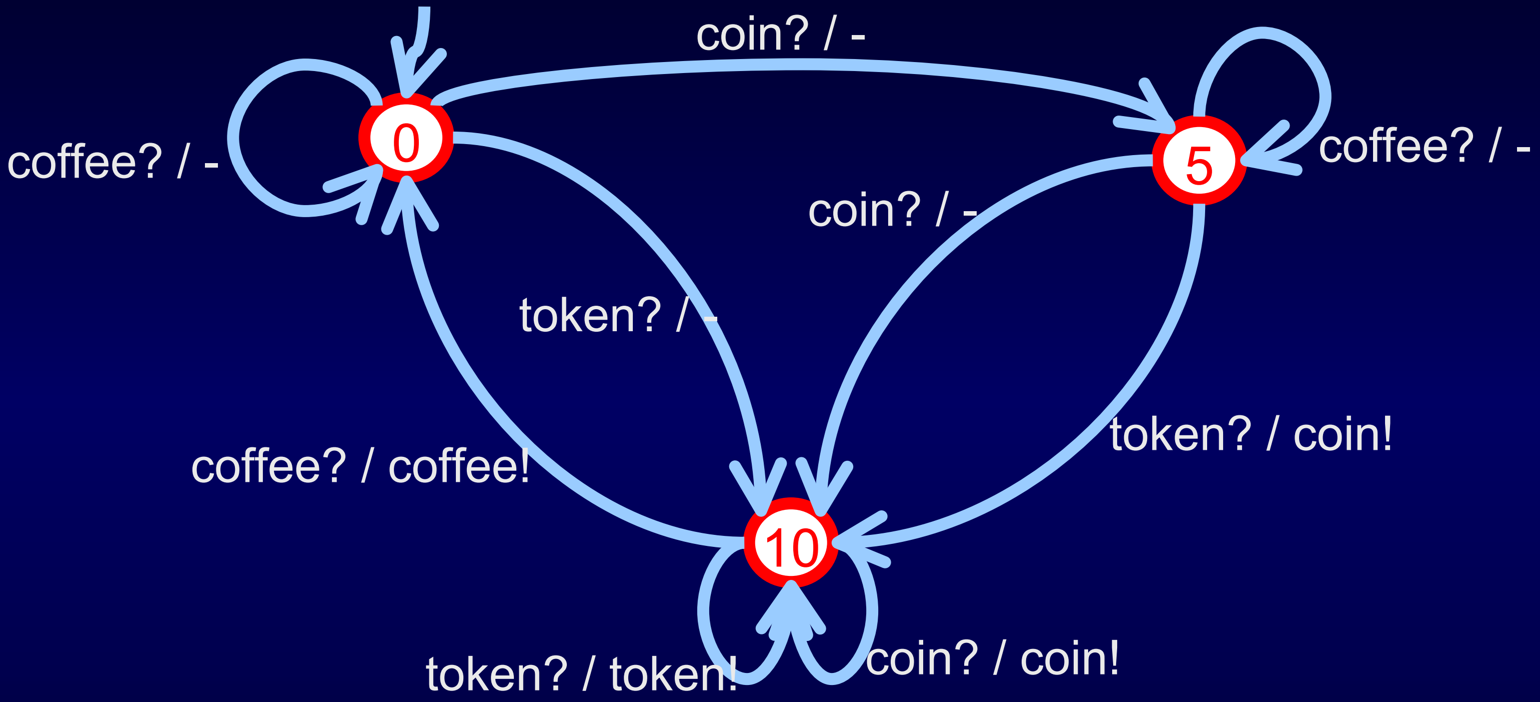
Transition Testing-4: UIO

Transition Testing-4: UIO

UIO sequences

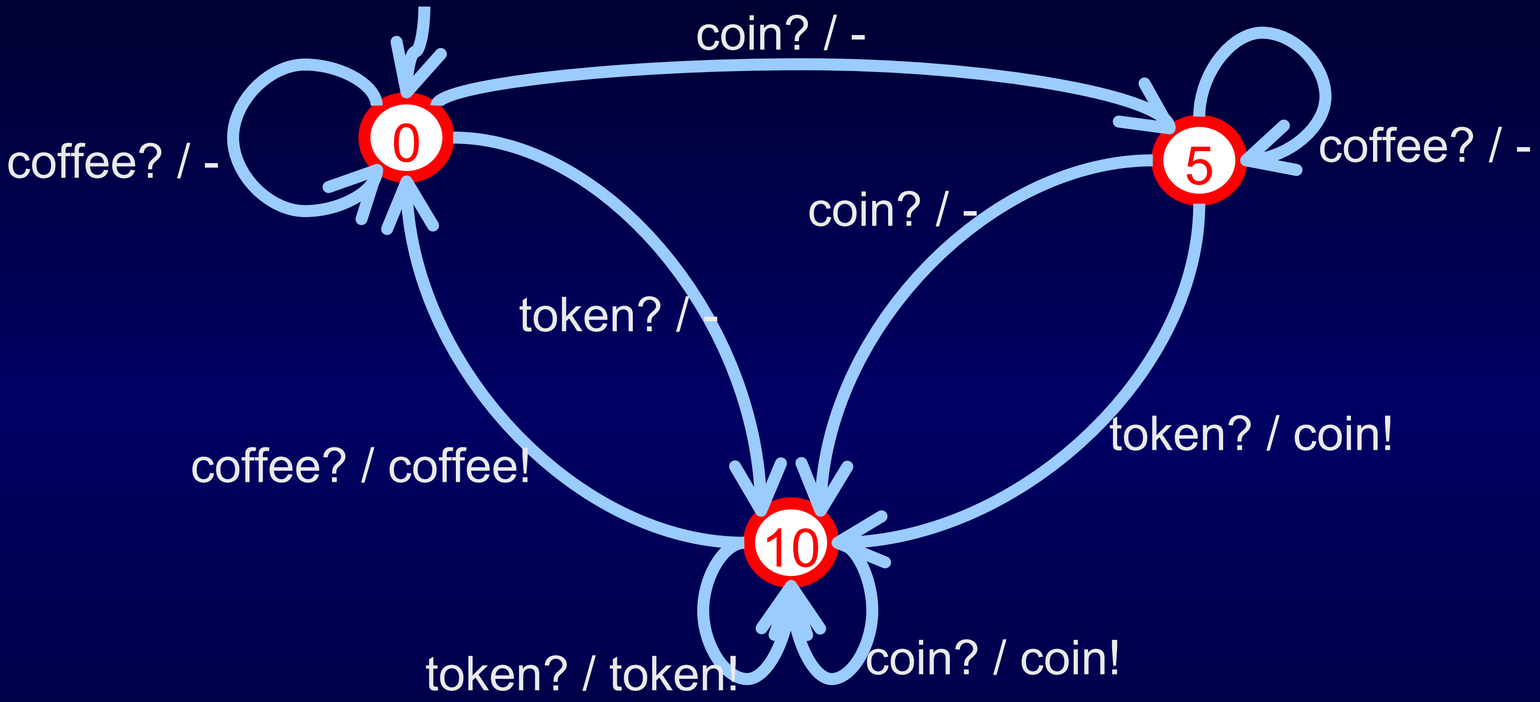
Transition Testing-4: UIO

UIO sequences



Transition Testing-4: UIO

UIO sequences



state 0 : coin? / - coffee? / -

state 5 : token? / coin!

state 10 : coffee? / coffee!

