

Testing Object-Oriented Software

Class Testing

Class Testing

- Introduction
- Accounting for Inheritance
- **Testing Method Sequences**
 - Data slices
 - Methods' preconditions and postconditions
- State-based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Motivations & Issues

- Using Inheritance context coverage, control- and data-flow techniques can be used to test methods - what about classes?
- It is argued that testing a class aims at finding the *sequence of operations* for which a class will be in a state that is contradictory to its invariant or what output is expected
- Testing classes for all possible sequences is not usually possible
- The resources required to test a class increase exponentially with the increase in the number of its methods
- It is necessary to devise a way to reduce the number of sequences and still provide sufficient confidence

Example

- Coin box of a vending machine implemented in C++
- The coin box has a simple functionality and the code to control the physical device is omitted
- It accepts only quarters and allows vending when two quarters are received
- It keeps track of total quarters received (`totalQrts`), the current quarters received (`curQrts`), and whether vending is enabled (`allowVend`)
- Functions: adding a quarter, returning current quarters, resetting the coin box to its initial state, and vending

CCoinBox Code

```
class CCoinBox
{
    unsigned totalQtrs;
    unsigned curQtrs;
    unsigned allowVend;

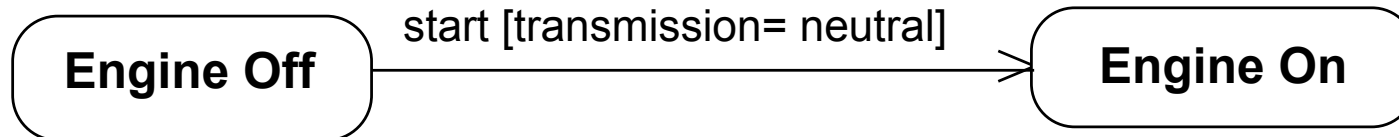
public:
    Ccoinbox() {Reset();}
    void AddQtrs();
    void ReturnQtrs()
    {curQtrs=0;}
    unsigned isAllowedVend()
    {return allowVend;}
    void Reset() {
        totalQtrs = 0;
        allowVend = 0;
        curQtrs = 0;
    }
    void Vend();
};
```

```
void CCoinBox ::AddQtr()
{
    curQtrs = curQtrs + 1;
    if (curQtrs > 1)
        allowVend = 1;
}

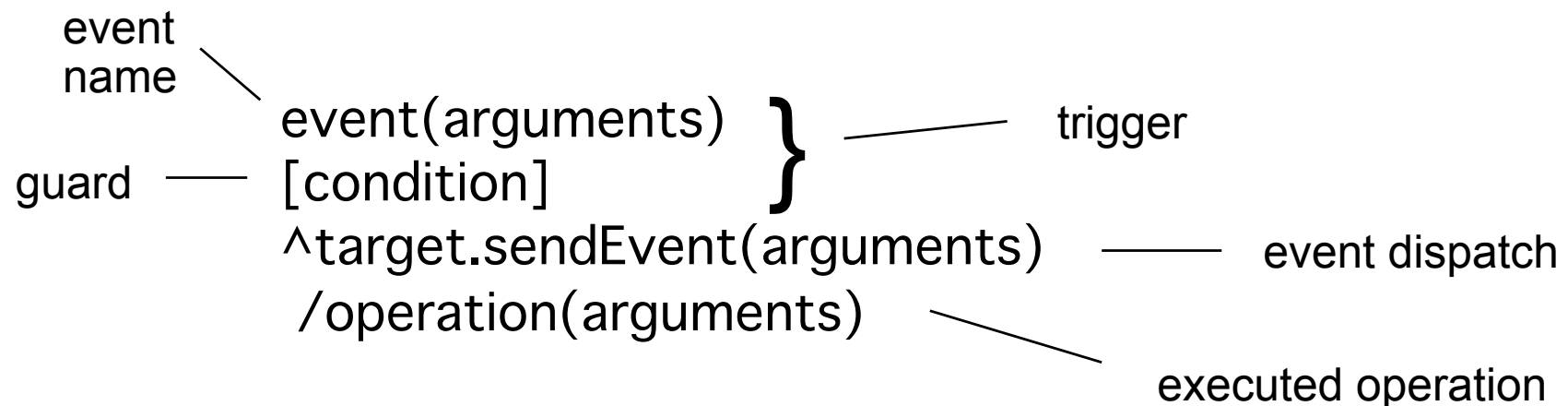
void CCoinBox::Vend()
{
    if(isAllowedVend())
    {
        totalQtrs = totalQtrs + 2;
        curQtrs = curQtrs - 2;
        if (curQtrs < 2) allowVend=0;
    }
}
```


UML Statechart Transitions

- state transitions caused by events, enabled by guard conditions



- have the following general form (all elements are optional)



Object-Oriented Class Testing

- Introduction
- Accounting for Inheritance
- Testing Method Sequences
 - Data slices
 - Methods' preconditions and postconditions
- State-based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Data Slices: Basic Principles

- Goal: Reduce the number of method sequences to test
- The (concrete) state of an object at a single instance of time is equivalent to the aggregated state of each of its data members at that instance
- The correctness of a class depends on
 - Whether the data members are correctly representing the intended state of an object
 - Whether the member functions are correctly manipulating the representation of the object

Slice

- A class can be viewed as a *composition* of slices
- A slice is a quantum of a class with only a single data member and a set of member functions such that each member function can manipulate the values associated with this data member
- Bashir and Goel's class testing strategy is to test *one slice* at a time
- For each slice, test possible sequences of the methods belonging to that slice - equivalent to testing a specific data member, i.e., class partial correctness
- Repeat for all slices to demonstrate class correctness

Slice Formalism

- A class K encapsulates a set of data elements and provides a set of operations
 - $K = \langle D(K), M(K) \rangle$
 - $D(K) = \{d_i \mid d_i @ K\}$
 - $M(K) = \{m_i \mid m_i @K\}$

(where @ denotes the relationship between a class and its elements)
- A Data Slice
 - $\text{Slice}_{d_i}(K) = \langle d_i, M_{d_i}(K) \rangle$
 - $d_i \in D(K)$
 - $M_{d_i}(K) = \{m_i \mid m_i @@ d_i\}$

(where @@ denotes the usage relationship)

Issues

- This is a code-based approach: Many potential problems.
- What if the code is faulty and a method that should access a data member does not? Then sequences of methods that interact through that data member won't be tested and the fault may remain undetected
- How to identify legal sequences of methods?
- What strategy to use when there is a large, possibly infinite, number of legal sequences?

Example C++ (I)

- Part of the GNU C++ library

```
class SampleStatistic {
    friend TestSS;

protected :
    int n;
    double x;
    double x2;
    double minValue, maxValue;

public :
    sampleStatistic();
    virtual ~SampleStatistic();
    virtual void reset();

    virtual void operator += (double);
    int samples();
    double mean();
    double stdDev();
    double var();
    double min();
    double max();
    double confidence(int p_percentage);
    double confidence (double p_value);
    void error(const char * msg);
};
```

Example C++ (II)

```
SampleStatistic::mean() {  
    if (n>0)  
        return (x/n);  
    else  
        return (0.0);  
}  
  
SampleStatistic::operator+=(double  
value) {  
    n += 1;  
    x += value;  
    x2 += (value * value);  
    if(minValue > value)  
        minValue = value;  
    if(maxValue < value)  
        maxValue = value;  
}  
  
double SampleStatistics::stdDev()  
{  
    if (n<=0 || this->var()<=0)  
        return(0);  
    else  
        return (double)sqrt(var());  
}  
  
double SampleStatistics::var() {  
    if (n>1)  
        return ((x2 - ((x*x)/2)) / (  
n-1));  
}
```

Generate MaDUM

- *MaDUM*: Minimal Data Member Usage Matrix
- $n*m$ matrix where n is the number of data members and m represents the number of member methods - it reports on the usage of data members by methods
- *Different usages*: reads, reports, transforms
- Account for *indirect* use of data members, through intermediate member functions

Categorize Member Functions

- Categories: Constructors, Reporters, Transformers, Others
- $M_{d_i}(K) = \{R_{d_i}, C, T_{d_i}, O_{d_i}\}$
 - $R_{d_i} = \{r_{d_i} \mid r_{d_i} \text{ is a reporter method for data member } d_i\}$
 - $C = \{c_i \mid c_i \text{ is a constructor of class } K\}$
 - $T_{d_i} = \{m_{d_i} \mid m_{d_i} \notin R_{d_i} \text{ and } m_{d_i} \notin C \text{ and } m_{d_i} \rightarrow d_i\}$
 - $O_{d_i} = \{o_{d_i} \mid o_{d_i} @@ d_i \text{ and } o_{d_i} \notin R_{d_i} \text{ and } o_{d_i} \notin C \text{ and } o_{d_i} \notin T_{d_i}\}$
- Account for *indirect* use of data members, through intermediate member functions

MaDUM for SampleStatistic

	Sample Statistic	reset	+=	samples	mean	stdDev	var	min	max	confi- dence (int)	confi- dence (dbl)	error
n	t	t	t	r	o	o	o			o	o	
x	t	t	t		o	o	o			o	o	
x2	t	t	t			o	o			o	o	
min- Value	t	t	t					r				
max- Value	t	t	t						r			

Matrix account for indirect use, through called methods

- `stdDev()` does not directly access `x` and `x2`, but calls `var()` that does
- All `SampleStatistic()` does is call `reset()`

Test Procedure

- Classification of methods is used to decide the test steps to be taken:
 1. Test reporters
 2. Test constructors
 3. Test transformers
 4. Test others
- We would like to automate that procedure to the extent possible

Test the Reporters

- I would simply make sure that all classes have get and set methods for all their data members (standard class interface)
- Then I would systematically set and get values of data members and compare the input of the former with the output of the latter.

Test the Constructors

- Constructors initialize data members
- You may have several per class
- We test that
 - All data members are correctly initialized
 - All data members are initialized in correct order
- Run the constructor and append the *reporter* method(s) for each data member
- Verify if in correct initial state (state invariant)
- Only one simple constructor in `SampleStatistic`

Test the Transformers

- Rationale: Test interactions between methods
- For each slice d_i
 1. Instantiate object under test with constructor (already tested)
 2. Create sequences, e.g., all *legal* permutations of methods in T_{d_i}
 - Maximum # sequences: $|C| * |T_{d_i}|!$
 - For example, if 7 member functions in T_{d_i} , with one constructor, this leads to 5040 possible permutations!
 - Question: are permutations sufficient to test interactions between methods ?
 3. Append the reporter method(s) (already tested)
- If several control flow paths in member function: All the paths where the slice d_i is being manipulated must be executed

Test Others

- They do not change the data member d_i
- They do not report on the data member d_i
- They may not even use the data member d_i
- They may not use and change *ANY* data member
- They should be tested as a *stand-alone entity*
- Any standard test technique can be used (WB, BB, Mutation)

CCoinBox Slices

- Slice allowVend:
 - $T_{\text{allowVend}}(\text{CCoinBox}) = \{\text{Reset}, \text{AddQrt}, \text{ReturnQrts}, \text{Vend}\}$
- Slice curQrts:
 - $T_{\text{curQrts}}(\text{CCoinBox}) = \{\text{Reset}, \text{AddQrt}, \text{ReturnQrts}, \text{Vend}\}$
- As a result of the fault, possibly important, sequences would not have been tested
- When the code is correct, both slices show the same set of methods!

