Cpt S 422: Software Engineering Principles II

Object-Oriented (Class) testing – Part 2

Dr. Venera Arnaoudova

WASHINGTON STATE UNIVERSITY

Object-Oriented Testing

- ✓ Introduction
- ✓ Accounting for Inheritance
- ❑ Testing Method Sequences
  - ➢ Data slices
  - ➢ Methods' preconditions and postconditions
- ❑ State-based Testing

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 to the output that is expected
- Testing classes for all possible sequences is not usually possible
  - 10 methods ➔ 10! = 3,628,800 sequences ….
- 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`)
- Functionality: 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();
};
```

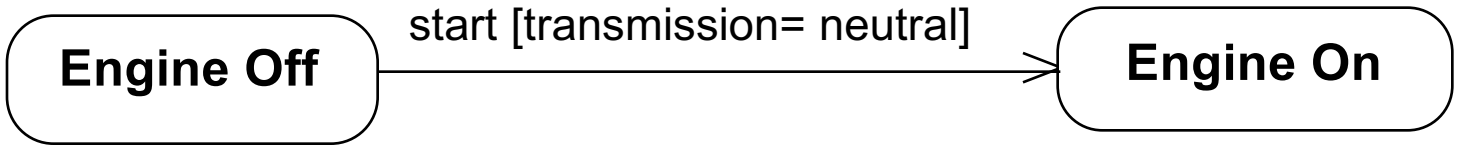
```
void CCoinBox ::AddQtr()
{
   curQtrs = curQtrs + 1;
   if (curQtrs > 1)
        allowVend = 1;
}