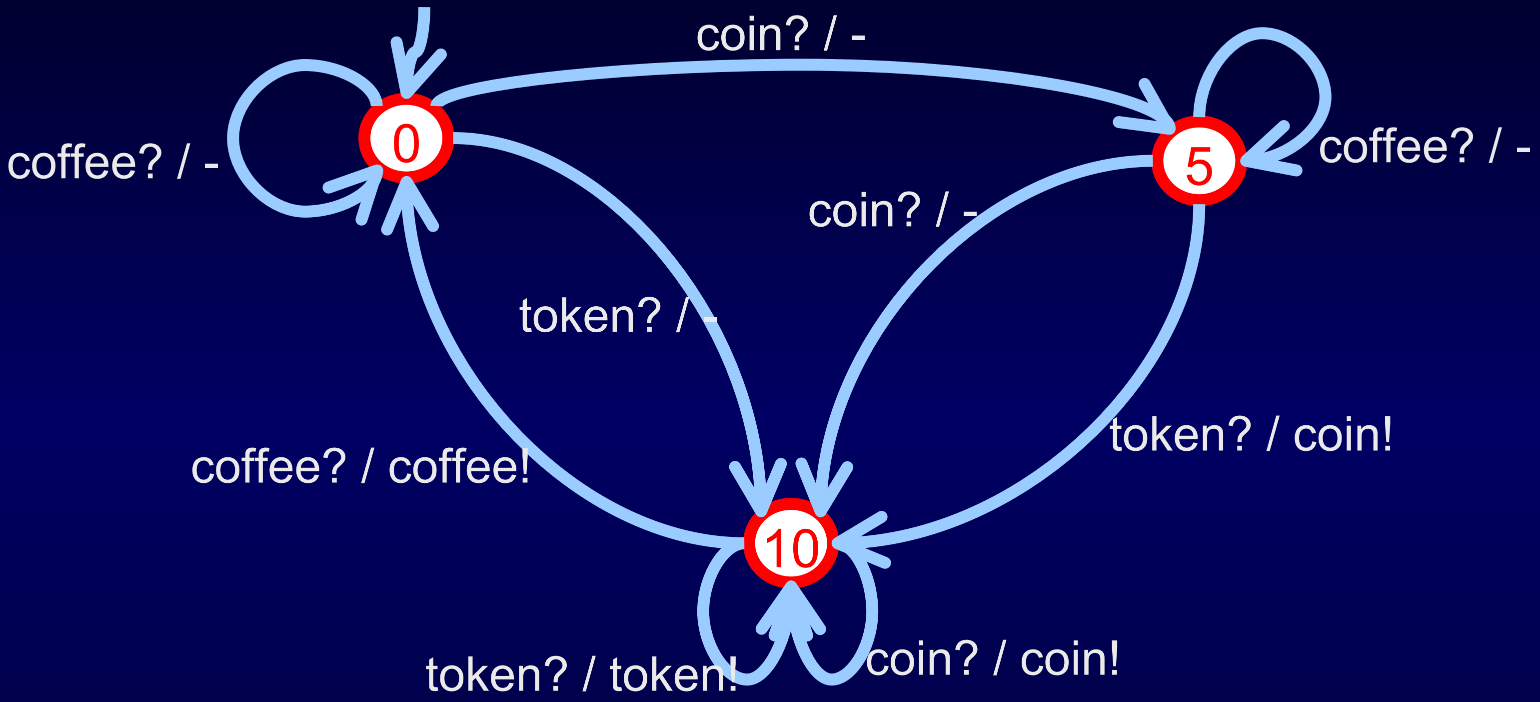
Transition Testing-4: DS

Transition Testing-4: DS

DS sequence

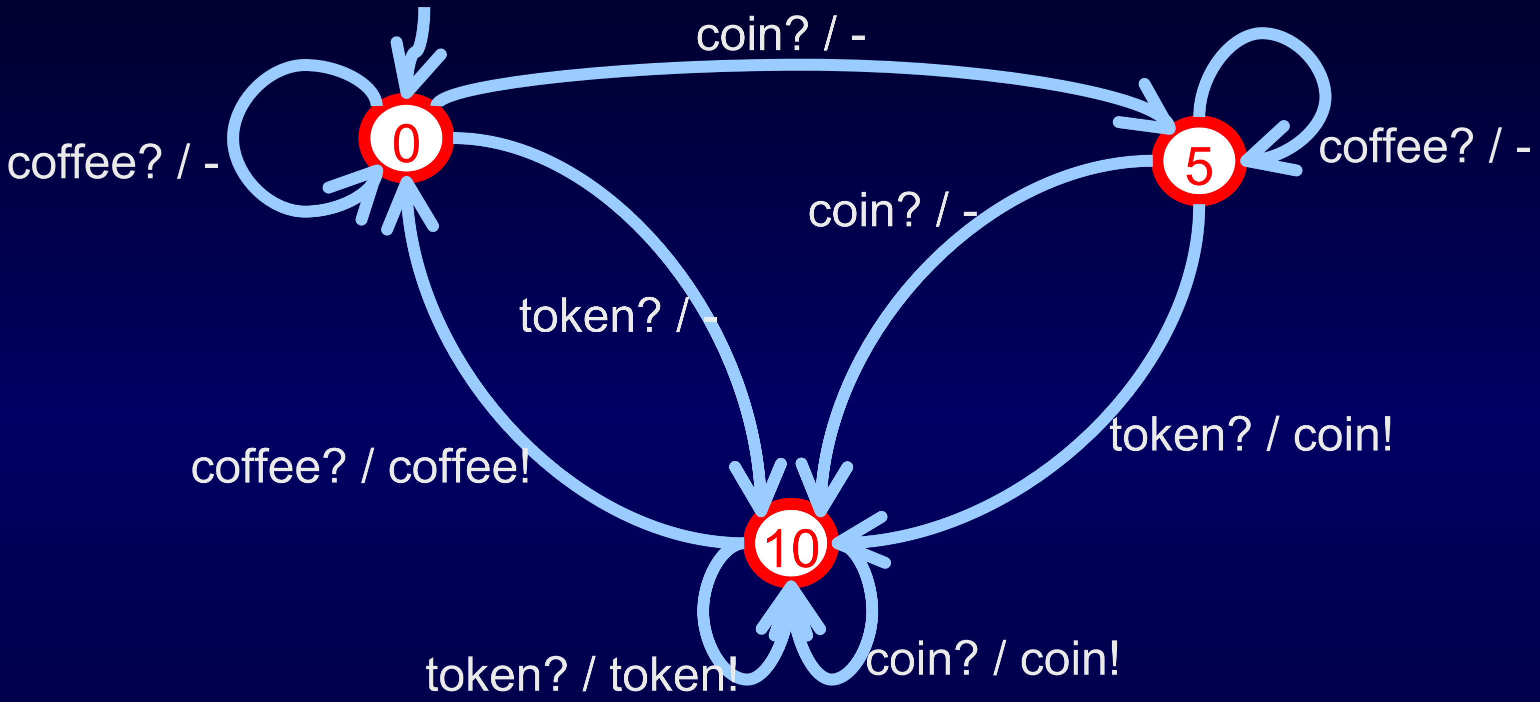
Transition Testing-4: DS

DS sequence



Transition Testing-4: DS

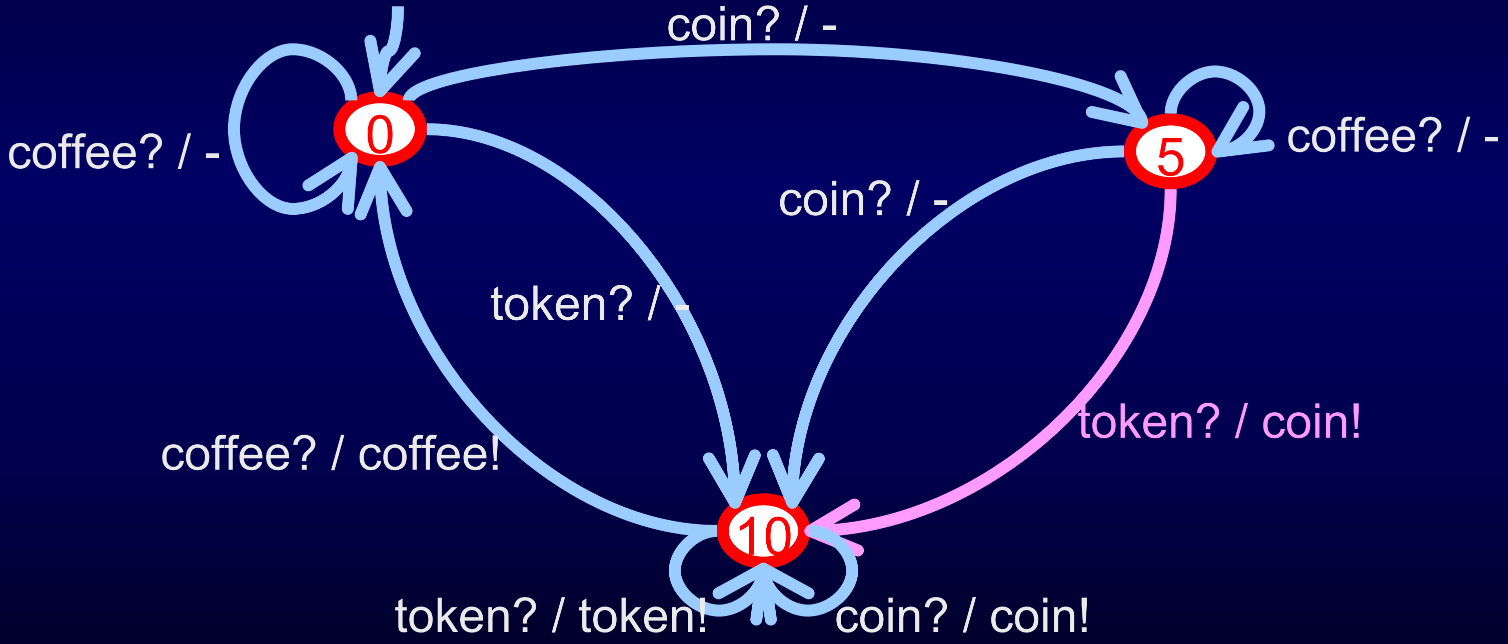
DS sequence



DS sequence : token? output state 0 : -