Discussions

- Slicing may not be helpful for many classes (see `CCoinBox` example)
- # sequences may still be large
- Many sequences may be impossible (illegal)
- Automation? , e.g., oracle, impossible sequences
- Are we missing many faults by testing slices independently? , e.g., if faults lead to incorrectly defined slices
- Implicitly aimed at classes with no state-dependent behavior? (transformers may need to be executed several times to reach certain states and reveal state faults)

Testing Derived Classes

- When two classes are related through inheritance, a class is *derived* from another class
- The derived class may add facilities or modify the ones provided by the *base* class - it inherits data members and methods from its base class
- Two extreme options for testing a derived class:
 - *Flatten* the derived class and retest all slices of the base class in the new context
 - Only test the *new/redefined* slices of derived class

Bashir and Goel's Strategy

- Assuming the base class has been *adequately* tested, what needs to be tested in the derived class?
- Extend the MaDUM of the base class to generate $\text{MaDUM}_{\text{derived}}$
 - Take the MaDUM of the base class
 - Add a row for each newly defined or re-defined data member of the derived class
 - Add a column for each newly defined or re-defined member function of the derived class

Example: SampleHistogram

```
Class SampleHistogram : public SampleStatistic {
protected:
    short howmanybuckets;
    int *bucketCount;
    double *bucketLimit;
public:
    SampleHistogram(double low, double hi, double bucketWidth
        = -1.0);
    ~SampleHistogram();
    virtual void reset();
    virtual void operator+=(double);
    int similarSamples(double);
    int buckets();
    double bucketThreshold(int i);
    int inBucket(int i);
    void printBuckets(ostream&);
}
```

Filling the $\text{MaDUM}_{\text{Derived}}$





- $C(\text{method})$ = column corresponding to "method" in MaDUM
- If a newly defined member function m_{derived} calls an inherited member function m_{base} of the base class, then the column of the two methods are unioned and the result is stored in the column of the m_{derived}

$$C(m_{\text{derived}}) = C(m_{\text{derived}}) \cup C(m_{\text{base}})$$

- Even though the base class has no a priori knowledge about the data members defined in its derived classes, it may still act on them through dynamic binding and polymorphism. Such a scenario would arise if a member function $m1_{\text{base}}$ calls another member function $m2_{\text{base}}$ and the method $m2$ is redefined ($m2_{\text{derived}}$) in one of the derived classes

$$C(m1_{\text{base}}) = C(m1_{\text{base}}) \cup C(m2_{\text{derived}})$$

MaDUM SampleHistogram

	Sam p - Stat			sam - ples	mean	stdDev	var	min	max	conf . (int)	conf . (dbl)	error	Sam p - Histo			similar - samples	buckets	bucket - Thres - hold	inBucket	print - buckets
n		t	t	r	o	o	o			o	o			t	t					
x		t	t		o		o							t	t					
x2		t	t				o							t	t					
minValue		t	t					r						t	t					
maxValue		t	t						r					t	t					
howMany - Buckets													t	o	o	o	r	o	o	o
bucket Count													t	t	t	r			r	o
bucket Limit													t		o	o		r		o
	C	T	T	R	O	O	O	R	R	O	O	O	C	T	T	O	R	O	O	O

- The MaDUM of SampleHistogram has eight rows, three of them for local data members, and twenty columns, eight for local member functions (2 of them re-defined)
- Both reset and += are re-defined/overridden in SampleHistogram and invoke their counterpart in SampleStatistics - they therefore indirectly access all the data members of SampleStatistics

Test Procedure for Derived Class

- *Local attributes*: Similar to base class testing
- *Retest inherited attributes (I.e., their slices)*:
 - If they are directly or indirectly accessed by a new or re-defined method of the derived class.
 - Check upper right quadrant of $MaDUM_{derived}$
 - We have to ascertain which inherited attributes mandate re-testing - $MaDUM_{derived}$ can be used for automation
 - Once these inherited attributes are identified, the test procedure is similar to slice testing in the base class, but using inherited *and* new/ redefined methods => more testing ...

Identify Inherited Attributes to be Re-tested

- An inherited data member needs to be re-tested if the number of entries in its MaDUM row has increased between $\text{MaDUM}_{\text{base}}$ and $\text{MaDUM}_{\text{derived}}$.
- For SampleHistogram, the set of inherited data members that needs re-testing includes all inherited data members {n, x, x2, minValue, maxValue}

Object-Oriented Class Testing

- Introduction
- Accounting for Inheritance
- Testing Method Sequences
 - Data slices
 - Methods' preconditions and postconditions
- State-based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Method's pre- and post-conditions

- Method pre-condition:
 - A predicate that must be true before an operation is invoked.
 - Specifies constraints that a caller must meet before calling an operation.
- Method post-condition:
 - A predicate that must be true after an operation is invoked.
 - Specifies constraints that the object must ensure after the invocation of the operation.
- Possible notations:
 - Natural language
 - Object Constraint Language (OCL) in the context of UML
 - Extensions to programming languages
 - Eiffel
 - JML

Example - Queue

Class Queue (unbounded queue)
Attribute: Number of elements
in the queue, count

Init(q:Queue)

pre: Queue q does not exist

post: Queue q exists and is
empty

Empty(q:Queue)

pre: Queue q exists

post: Returns 1 if q is empty
(count=0), 0 otherwise
(count>0)

Eque(q:Queue, e:Element)

pre: Queue q exists

post: Element e has been added
to the tail of queue q, and q is
not empty (count=old(count)+1)

Dque(q:Queue, e:Element)

pre: Queue q exists and is
not empty (count>0)

post: Element e has been
removed from q (count=old
(count)-1)

Top(q:Queue, e:Element)

pre: Queue q exists and is
not empty (count>0)

post: The first element is
returned (e)

Approach

Source: F. J. Daniels, K.C. Tai, "Measuring the Effectiveness of Method Test Sequences Derived from Sequencing Constraints", IEEE Trans. Software Engineering, 1999.

1. Get pre- and post-conditions
 - e.g., from UML analysis and design documents
 - or devise them
2. Derive method sequence constraints from pre- and post-conditions
 - They indicate which method sequences (pairs) are allowed or not and under which conditions
 - Automation?
3. Choose a criterion
 - We will define 7 criteria
4. Derive method sequences satisfying the criterion from the method sequence constraints
 - Automation?

Sequencing Constraints

- Pre- and post-conditions imply method sequencing constraints for pairs of methods.
- Assuming m_1 and m_2 are two methods of a class, a sequencing constraints between m_1 and m_2 is defined as a triplet:
(m_1, m_2, C)
 - Such a triplet indicates that m_2 can be executed after m_1 under condition C .
- C is a Boolean expression or a Boolean literal (`True`, `False`).
 - $C = \text{True} \Rightarrow m_2$ can always be executed after m_1 .
 - m_1 's post-condition implies m_2 's pre-condition
 - $C = \text{False} \Rightarrow m_2$ can never be executed after m_1 .
 - m_1 's post-condition implies the negation of m_2 's pre-condition
 - $C = \text{BoolExp} \Rightarrow m_2$ can be executed after m_1 under some conditions.
 - `BoolExp` lists the conditions under which the sequence is possible (disjunctive normal form): $C = C_1 \vee C_2 \vee \dots$

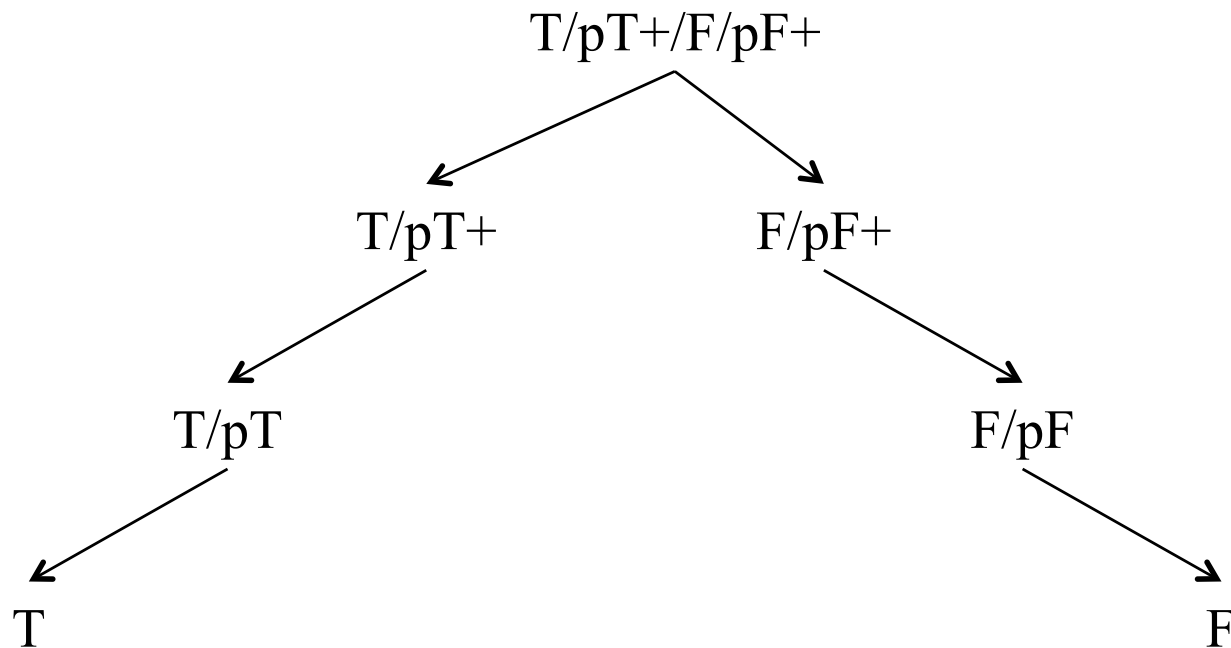
Criteria (I)

- **Always Valid Coverage (T)**
 - Each always-valid constraint must be covered at least once
 - i.e., each $(m1, m2, T)$.
- **Always/Possibly True Coverage (T/pT)**
 - Each always-valid constraint and each possibly-true constraint must be covered at least once.
 - i.e., each $(m1, m2, T)$ and each $(m1, m2, C)$ using one of the disjunction in C (C_1 or C_2 or ...)
- **Always/Possibly True Coverage Plus (T/pT+)**
 - Each always-valid constraint and each possibly-true constraint (using in turn each of the disjunctions) must be covered at least once.
 - i.e., each $(m1, m2, T)$ and each $(m1, m2, C_i)$ using each of the disjunction in C (C_1, C_2 and ...)