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

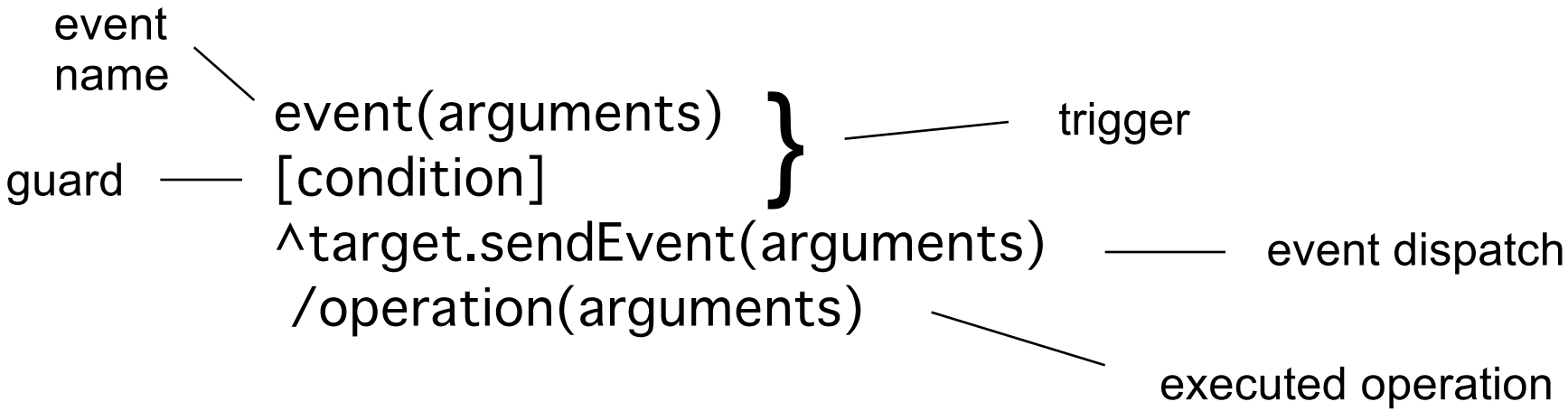
UML Statechart Transitions

Reminder

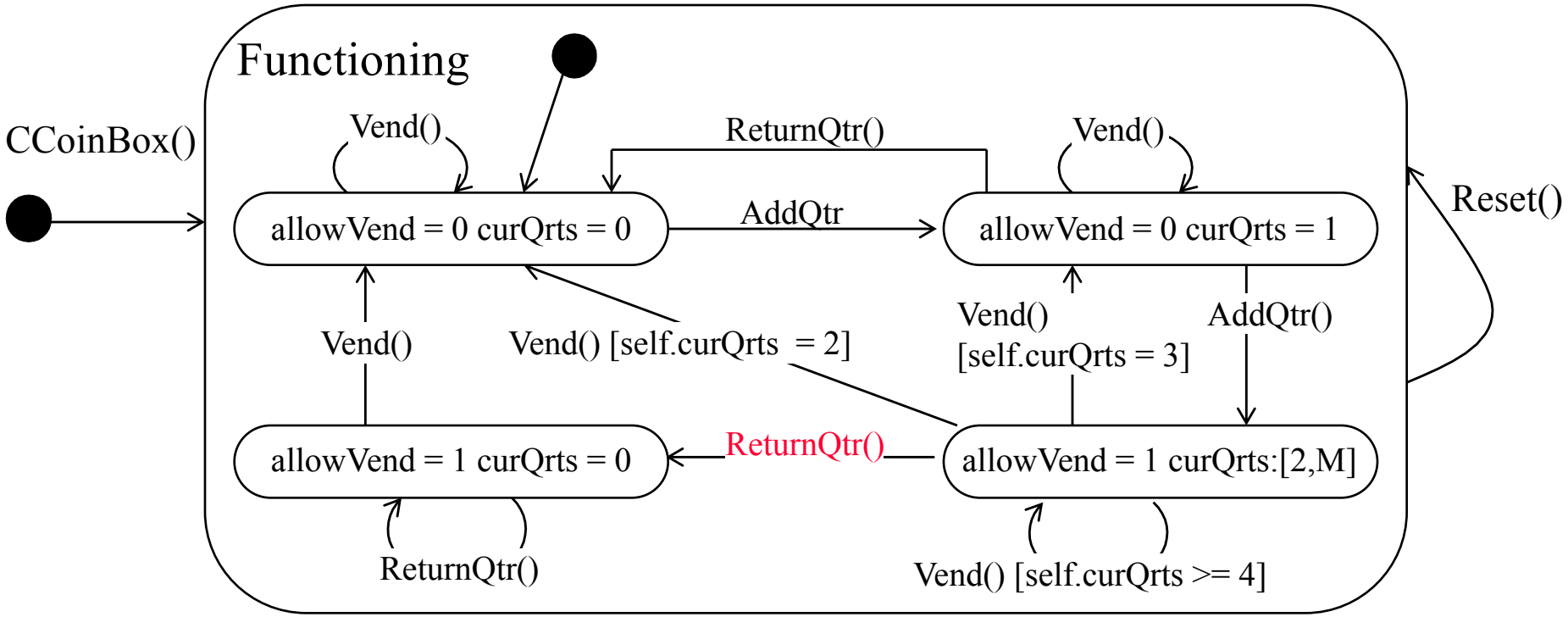
- state transitions caused by events, enabled by guard conditions



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



UML Statechart for CCoinBox



- The statechart is not fully specified to avoid cluttering the diagram
- The state diagram indicates that the scenario:

  `AddQtrs() AddQtrs() ReturnQtrs() Vend()`

  is possible! (=> FREE DRINKS!)

Detecting the Fault

- This is a ***state dependent*** fault, in the sense that executing some operations turn out not to be possible in certain states where they are legal or, alternatively that some operation sequences which are theoretically not possible are executed

- Method testing alone is not enough
  - Structural testing would conclude that, e.g., each edge has been executed
  - The omission may remain unforeseen even with functional testing

- ***The failure will result from involving a certain sequence of operations***

Object-Oriented Testing

- ✓ Introduction
- ✓ Accounting for Inheritance
- ✓ Testing Method Sequences
  - ➢ Data slices
  - ➢ Methods' preconditions and postconditions
- ❑ State-based Testing

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., the class partial correctness
- Repeat for all slices to increase the confidence of the class correctness

Slice Formalism

- A class K encapsulates a set of data members and provides a set of methods (operations)
  - $K = <D(K), M(K)>$
  - $D(K) = \{d_i \mid d_i @ K\}$
  - $M(K) = \{m_i \mid m_i @ K\}$

  *(where **@** denotes the relationship between a class and its elements i.e., is in, **belongs to** ….)*

- A Data Slice
  - $Slice_{di}(K) =< d_i, M_{di}(K) >$
    - $d_i \in D(K)$
    - $M_{di}(K) = \{m_i \mid m_i @@ d_i\}$

      *(where **@@** denotes the **usage** relationship)*

Issues

- What if the code is faulty and a methods 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?

  - We do not have to use "a method" only once in a sequence!