output state 5 : coin!
output state 10 : token!

Transition Testing –4 done

Transition Testing –4 done



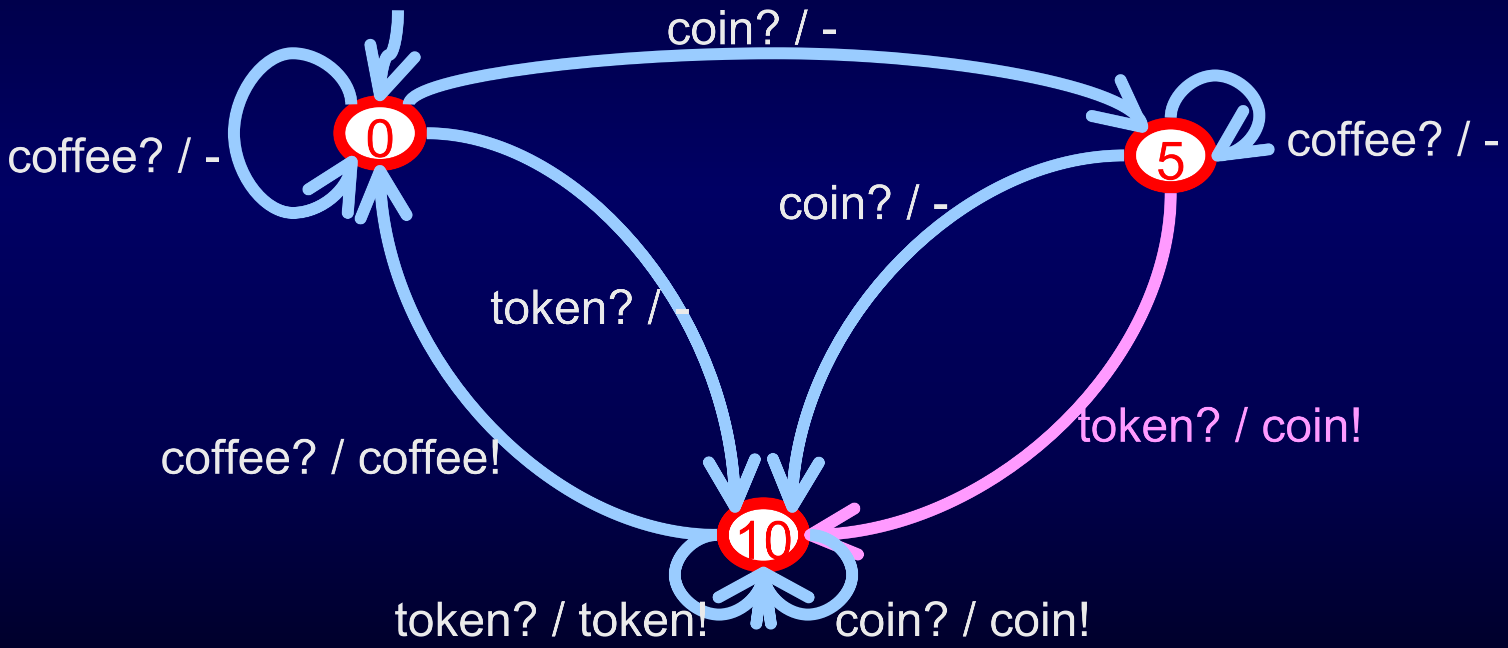
Transition Testing –4 done

- To test token? / coin! :

  go to state 5 : token? coffee? coin?

  give input token? check output coin!

  Apply UIO of state 10 : coffee? / coffee!