Criteria (II)

- Never Valid Coverage (F)
 - Each never-valid constraint must be covered at least once.
 - i.e., each $(m1, m2, F)$
- Never Valid/Possibly False (F/pF)
 - Each never-valid constraint and each possibly false constraint must be covered at least once.
 - i.e., each $(m1, m2, F)$ and each $(m1, m2, \text{not}(C))$
- Never Valid/Possibly False Plus (F/pF+)
 - Each never-Valid constraint and each possibly false constraint must be covered at least once.
 - $\text{not } C = C'_1 \vee C'_2 \vee \dots$
 - i.e., each $(m1, m2, F)$ and each $(m1, m2, \text{not}(C'_1))$, $(m1, m2, \text{not}(C'_2))$...
- Always/Possibly True Plus/Never/Possibly False Plus (T/pT+/F/pF+)

Subsumption



Example - Queue

Sequencing constraints:

$C_1. (\#, \text{Init}, T)$	$C_6. (\#, \text{Eque}, F)$	$C_{11}. (\#, \text{Dque}, F)$	$C_{16}. (\#, \text{Top}, F)$
$C_2. (\text{Init}, \text{Eque}, T)$	$C_7. (\text{Eque}, \text{Dque}, T)$	$C_{12}. (\text{Dque}, \text{Eque}, T)$	$C_{17}. (\text{Top}, \text{Dque}, T)$
$C_3. (\text{Init}, \text{Dque}, F)$	$C_8. (\text{Eque}, \text{Top}, T)$	$C_{13}. (\text{Dque}, \text{Init}, F)$	$C_{18}. (\text{Top}, \text{Eque}, T)$
$C_4. (\text{Init}, \text{Init}, F)$	$C_9. (\text{Eque}, \text{Eque}, T)$	$C_{14}. (\text{Dque}, \text{Dque}, C)$	$C_{19}. (\text{Top}, \text{Init}, F)$
$C_5. (\text{Init}, \text{Top}, F)$	$C_{10}. (\text{Eque}, \text{Init}, F)$	$C_{15}. (\text{Dque}, \text{Top}, C)$	$C_{20}. (\text{Top}, \text{Top}, T)$

Where: $C = \text{count} > 0$

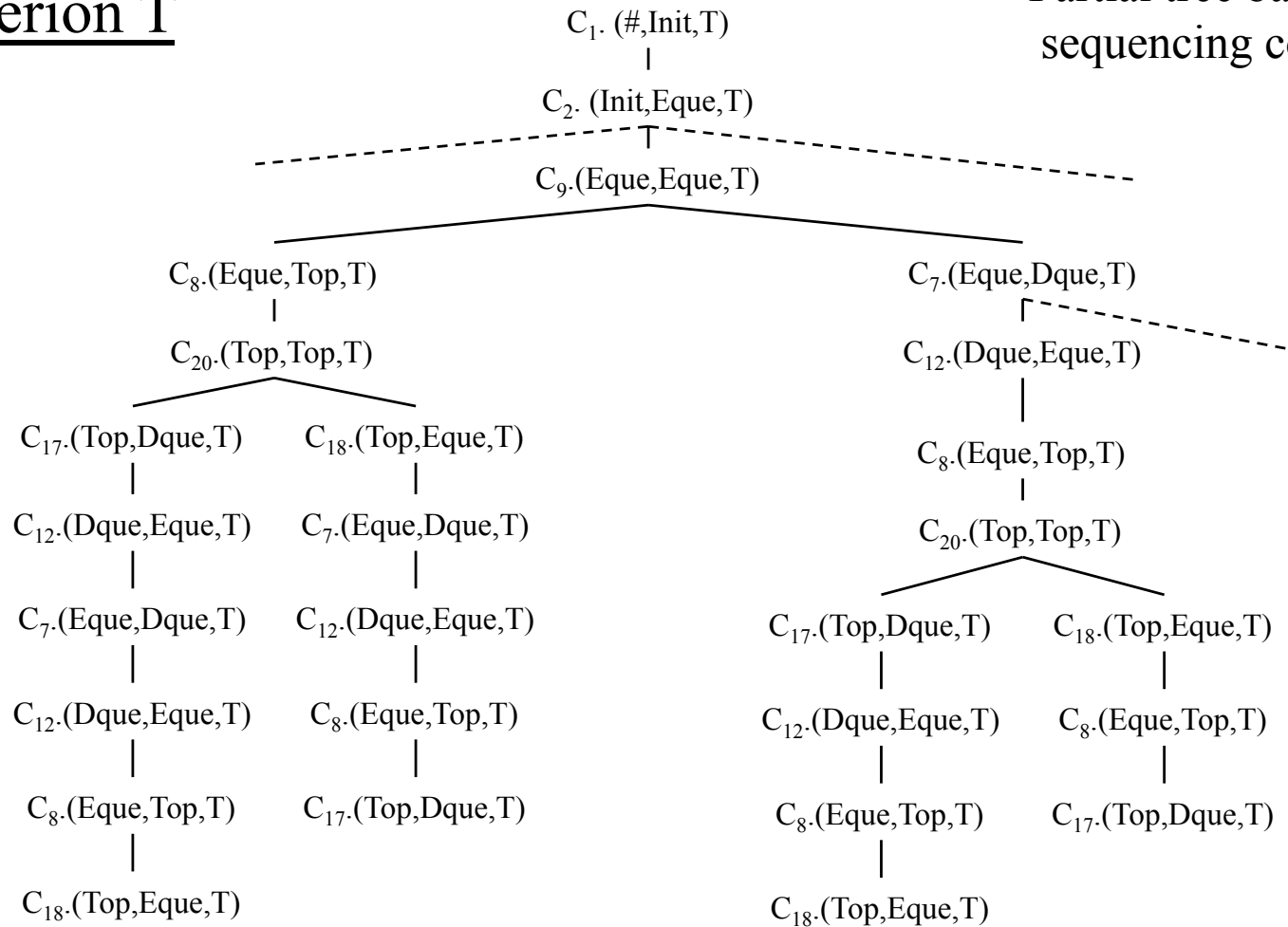
Criterion T requires the use of:

$C_1. (\#, \text{Init}, T)$	$C_2. (\text{Init}, \text{Eque}, T)$	$C_7. (\text{Eque}, \text{Dque}, T)$	$C_{12}. (\text{Dque}, \text{Eque}, T)$
$C_{17}. (\text{Top}, \text{Dque}, T)$	$C_8. (\text{Eque}, \text{Top}, T)$	$C_{18}. (\text{Top}, \text{Eque}, T)$	$C_9. (\text{Eque}, \text{Eque}, T)$
$C_{20}. (\text{Top}, \text{Top}, T)$			

Example - Queue

Criterion T

Partial tree based on sequencing constraints



Discussion

- Automation?
 - From pre- and post-conditions to sequencing constraints
 - From sequencing constraints to 'complete' sequences
- Several sequences can be adequate for a particular criterion
 - Which branch in the tree do we choose?
 - Are they equivalent in terms of fault detection?
 - E.g., the set of statements executed in methods may be different
 - May need to cover some pT constraints to cover certain T ones
- Selecting another criterion than T
 - Criterion F
 - Another representation than the tree we used before may be necessary
 - Criterion pT+
 - The execution of sequence ab , constrained by (a, b, C_Bool) , may require specific calls before the call to a (because of pre- and post-conditions of a).
- Empirical comparison of the different criteria

Empirical Study

- From Daniels and Tai
- 3 C++ programs
- Mutation tool Proteum for C
- 71 mutation operators, 155 mutants
- Criteria (and their names) are not exactly the ones defined here for T/pT+ and F/pF+
- Different oracles: simple self-checks, checking return values (oracles), checking postconditions
- When using appropriate oracles (e.g., check post-conditions and return values), most or all mutants were killed with T/pT-type criteria
- The thoroughness of the oracle has a very significant impact on results.

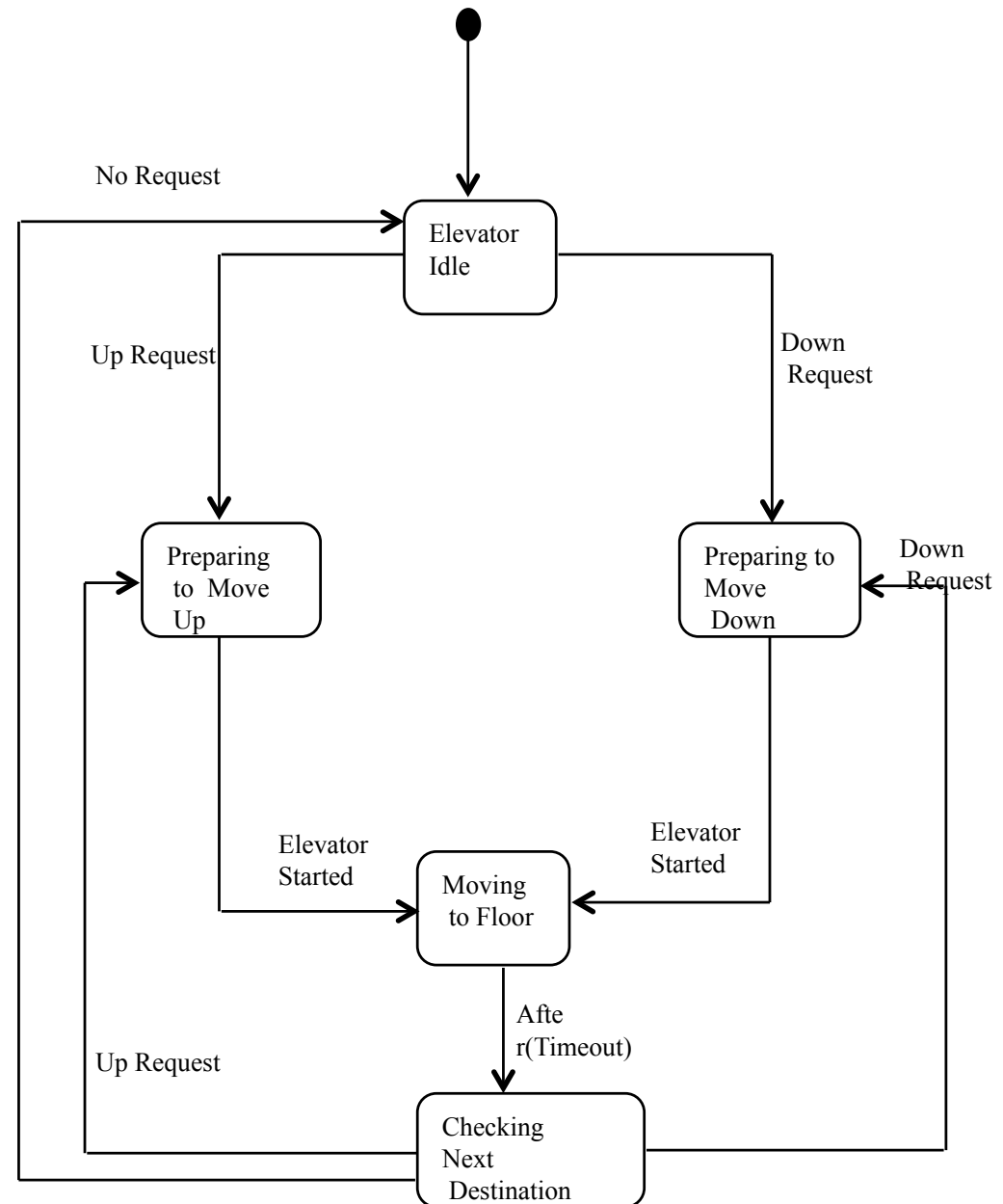
Object-Oriented Class Testing

- Introduction
- Accounting for Inheritance
- Testing Method Sequences
- **State-Based Testing**
 - Methodology
 - Case studies and simulations
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Motivations