Example C++ (I)

```
class SampleStatistic {
  //test driver:
  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(cont char * msg);
};
```

Part of the GNU C++ library

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
- Account for *indirect* use of data members, through intermediate member functions
- *Different usages*: reads, reports, transforms
- Use the MaDUM to devise a testing strategy

Categorize Member Functions

- Categories: Constructors, Reporters, Transformers, Others
- $M_{di}(K) = \{R_{di}, C, T_{di}, O_{di}\}$,
  - $C = \{c_i \mid c_i \text{ is a constructor of class K}\}$
  - $R_{di} = \{r_{di} \mid r_{di} \text{ is a reporter method for data member } d_i\}$
  - $T_{di} = \{m_{di} \mid m_{di} \notin R_{di} \text{ and } m_{di} \notin C \text{ and } m_{di} \rightarrow d_i\}$
  - $O_{di} = \{o_{di} \mid o_{di} @@ d_i \text{ and } o_{di} \notin R_{di} \text{ and } o_{di} \notin C \text{ and } o_{di} \notin T_{di}\}$
- Others: printError(), destructors …

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));
}
```

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.

- Bashir and Goel took a different approach
  - Generate random sequences of methods for a data member and append a reporter at the end of the sequence
  - Verify that the value reported by the reporter is the same when we directly access the data member

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 the 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`. Since all it does is calling `reset`, we may decide not to test it.

Test the Transformers and Others

- For each slice d_i

  1. Instantiate the object under test with a constructor (already tested)
  2. Create sequences, e.g., all legal permutations of methods in T_{di}
     - ✓ Maximum # sequences: $|C| * |T_{di} \cup O_{di}|!$
     - ✓ For example, if 3 member functions in T_{di} and 4 member functions in O_{di}, with one constructor, this leads to 5040 possible permutations!
  3. Append the reporter method(s) (already tested)

- If several paths in member function: All the paths where the slice d_i is being manipulated must be executed

Test everything else

- Some functions may not use the data members
- They do not change the state of the object
- They neither report the state of any data member nor transforms the state in anyway
- They should be tested as any other functions
- Only their functionality as *stand-alone entities* needs to be checked
- Any standard test technique can be used

CCoinBox Slices

Will not be here !

- Slice allowVend:
  - $T_{allowVend}$(CCoinBox) = {Reset, AddQrt, ReturnQrts,Vend}
- Slice curQrts:
  - $T_{curQrts}$(CCoinBox) = {Reset, AddQrt, ReturnQrts,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 in all cases (see CCoinBox example)
- # sequences may still be large
- Many sequences may be impossible (illegal)
- Automation? , e.g., Oracle, impossible sequences
- 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, the subclass is also called *derived* class because it is derived from the superclass (*base* class)

- The derived class may add functionality or modify the functionality provided by the *base* class – it inherits data members and methods of 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 $MaDUM_{derived}$

  - 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();                  //re-defined
        virtual void operator+=(double); //re-defined
        int similarSamples(double);
        int buckets();
        double bucketThreshold(int i);
        int inBucket(int i);
        void printBuckets(ostream&);
}
```

Example: SampleHistogram (cont.)

```
void SampleHistogram::reset(){
   this->SampleStatistic::reset();
   if(howManyBuckets>0){
        for(int i=0; i<howManyBuckets; i++){
               bucketCount[i]=0;
        }
   }
}
void SampleHistogram::operator+=(double value){
   int i;
   for (i=0; i<howManyBuckets; i++){
        if(value<bucketLimit[i] break;
   }
   bucketCount[i]++;
   this->SampleStatistic::operator+=(value);
}
```

MaDUM SampleHistogram

	Samp-Stat	reset	+ =	sam-ples	mean	stdDev	var	min	max	conf. (int)	conf. (dbl)	error	Samp-Histo	reset	+ =	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

Filling the $MaDUM_{Derived}$

- 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 $m_{derived}$

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

- 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 $m1_{base}$ calls another member function $m2_{base}$ and the method $m2_{derived}$ is redefined in one of the derived classes

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

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.
  - 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
- For SampleHistogram, the set of inherited data members that needs re-testing includes all inherited data members {n, x, x2, minValue, maxValue}

Object-Oriented Testing

- ✓ Introduction
- ✓ Accounting for Inheritance
- ✓ Testing Method Sequences
  - ✓ Data slices
  - ➢ Methods' preconditions and postconditions
- ❑ State-based Testing