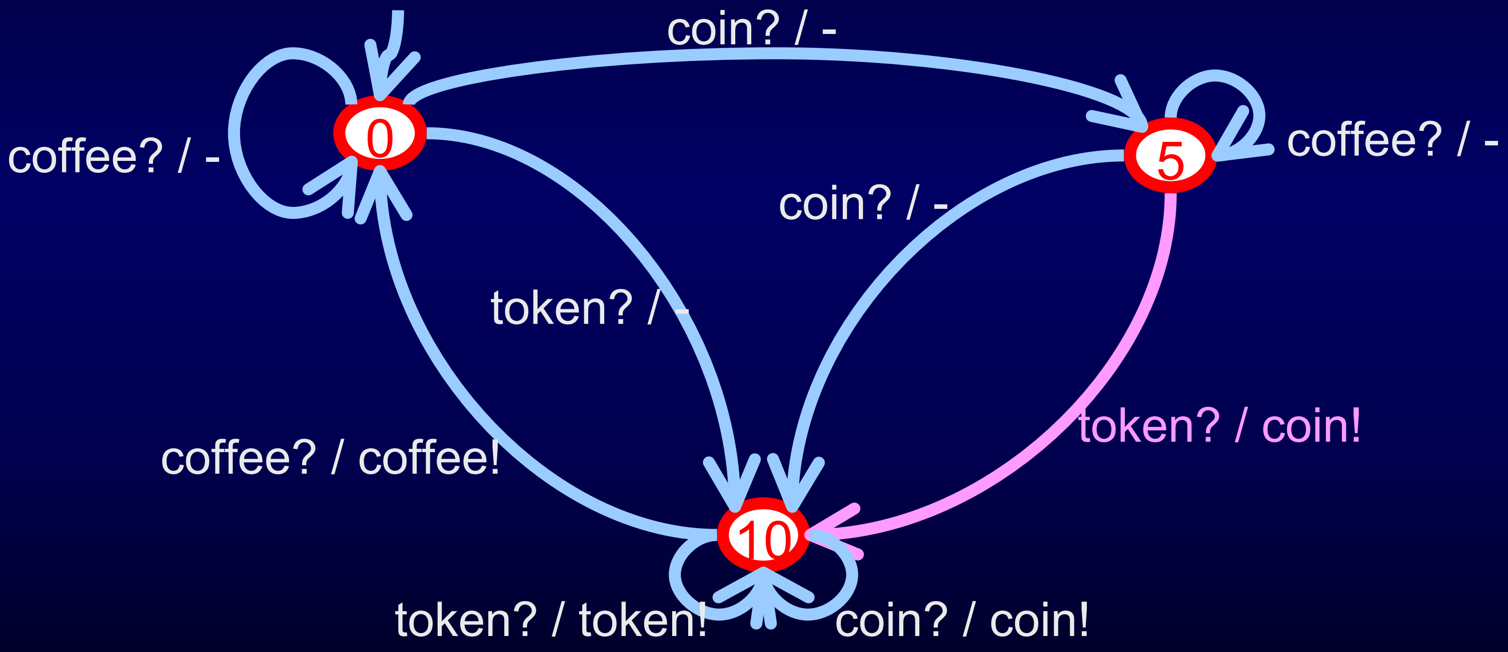
Transition Testing –4 done

- To test token? / coin! :

  go to state 5 : token? coffee? coin?

  give input token? check output coin!

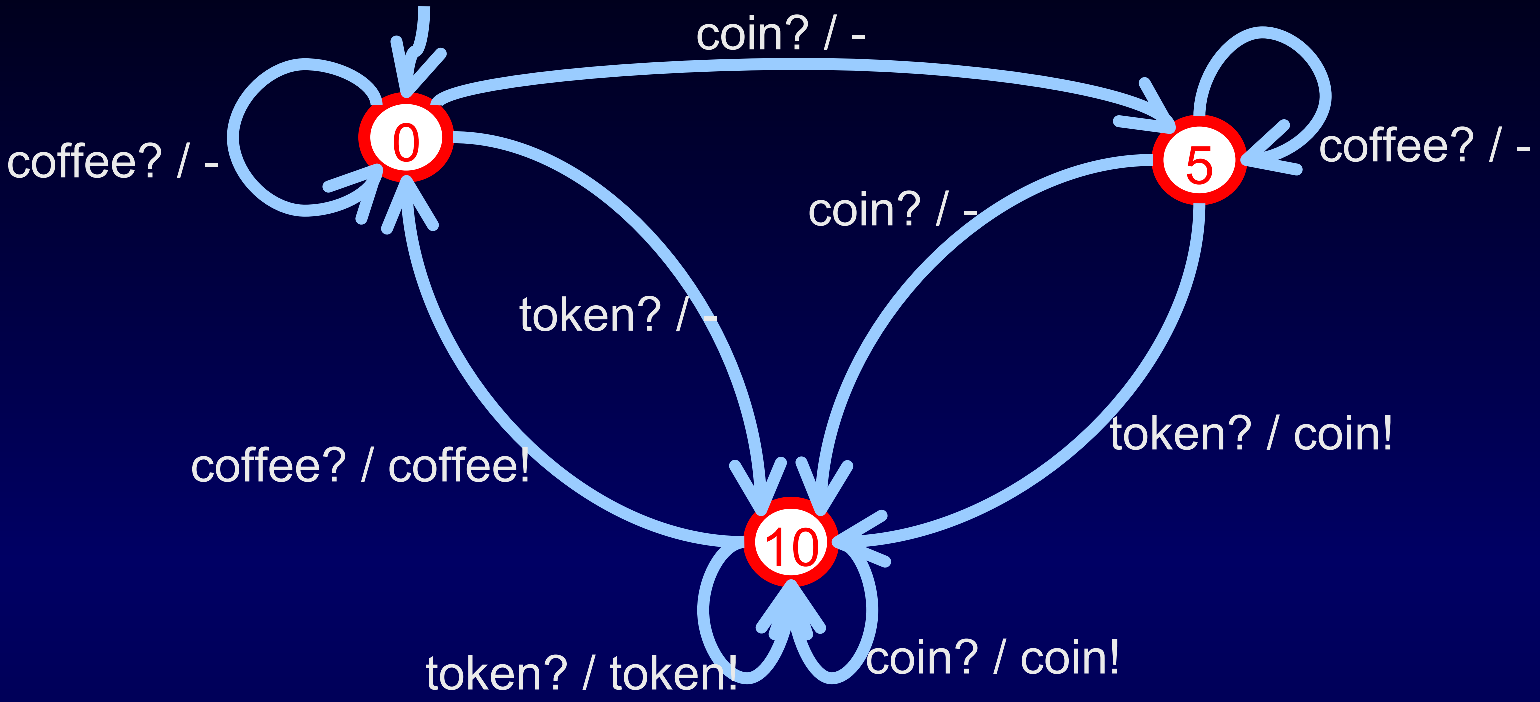
  Apply UIO of state 10 : coffee? / coffee!