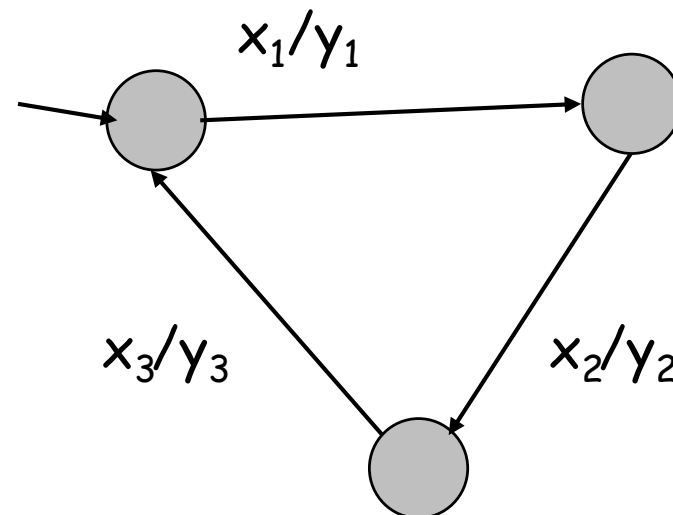
- Does not apply to class (cluster) testing only. Can be applied to any component (e.g., subsystem) modeled by a state machine - we will however refer to classes in the slides
- Started with the testing of communication protocols
- Finding which method sequences to execute to gain confidence in the classes under test may not be easy
- Bashir and Goel's approach can be used for simpler classes. But more complex, *modal* classes should have their behavior modeled by a state-transition model (e.g., Finite State Machine or FSM)
- Such a model can be used as a basis to measure the coverage of class testing and derive test cases to improve that coverage
- This is a natural information source for class testing in a UML context where *statecharts* (a specific state model) are defined for classes whose behavior is state dependent
- The question is now, what does "coverage" mean in the context of a state-transition model?

Simple State Machine



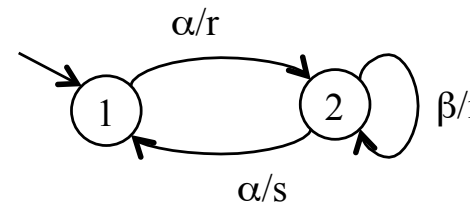
Finite State Machines (I)

- An initial state
- A set of states
- Finite sets of input (X) and output (Y) events
- Transfer function
- Output function



Finite State Machines (II)

S : finite set of states
 S_0 : initial state
 Σ : finite input alphabet
 Ω : finite output alphabet
 δ : transfer function, $\delta: S \times \Sigma \rightarrow S$
 λ : output function, $\lambda: S \times \Sigma \rightarrow \Omega$



Graphical representation

state	α	β
1	2/r	?
2	1/s	2/r

Tabular representation

First steps:

-Deterministic ? Fully specified? Minimal ? Strongly connected ?
 - Verify completeness, e.g., add a transition ' $\beta/-$ ' which loops on

- Choosing test coverage: All states, all transitions, ...
- Impact on test effort?
- Controlability (reach state i): does a 'reset' function exist?
- Observability (observe state i): does a 'status' function exist?

Properties of FSMs

- *Completely specified*: An FSM is said to be completely specified if from each state in M there exists a transition for each input symbol.
- *Strongly connected*: An FSM M is considered strongly connected if for each pair of states (q_i, q_j) there exists an input sequence that takes M from state q_i to q_j .
- *Minimal*: A FSM M is considered minimal if the number of states in M is less than or equal to any other FSM equivalent to M .

Application Domains of FSMs

- Modeling GUIs, network protocols, pacemakers, Teller machines, WEB applications, safety software modeling in nuclear plants, and many more.
- While the FSM's considered in examples are abstract machines, they are abstractions of many real-life machines.
- Embedded systems can commonly be modeled with FSMs: Many real-life devices have computers embedded in them. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses.

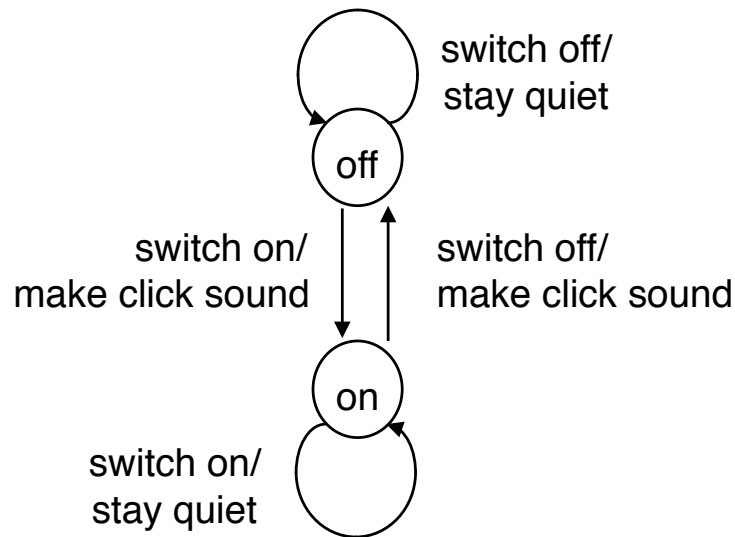
Embedded Systems

- An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another.
- The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by a finite state machine (FSM).
- Fore example, the elevator control FSM



Rocker Switch Example

- Events such as "switch on" and "switch off" may cause the machine to change state, as in the state diagram below for a light with a rocker switch.



input events	current state	next state	outputs
Switch on	on	on	-
Switch on	off	on	click
Switch off	on	off	click
Switch off	off	off	-

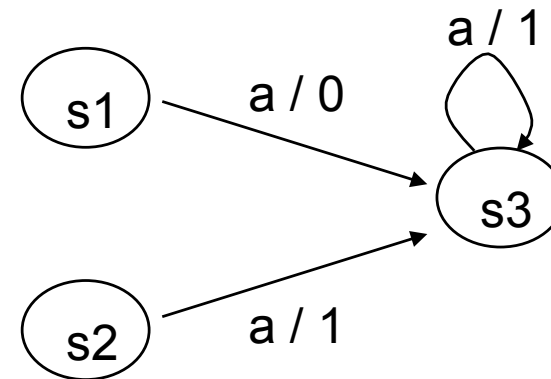
state event	On	Off
Switch on	-	click, on
Switch off	click, off	-

- Some events don't cause a state transition at all, as in attempting to turn on a light that is already on.
- Behaviour of the system in each state has to be defined: State ON - light is emitted out of bulb, State OFF - no light emitted.

Equivalence

- Two states are equivalent if
 - for every input sequence, the output sequences from the two states are the same

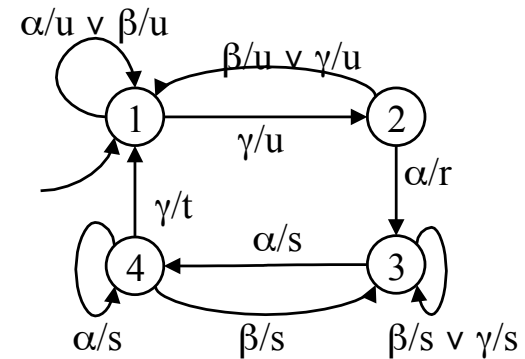
s1 and **s2** are equivalent?
s2 and **s3** are equivalent?



- A FSM with no two equivalent states is reduced or minimized

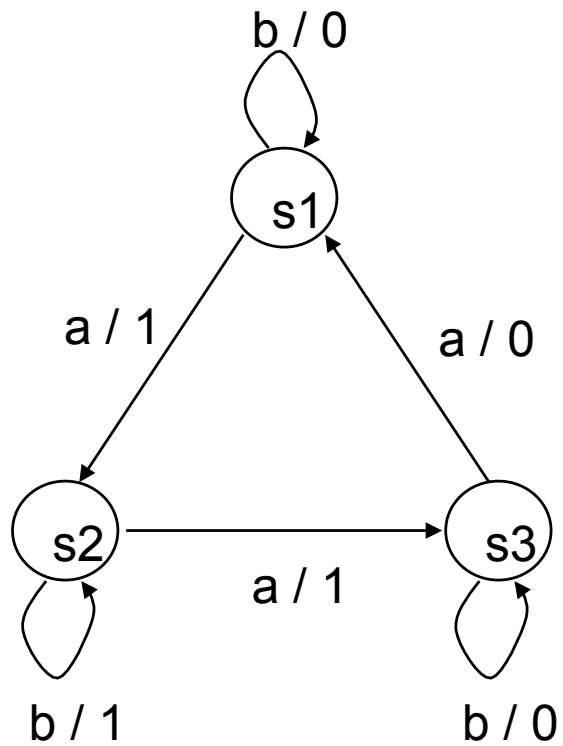
Fault Taxonomy

1. Missing or incorrect transition to a valid state, based on a correct input (Transfer fault)
 - Transition from 2 to 1 on input β is missing
 - Transition from 1 to 2 (on input γ) is in fact from 1 to 3
2. Missing or incorrect output (action), based on a correct input and transition
 - Transition from 1 to 2 (on input γ) outputs r (instead of u)
3. Corrupt state: Based on a correct input, the implementation computes a state that is not valid (additional state).
4. Sneak path (extra transition): The implementation accepts an input that is illegal or unspecified for a state
 - Transition from 1 to 4 on input φ
5. Illegal input failure: The implementation fails to handle an illegal message correctly (incorrect output, state corrupted)