Approach

1. Identify pre- and post-conditions
   - e.g., from UML 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

3. Choose a criterion
   - We will define 7 criteria

4. Derive method sequences satisfying the criterion from the method sequence constraints

Sequencing Constraints

- 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=True` $\Rightarrow$ `m2` can always be executed after `m1`, i.e., `m1`'s post-condition implies `m2`'s pre-condition
  - `C=False` $\Rightarrow$ `m2` can never be executed after `m1`, i.e., `m1`'s post-condition implies the negation of `m2`'s pre-condition
  - `C=BoolExp` $\Rightarrow$ `m2` can be executed after `m1` under some conditions. `BoolExp` (disjunctive normal form): $C=C_1 \vee C_2 \vee \ldots$

Criteria (I)

- Always Valid (T)
  - Each always-valid constraint must be covered at least once
  - i.e., each `(m1,m2,T)`.

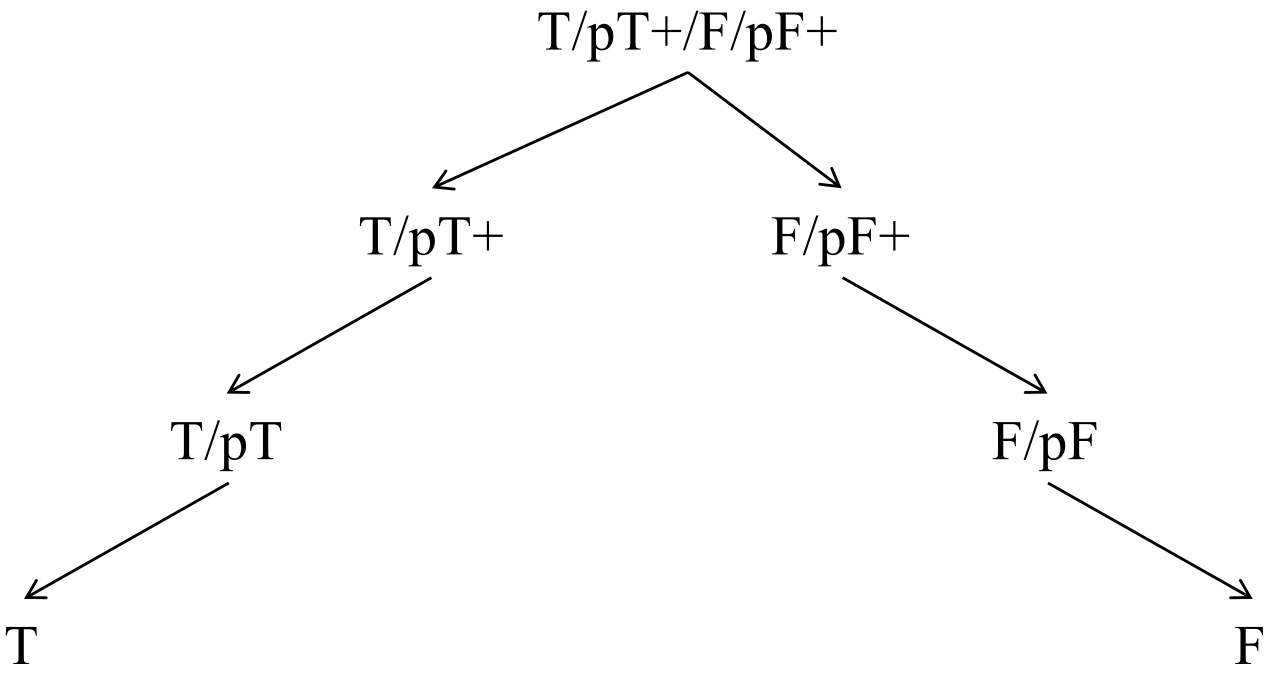
- Always/Possibly True (T/pT)
  - Each always-valid constraint and each possibly-true constraint must be covered at least once.
  - i.e., each `(m1,m2,T)` **and** each `(m1,m2,C)` using *one* of the disjunction in `C` (C_1 or C_2 or …)

- Always/Possibly True Plus (T/pT+)
  - Each always-valid constraint **and** each possibly-true constraint using 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 and C_2 and …)

Criteria (II)

- Never Valid (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 using one of the disjunction false constraint must be covered at least once.
  - i.e., each `(m1,m2,F)` **and** each `(m1,m2,not(C))`
- Never Valid/Possibly False Plus (F/pF+)
  - Each never-valid constraint and each possibly false constraint must be covered at least once.
  - $\texttt{not C} = C'_1 \vee C'_2 \vee \ldots$
  - i.e., each `(m1,m2,F)` and each `(m1,m2,not(`C'_1`))`, `(m1,m2,not(`C'_2`))` …
- Always/Possibly True Plus/Never/Possibly False Plus (T/pT+/F/pF+)

Subsumption



Example – Queue Abstract Data Type

Class Queue (unbounded queue)

count // Number of elements in the queue

Init(q:Queue)

pre: Queue q does not exist

post: Queue q exists and is empty

Eque(q:Queue, e:Element)

pre: Queue q exists

post: Element e is 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 is 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)

Example – Queue (cont.)