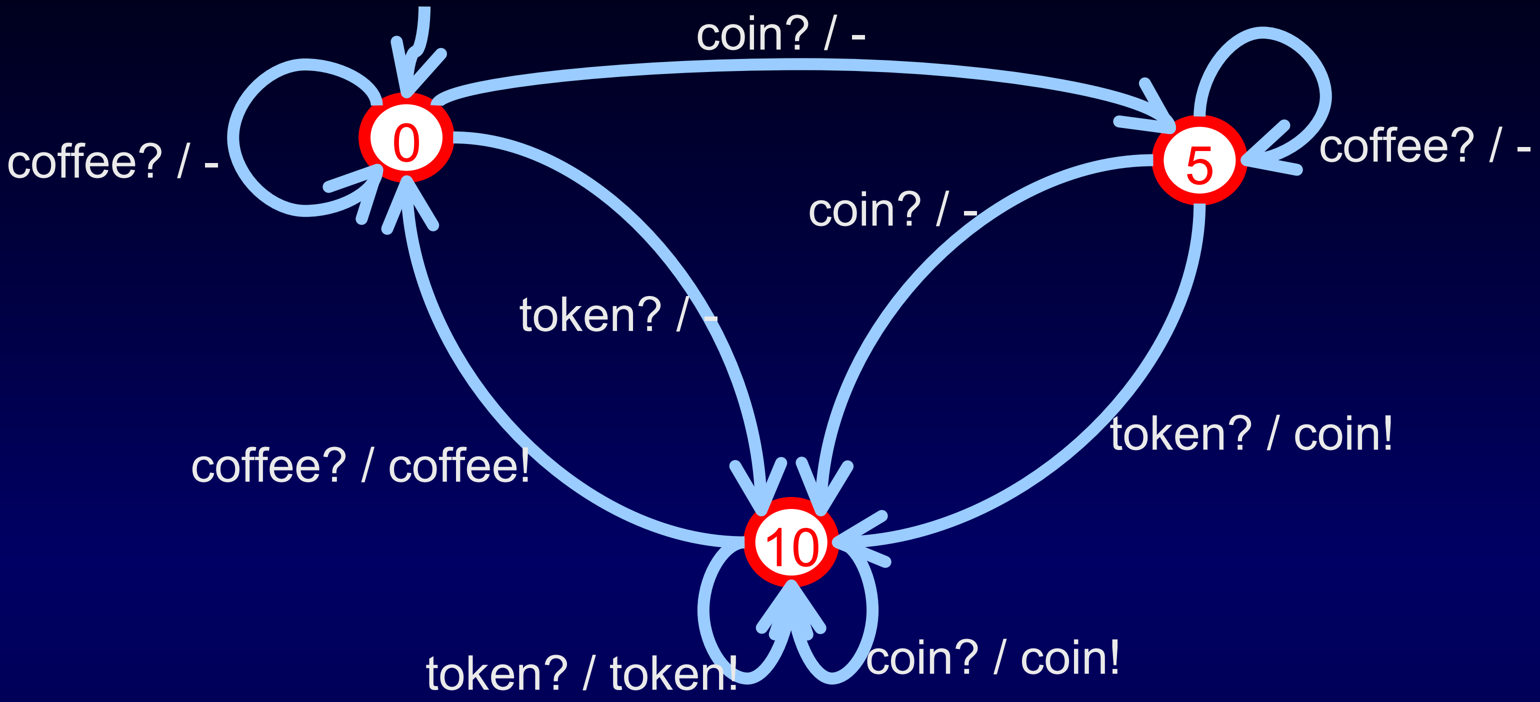
Test case : token? / * coffee? / * coin? / - token? / coin! coffee? / coffee!

Transition Testing - done

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 S1
  - Apply input a?
  - Check output x!
  - Verify state S2
- 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 S1
  - Apply input a?
  - Check output x!
  - Verify state S2
- 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
- If not: exponential growth in test length in number of extra states.