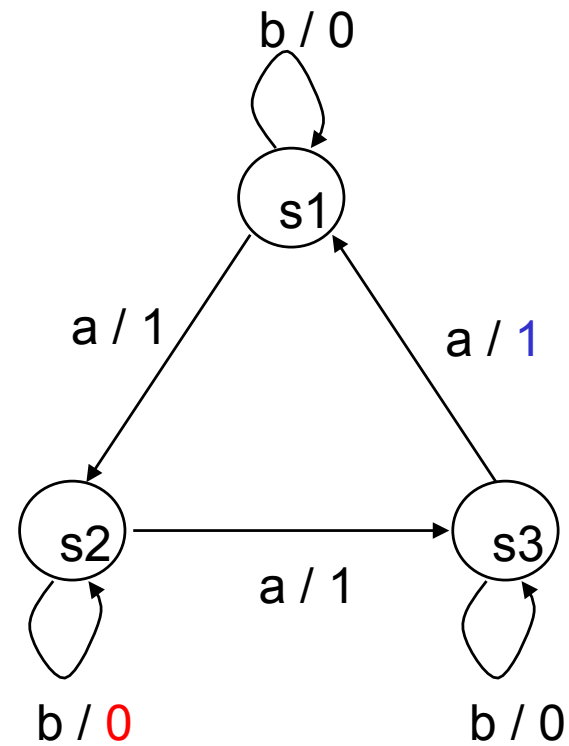


Output faults

Specification

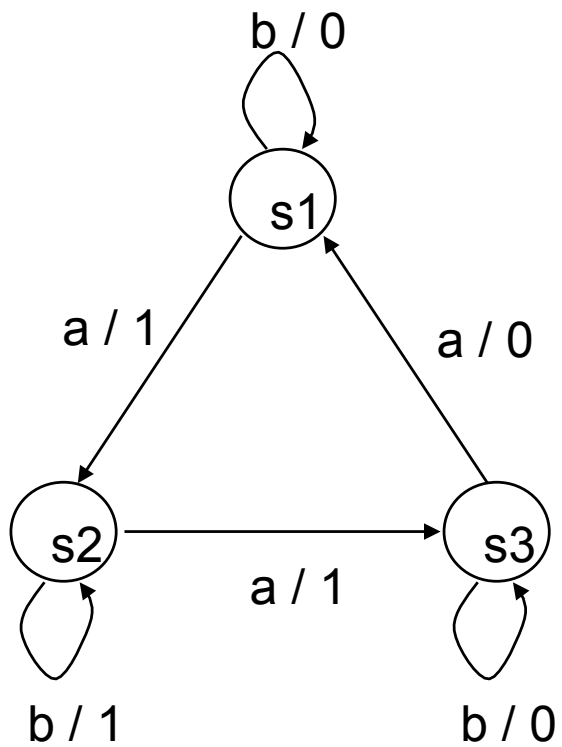


Implementation

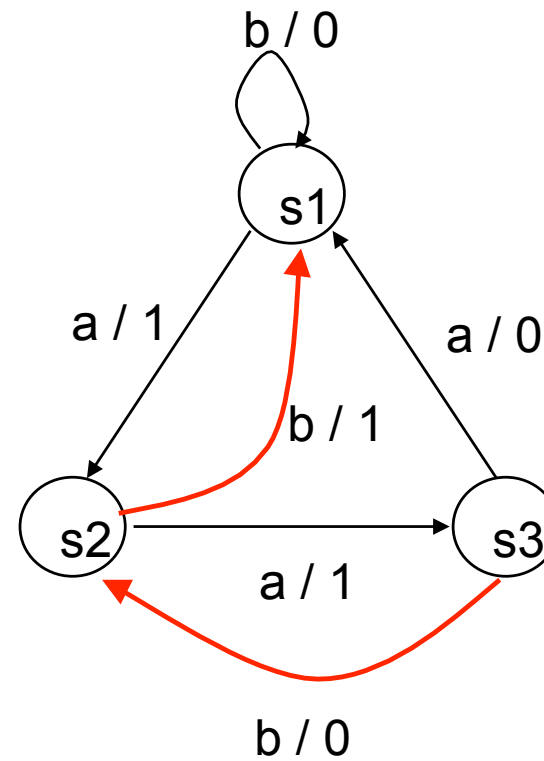


Transfer faults

Specification

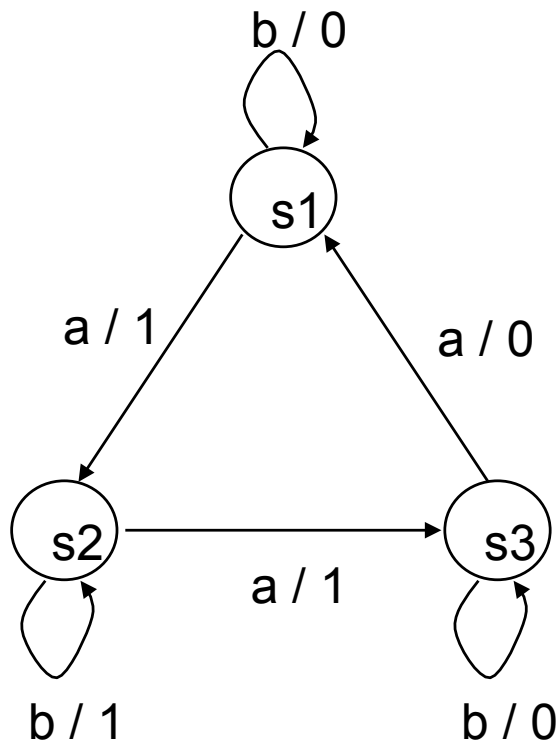


Implementation

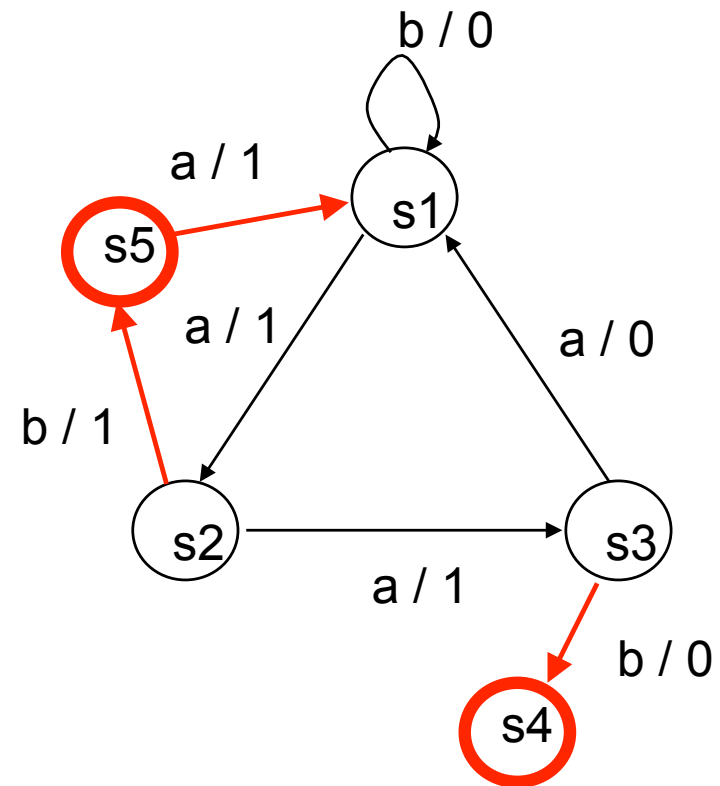


Transfer faults with additional (corrupt) states

Specification

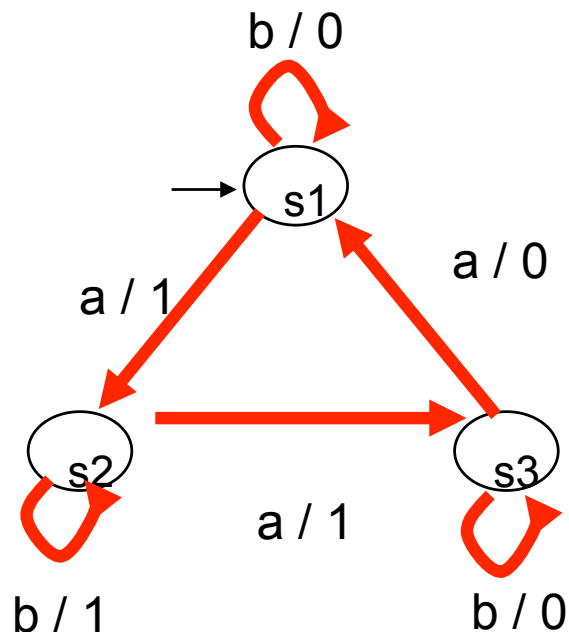


Implementation



Transition Tour (TT) (1)

- A transition tour of a FSM
 - A path starting at the initial state, traverses every transition **at least once**, and returns to the initial state



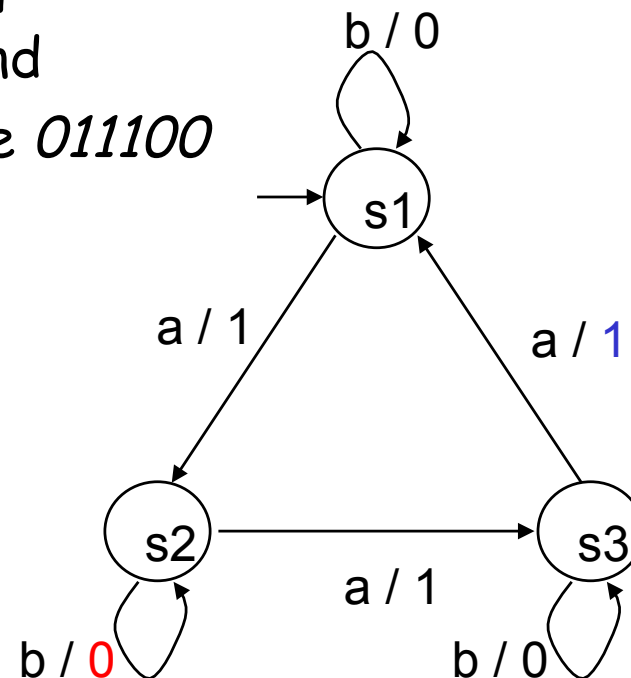
bababa

$s1 \rightarrow s1 \rightarrow s2 \rightarrow s2 \rightarrow s3 \rightarrow s3 \rightarrow s1$

0 1 1 1 0 0

TT-method (2)