Sequencing constraints:

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

Where: C = count>0

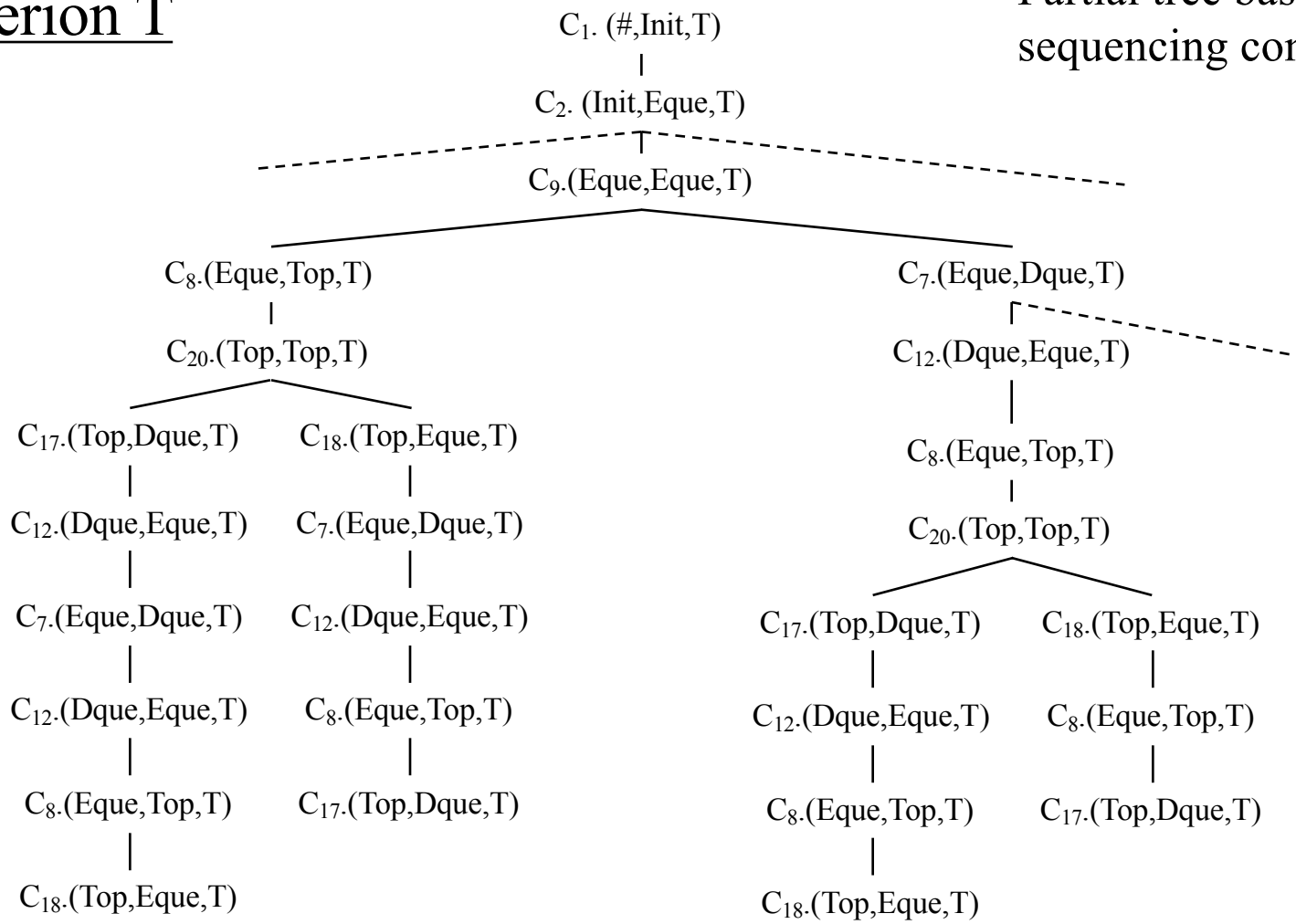
Criterion Always Valid (T) requires the use of:

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

Example – Queue (cont.)