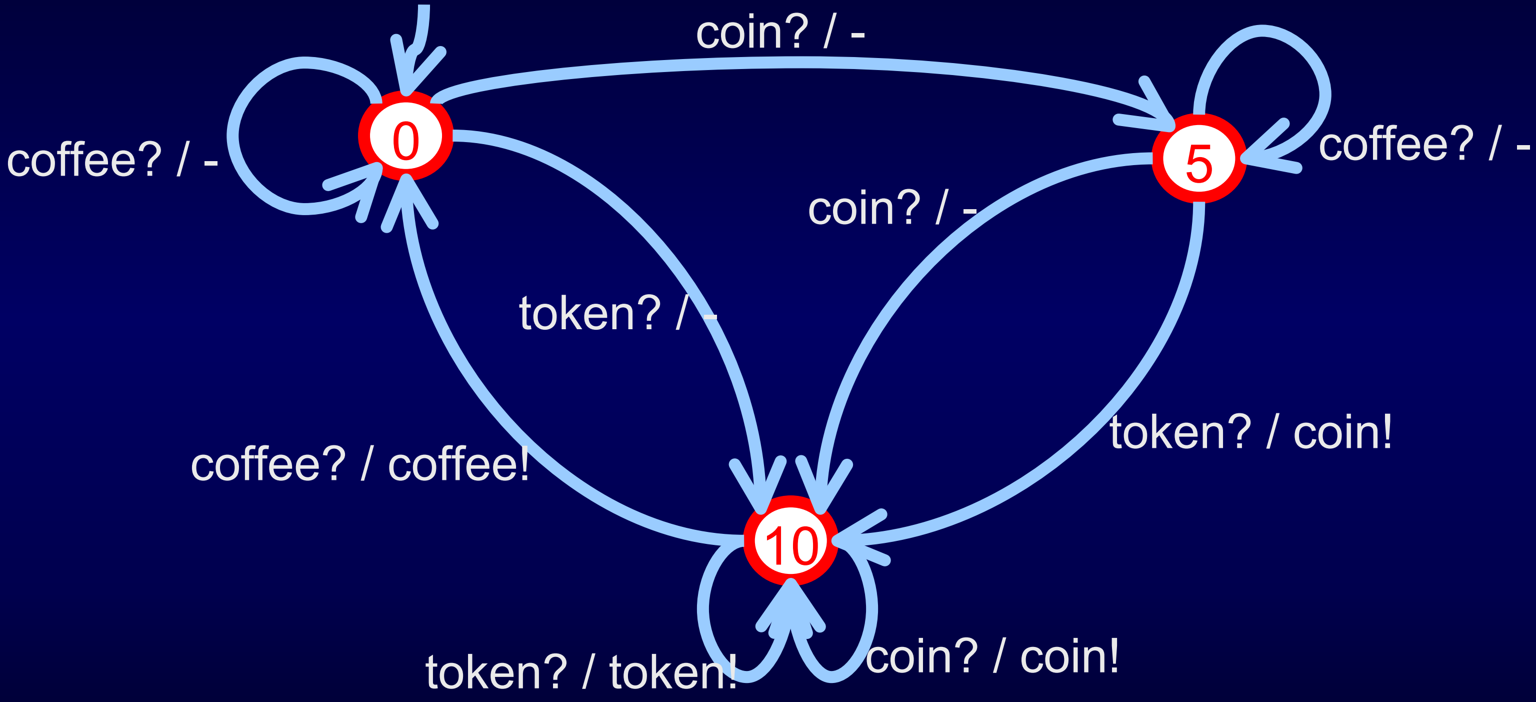
State Coverage

State Coverage

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

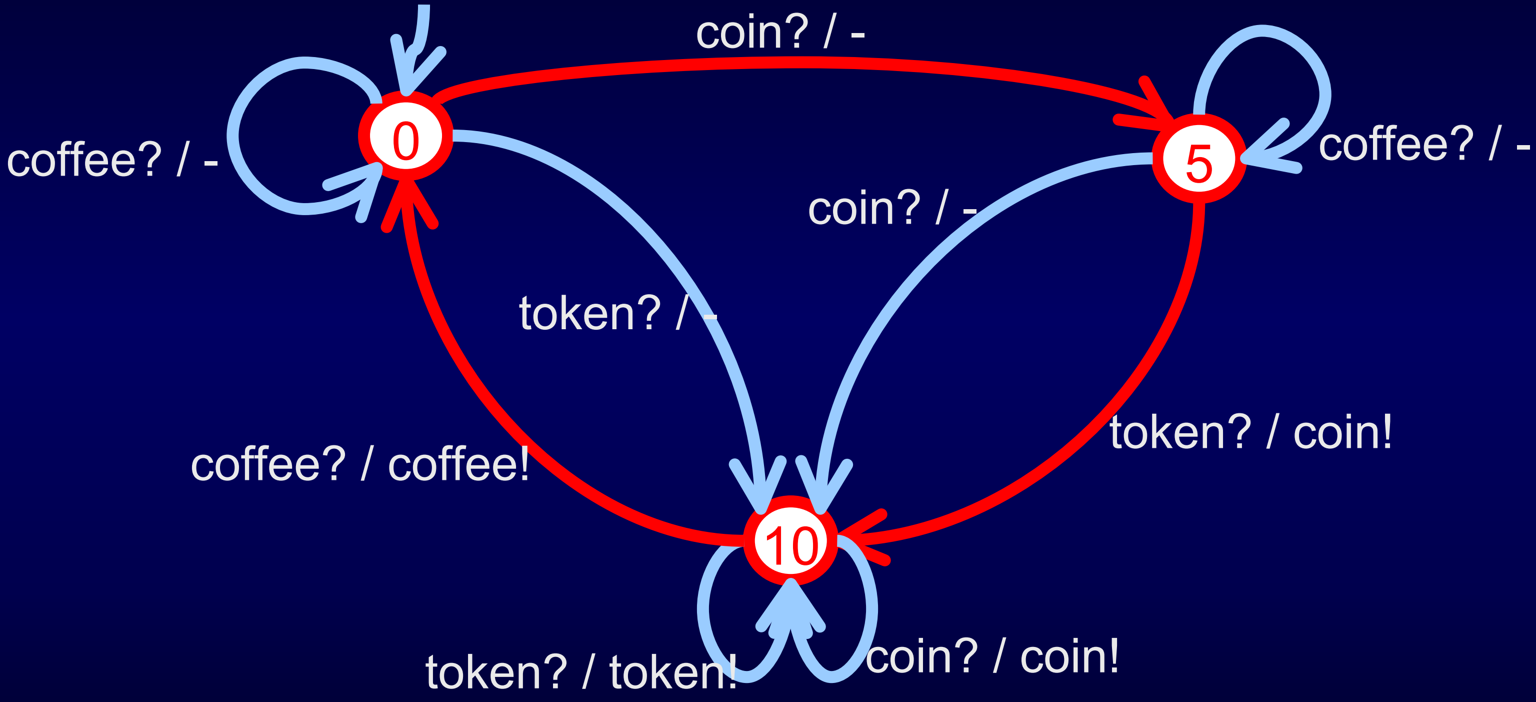
State Coverage

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



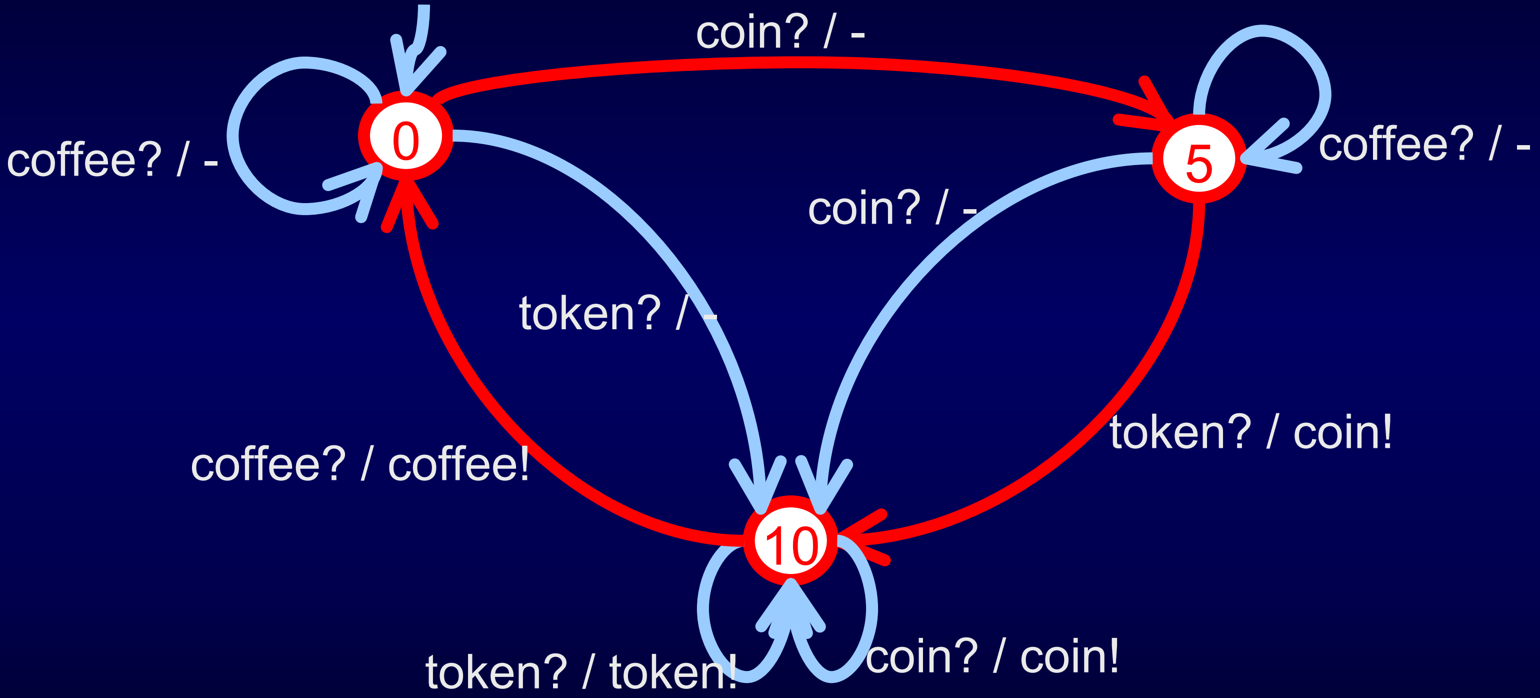
State Coverage

- 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?