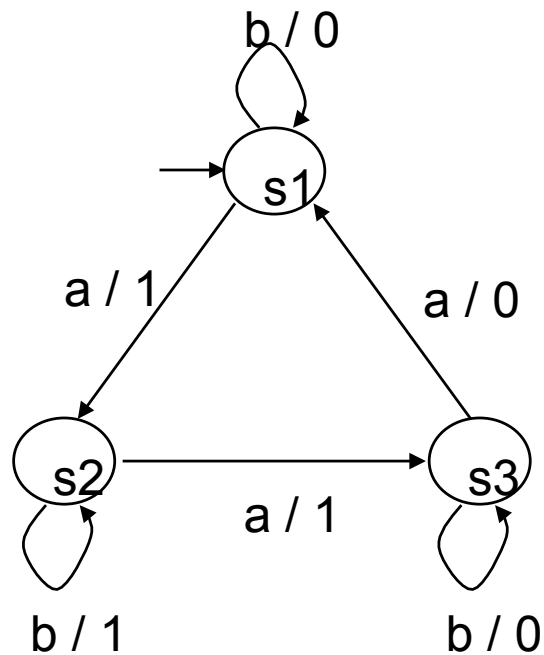
- From a transition tour, we identify a test suite consisting of an input sequence and its expected output sequence
 - The input sequence *bababa* and
 - Its expected output sequence *011100*
 - *Observed sequence is ?*
 - *Oracle: compare sequences*



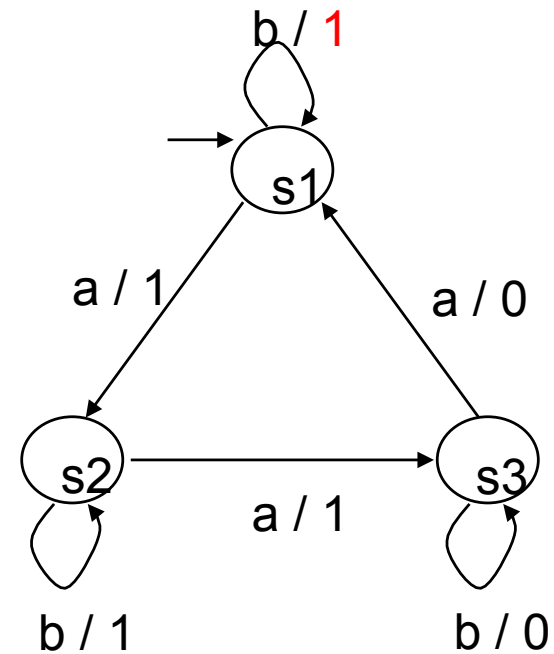
TT-method (3)

- Transition tours can find all output faults

Specification
Input sequence: *bababa*
Expected output sequence: *011100*



Implementation
Input sequence: *bababa*
Observed output sequence: *111100*



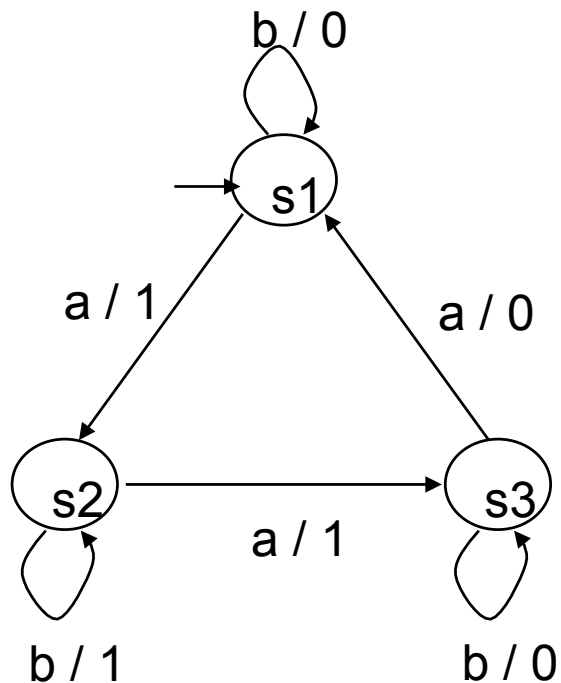
TT-method problem

- Transition tours cannot always *distinguish* transfer faults

Specification

Input sequence: *bababa*

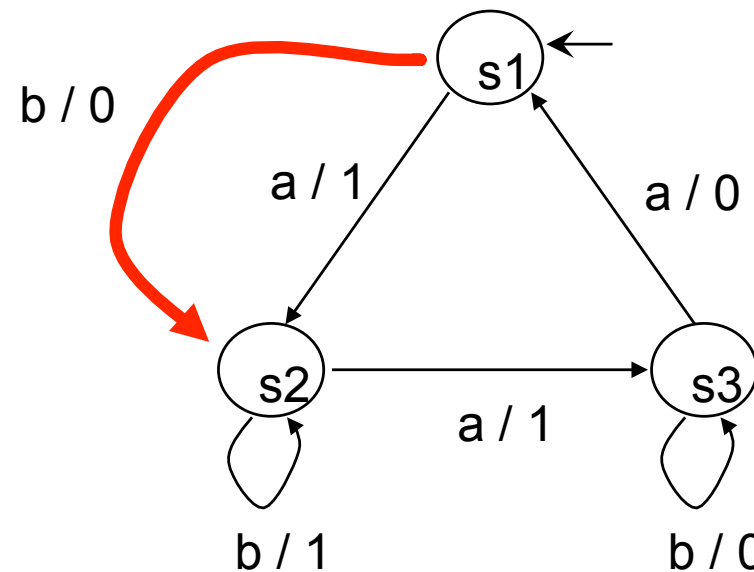
Expected output sequence: *011100*



Implementation

Input sequence: *bababa*

Observed output sequence: *010001*



Detecting Transfer Faults

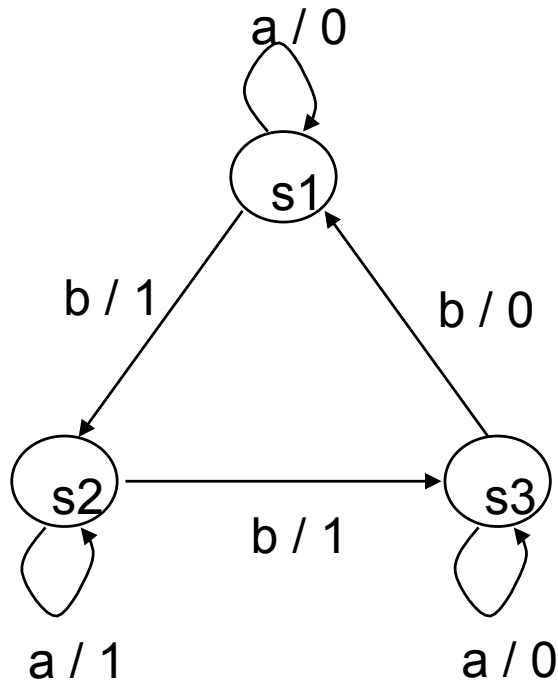
- The main idea
 - Generate a test suite such that, for every transition (s, i, o, s') :
 - Step 1: Puts the implementation into state s (Setup)
 - Step 2: Applies input i and check whether the actual output is o (Output fault)
 - Step 3: Determines whether the target state of the implementation is s' (Transfer fault)
 - Step 3: one possibility is that the implementation makes the state observable (e.g., "status" function) - but it is not always possible, so let's put it aside for now

Distinguishing sequence

- An input sequence is a **distinguishing sequence** if
 - After applying the input sequence, we can determine ***the source state*** by observing the produced output sequence

Example 1

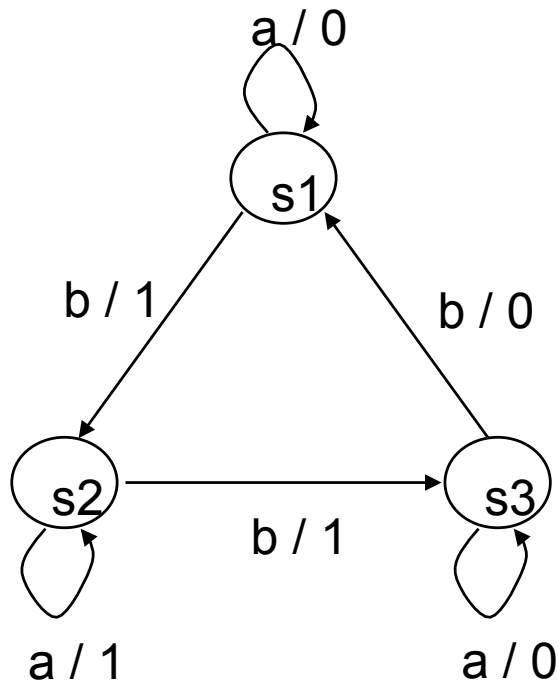
- *a* is not a distinguishing sequence ...



Initial state	Input seq	Output seq	Final state
S1	a	0	S1
S2	a	1	S2
S3	a	0	S3

Example 2

- *ab* is a distinguishing sequence

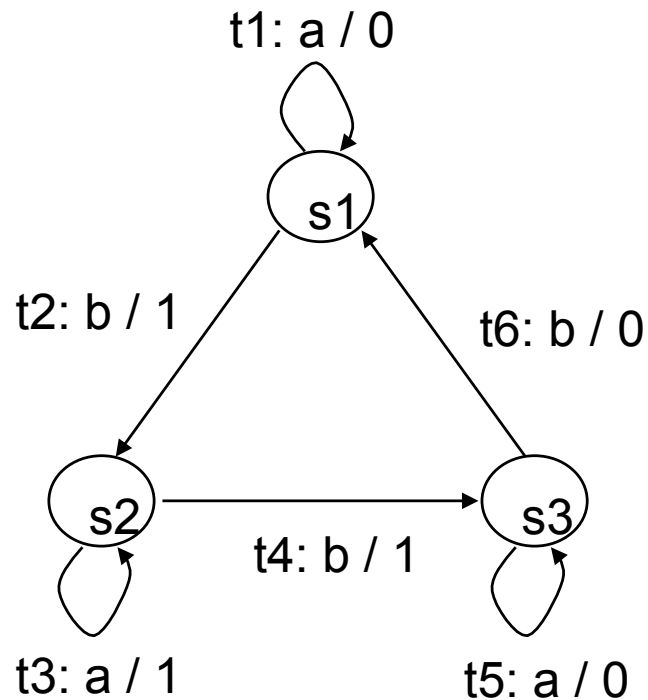


Initial state	Input seq	Output seq	Final state
S1	ab	01	S2
S2	ab	11	S3
S3	ab	00	S1

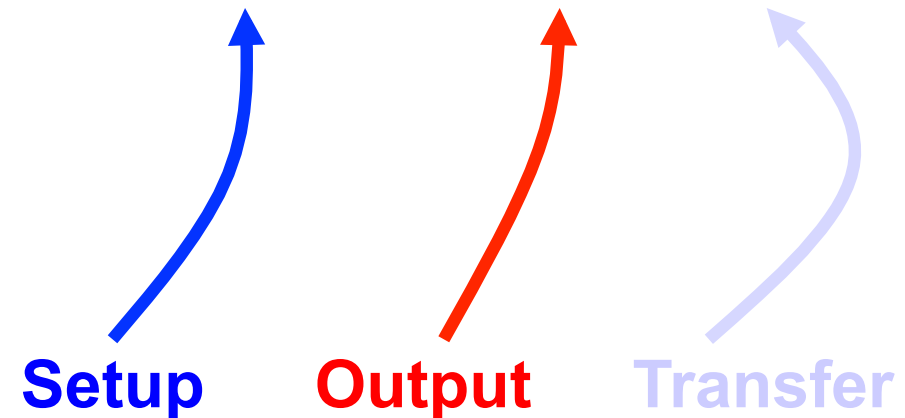
DS-method

- For every transition (s, i, o, s')
 - Step 1: Put the implementation into state s (Setup)
 - Step 2: Apply input i and check whether the actual output is o (Output fault)
 - Step 3: Determine whether the target state of the implementation is s' using a distinguishing sequence (Transfer fault)

DS-method: A test suite

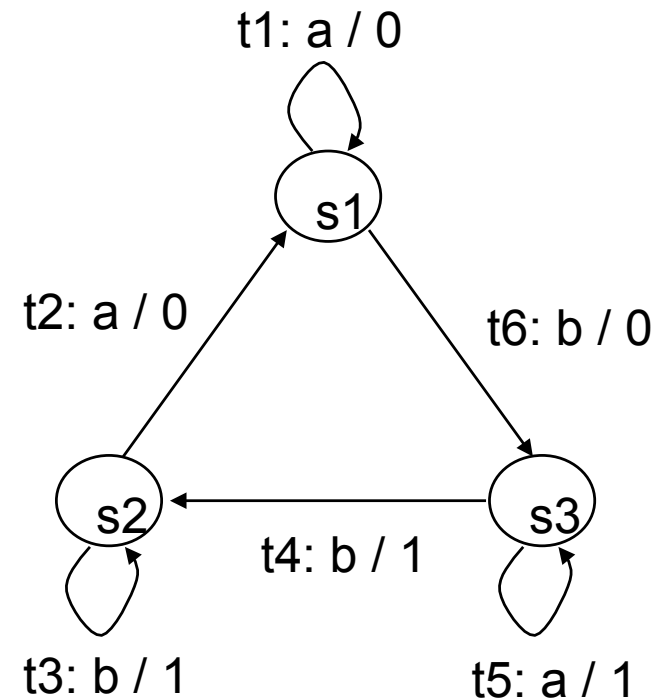


t1: **reset/null** **a/0** a/0 b/1
t2: **reset/null** **b/1** a/1 b/1
t3: **reset/null** **b/1** **a/1** a/1 b/1
t4: **reset/null** **b/1** **b/1** a/0 b/0
t5: **reset/null** **b/1** **b/1** **a/0** a/0 b/0
t6: **reset/null** **b/1** **b/1** **b/0** a/0 b/1



DS-method pros and cons

- Few FSMs possess a distinguishing sequence
- Even if an FSM has a distinguishing sequence, the sequence may be too long
- Example: there is no DS
 - A DS cannot start with a ($s1, s2$)
 - A DS cannot start with b ($s2, s3$)

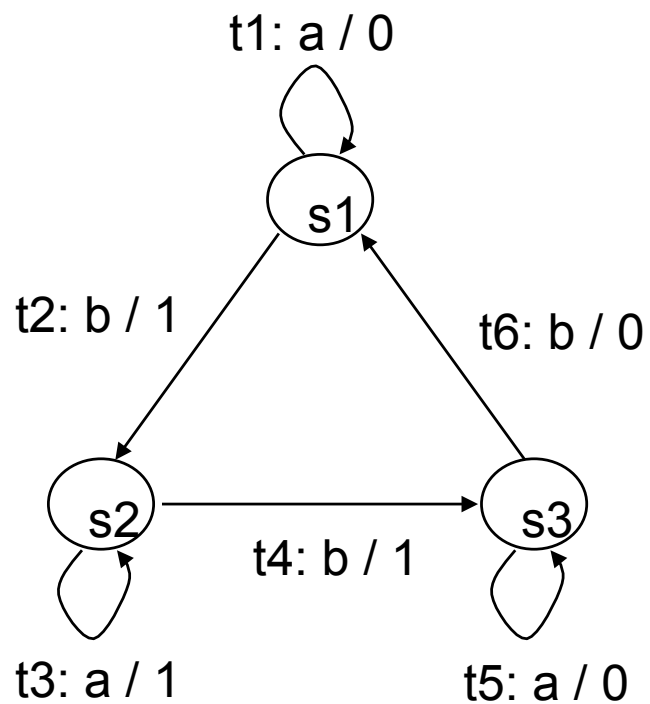


W-method

- A set of input sequences is a **characterizing set** if
 - *After applying all input sequences in the set, we can determine the source state by observing the output sequences*

Example

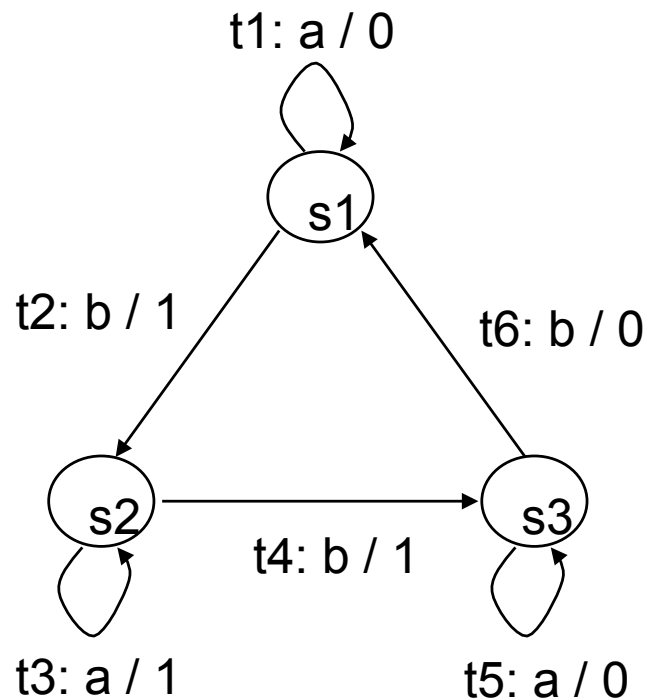
- $\{a,b\}$ is a characterizing set?



Initial state	Input seq	Output seq	Final state
S1	a	0	S1
S2	a	1	S2
S3	a	0	S3

Initial state	Input seq	Output seq	Final state
S1	b	1	S3
S2	b	1	S2
S3	b	0	S2

W-method: A test suite



t1: reset/null **a/0** a/0
t1: reset/null **a/0** b/1

t2: reset/null **b/1** a/1
t2: reset/null **b/1** b/1

t3: reset/null **b/1** **a/1** a/1
t3: reset/null **b/1** **a/1** b/1

t4: reset/null **b/1** **b/1** a/0
t4: reset/null **b/1** **b/1** b/0

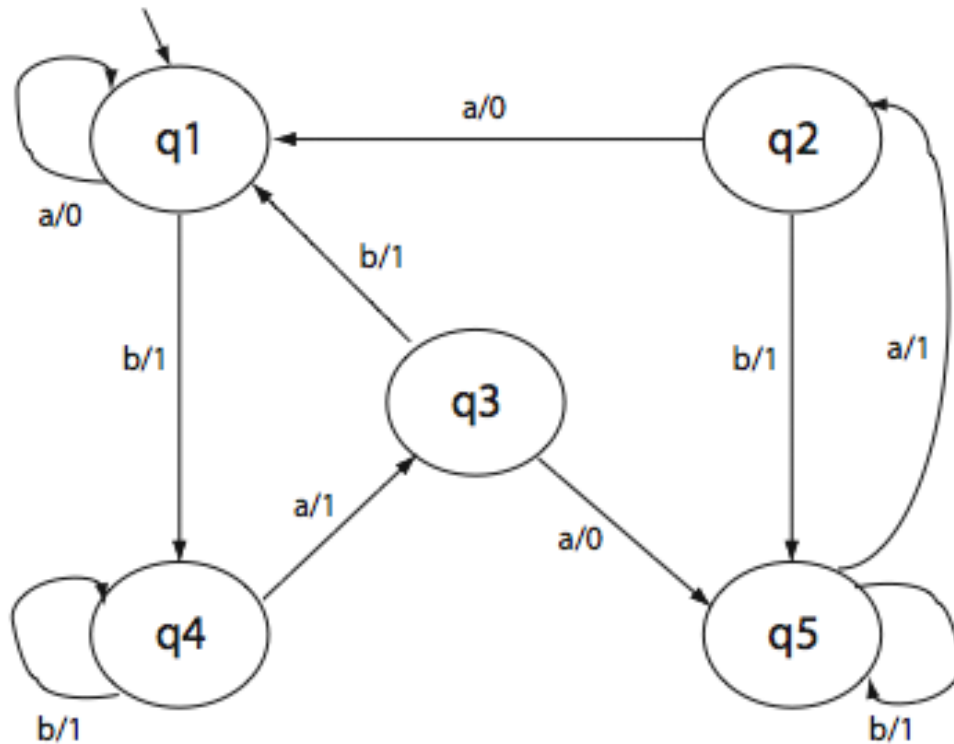
t5: reset/null **b/1** **b/1** **a/0** a/0
t5: reset/null **b/1** **b/1** **a/0** b/0

t6: reset/null **b/1** **b/1** **b/0** a/0
t6: reset/null **b/1** **b/1** **b/0** b/1

W-method pros and cons

- Although every FSM has a characterizing set, the set may have too many elements (expensive)
- Both distinguishing sequences and characterizing sets impose too strong requirements
 - We are just interested in determining whether the target state is a specific state or not
 - State identification versus state verification