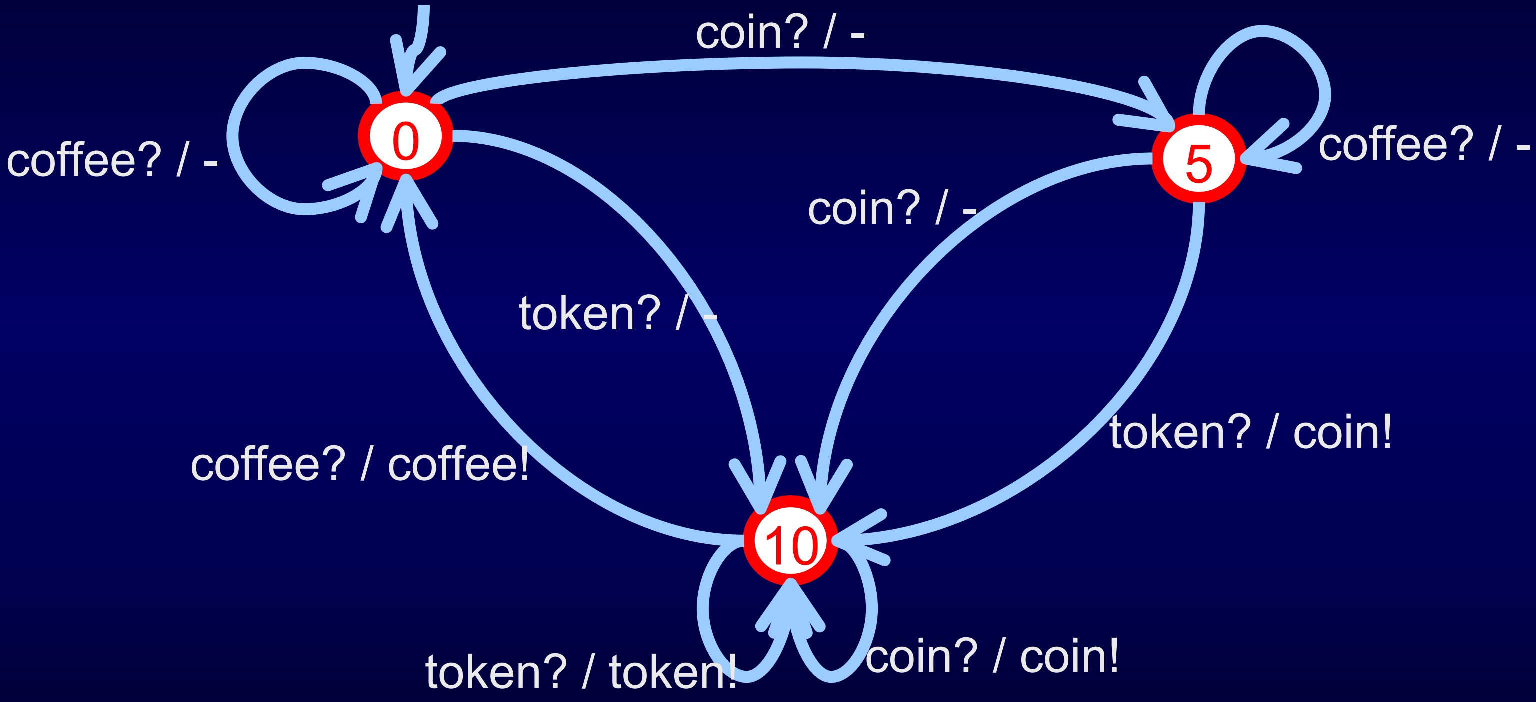
Transition Coverage

Transition Coverage

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

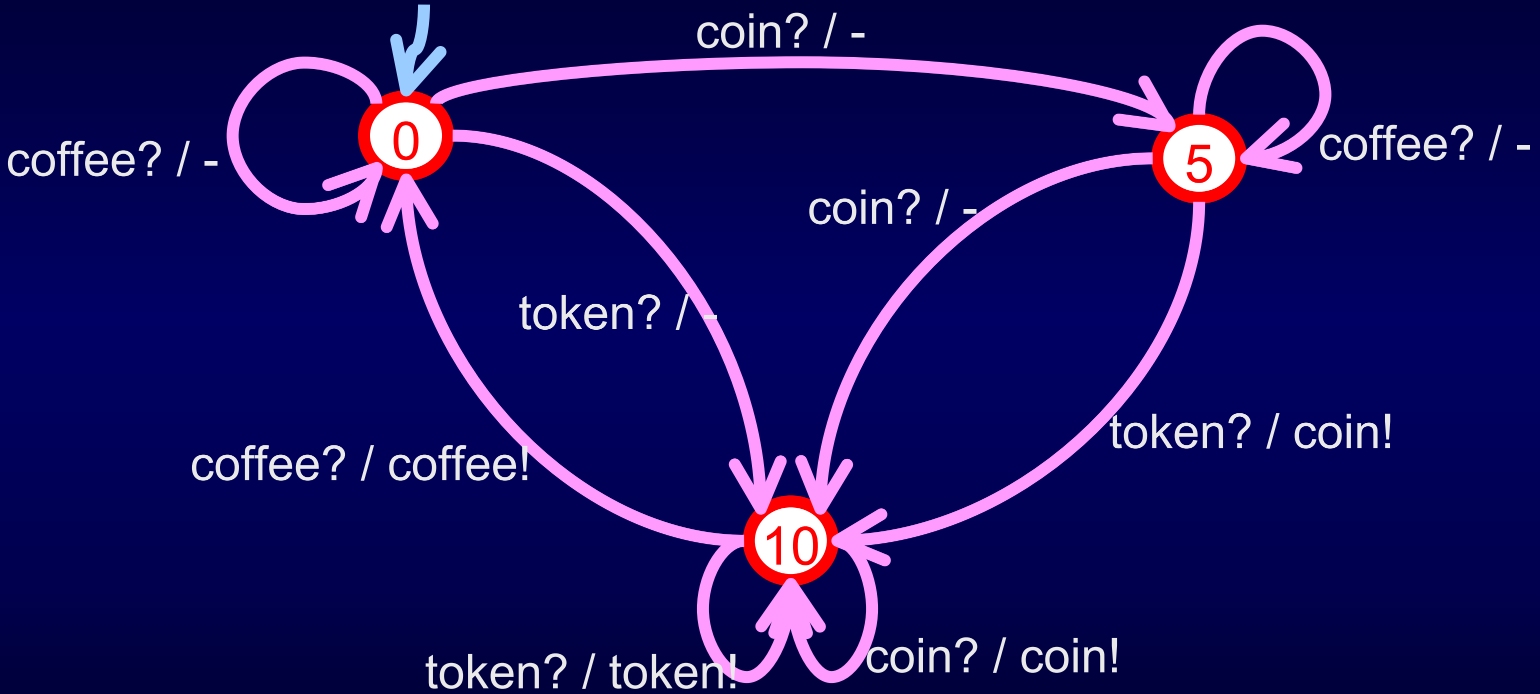
Transition Coverage

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



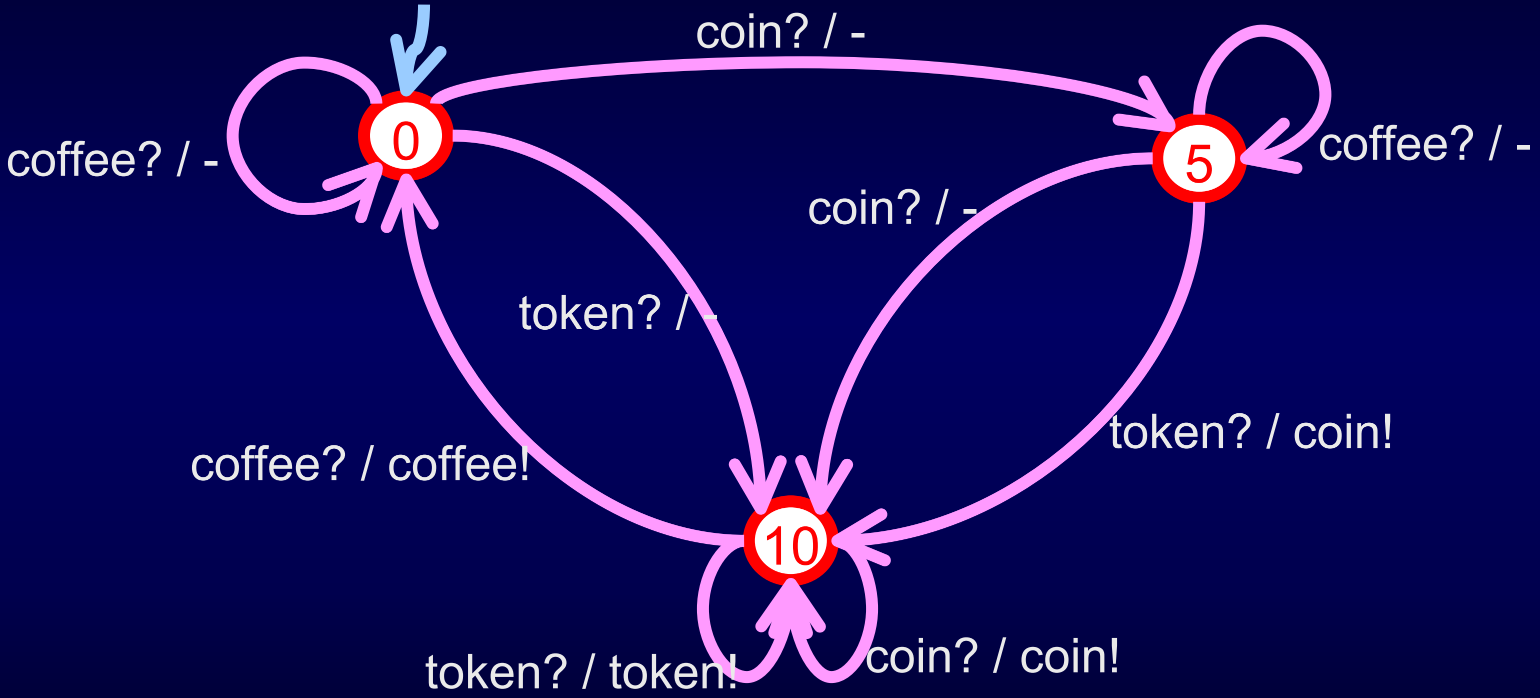
Transition Coverage

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



Transition Coverage

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



Test input sequence :

reset? coffee? coin? coffee? coin? coin? token? coffee? token? coffee? coin? token? coff

FSM Testing vs. LTS Testing

FSM Testing vs. LTS Testing

- FSM based is good if IUT is implemented as FSM with known (limited) number of states.
  - FSM has “more intuitive” theory
  - FSM test suite is complete
    -- but only w.r.t. assumption on number of states
  - FSM test theory has been around for a number of years

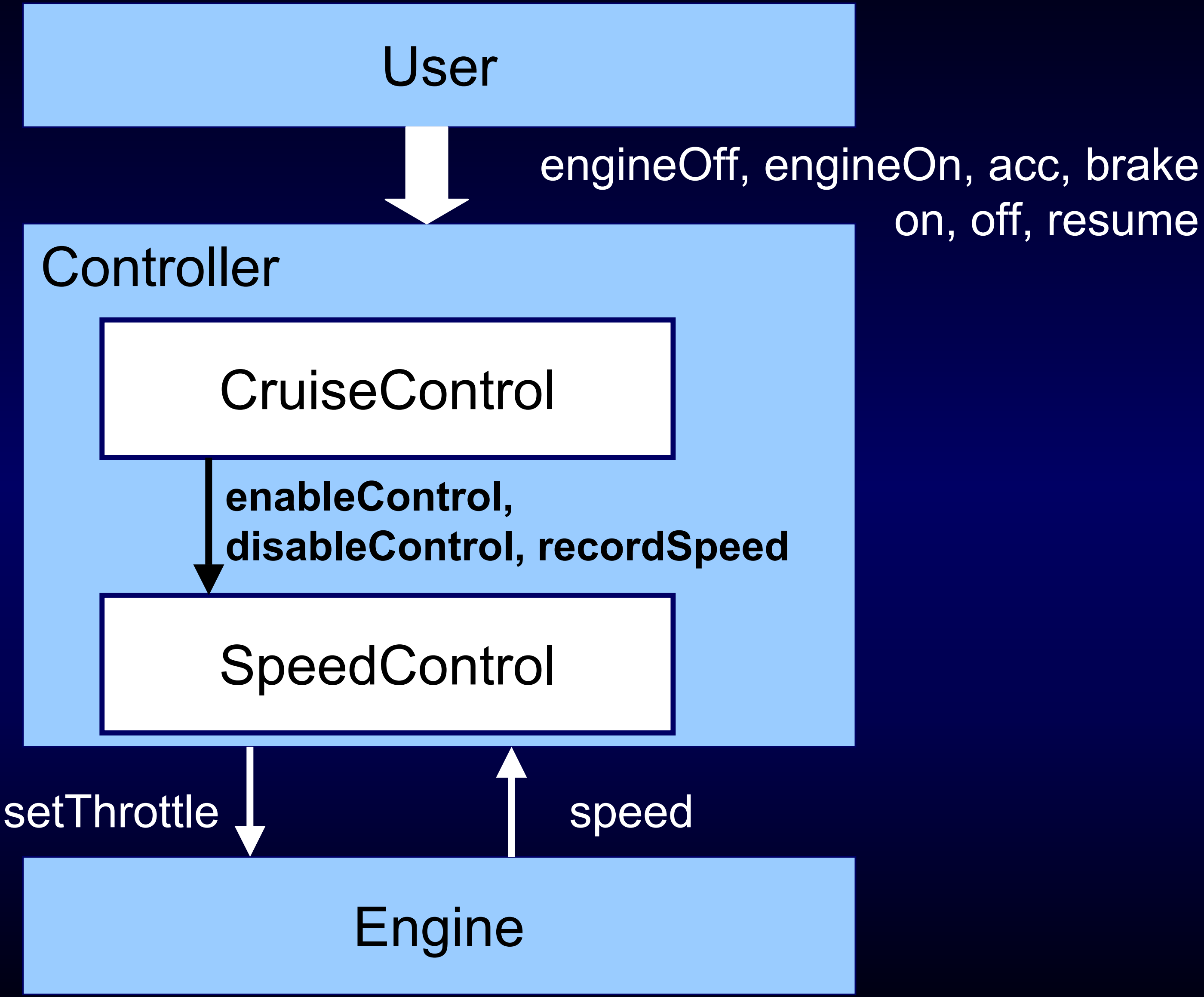
FSM Testing vs. LTS Testing

- FSM based is good if IUT is implemented as FSM with known (limited) number of states.
  - FSM has “more intuitive” theory
  - FSM test suite is complete
    -- but only w.r.t. assumption on number of states
  - FSM test theory has been around for a number of years
- Restrictions on FSM:
  - deterministic
  - completeness

FSM Testing vs. LTS Testing

- FSM based is good if IUT is implemented as FSM with known (limited) number of states.
  - FSM has “more intuitive” theory
  - FSM test suite is complete
    -- but only w.r.t. assumption on number of states
  - FSM test theory has been around for a number of years
- Restrictions on FSM:
  - deterministic
  - completeness
- *Not good for an abstract design model or complex IMPs*
  - FSM has always alternation between input and output
  - Difficult to specify interleaving in FSM
  - FSM is not compositional
  - IMP: Hardware, OS, Application Software: Number of states???

The Cruise Controller



User

engineOff, engineOn, acc, brake
on, off, resume

Controller

CruiseControl

**enableControl,
disableControl, recordSpeed**

SpeedControl

setThrottle

speed

Engine

END