Another Example of W



$W = \{baaa, aa, aaa\}$

$O(baaa, q1) = 1101$

$O(baaa, q2) = 1100$

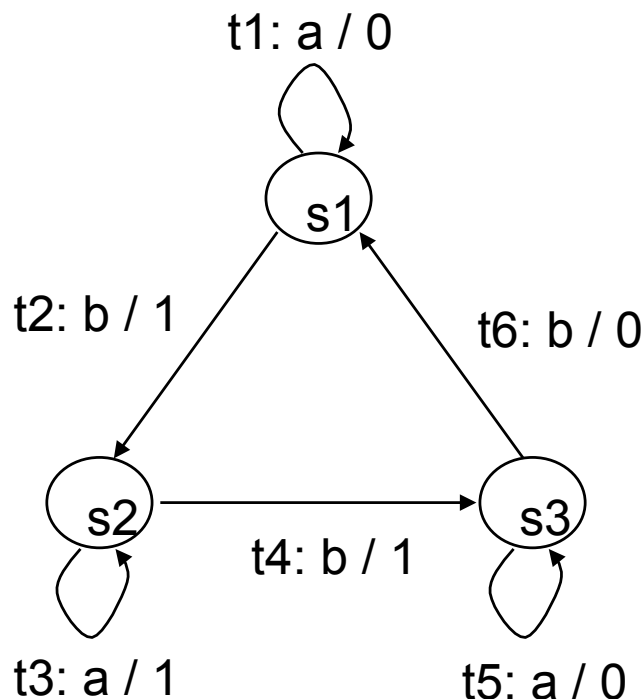
Thus $baaa$ distinguishes state $q1$ from $q2$
as $O(baaa, q1) \neq O(baaa, q2)$

Unique-input-output (UIO) - method

- Let s be a state.
- An input sequence is a **UIO sequence** for s if
 - After applying the input sequence, *we can determine the source state is s or not by observing the output sequence*

UIO-method & states s_2, s_3

- a is a UIO sequence for s_2
- b is a UIO sequence for s_3

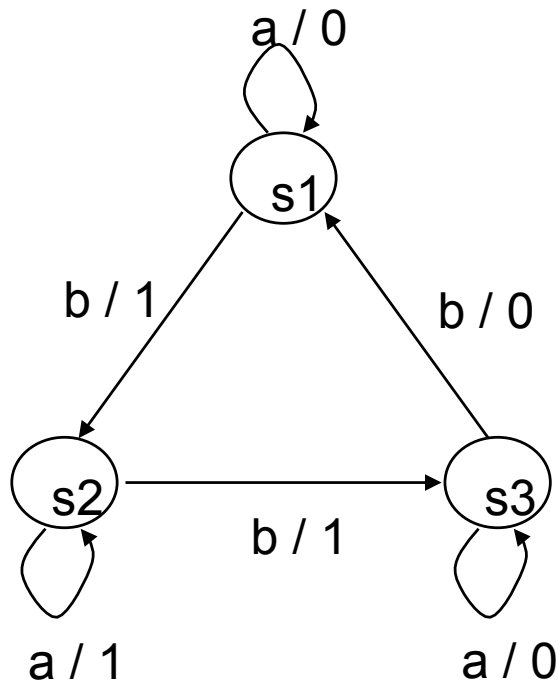


Initial state	Input seq	Output seq	Final state
S1	a	0	S1
S2	a	1	S2
S3	a	0	S3

Initial state	Input seq	Output seq	Final state
S1	b	1	S3
S2	b	1	S2
S3	b	0	S2

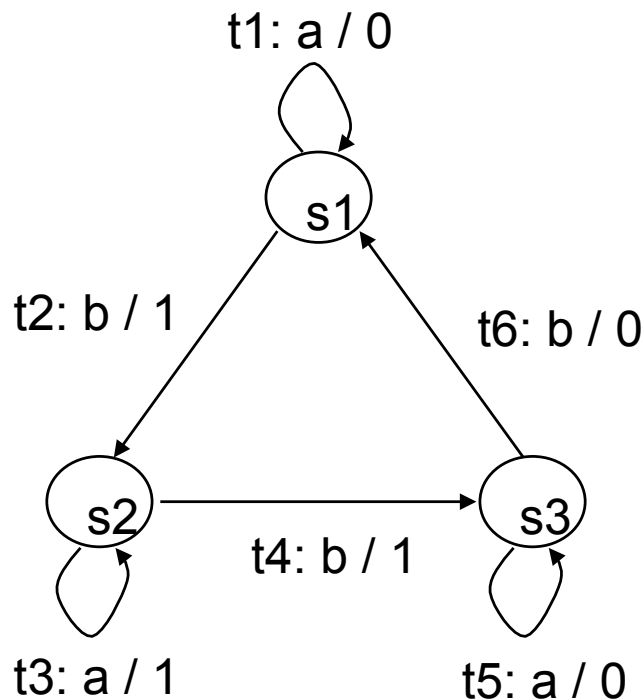
UIO-method and state s_1

- ab is a UIO sequence for s_1



Initial state	Input seq	Output seq	Final state
S1	ab	01	S2
S2	ab	11	S3
S3	ab	00	S1

UIO-method - Test suite



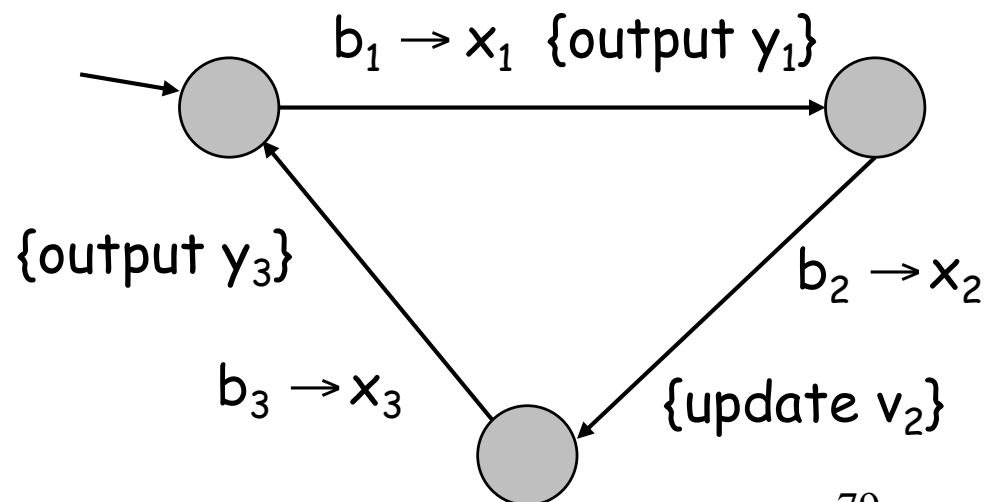
t1: **reset/null** **a/0** a/0 b/1
t2: **reset/null** **b/1** a/1
t3: **reset/null** **b/1** **a/1** a/1
t4: **reset/null** **b/1** **b/1** b/0
t5: **reset/null** **b/1** **b/1** **a/0** b/0
t6: **reset/null** **b/1** **b/1** **b/0** a/0 b/1

UIO-method pros and cons

- Many states in FSMs have UIO sequences
- UIO sequences are usually short
- However, fault detection capability is not powerful (See Mathur's book)

Extended FSMs

- A set of states
- An initial state
- Finite sets of input (X) and output (Y) events
- A finite set of variables (V)
- Transition relation
 - Guards
 - Update blocks



Tests for EFSM models

- EFSM executions depend on input signals and data
- A test consists of:
 - test sequence
 - test data
- Executability problem:
 - Find data to execute the test sequence
 - Undecidable, in general ☹️

EFSM control and data flow testing

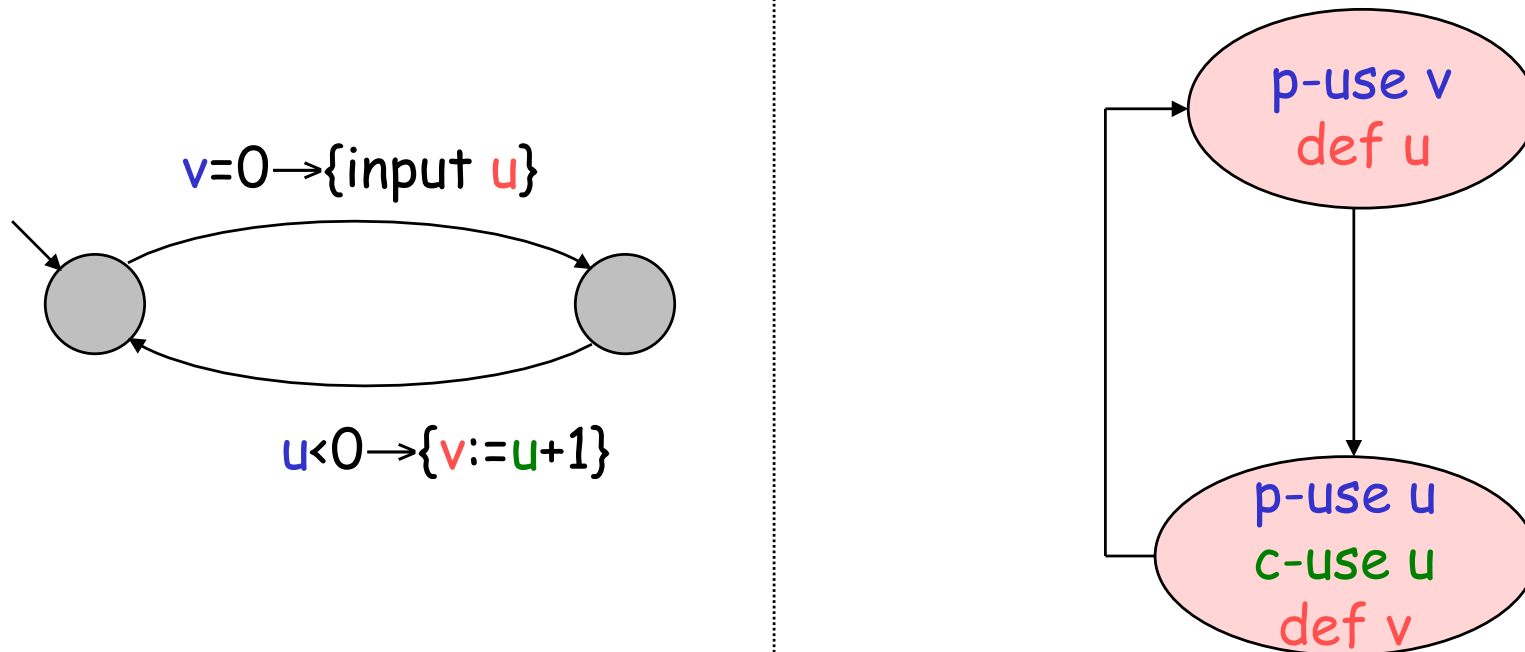
- EFSM executions are data-dependent
- Control flow FSM testing methods (e.g., cover all transitions) are not adequate for EFSM models
- Data flow testing methods account for data dependencies

Data flow: definitions and uses

- Data variables are
 - defined by inputs and updates (def)
 - used in
 - updates (c-use)
 - guards (p-use)
- Data-flow graph captures data dependencies

Data flow graph

- Directed graph with nodes labeled with definitions and uses of variables



Data-flow coverage criteria

- all-def
 - test suite covers each definition at least once
- all-use
 - cover each def-use association at least once
- all-du-paths
 - exercise all paths from each definition of a variable to every one of its uses.

all-use coverage

- Find at least one definition-free path for every def-use association