Criterion T

Partial tree based on sequencing constraints



Discussion

- 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
  - May need to cover some T constraints to cover certain F ones

Empirical Study

- From Daniels and Tai
- 3 C++ programs
- Mutation tool Proteum for C
- 71 mutation operators
- Most mutants were killed with the Always Valid (T) criterion and all of them with the Always/Possibly True (T/pT) criterion

Object-Oriented Class Testing

- ✓ Introduction
- ✓ Accounting for Inheritance
- ✓ Testing Method Sequences
- ❑ State-Based Testing

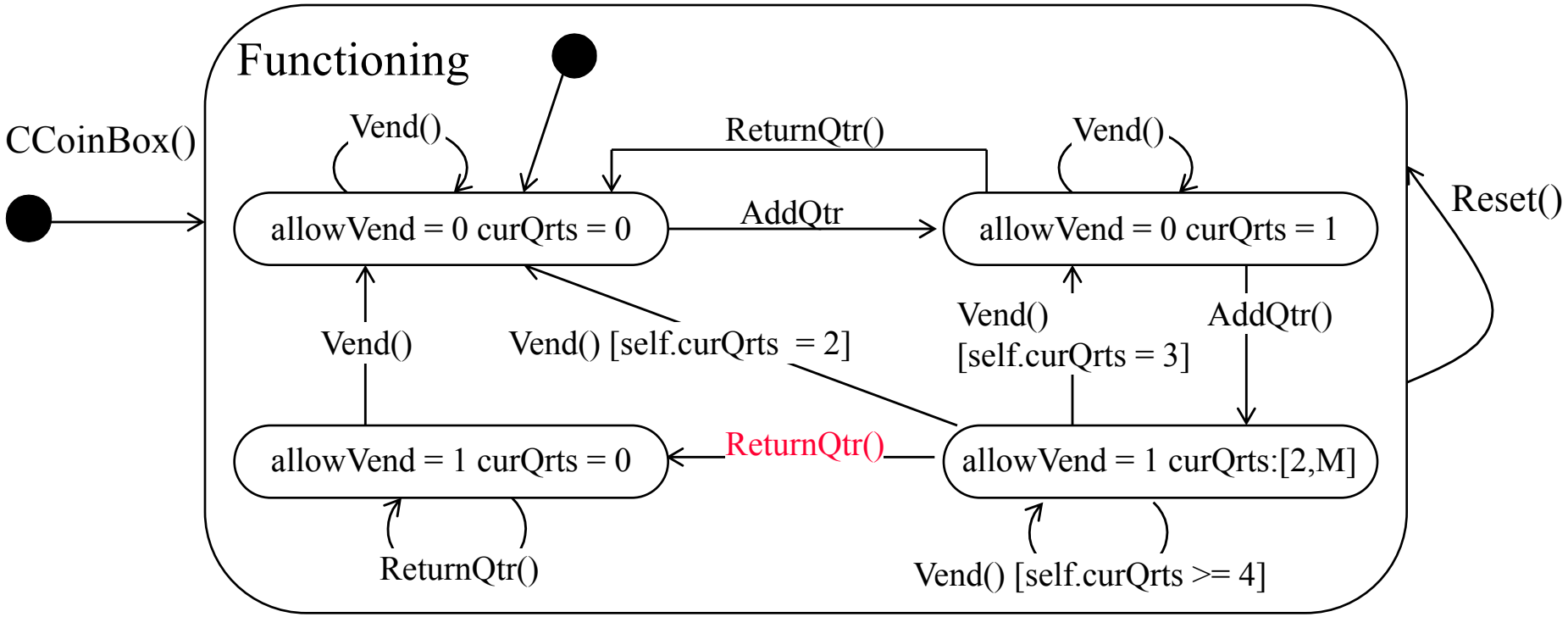
Chow's Methods for State Model Testing

- One of the earliest papers on the topic is Chow's paper (1978)
- The first step is to generate a ***transition*** *or* ***test tree*** from the statechart
- The tree paths includes all *round-trip* state-transition paths (as defined by Binder): transition sequences that begin and end with the same state (with no repetitions of state other than the sequence start/end state) and *simple paths* from the initial to the final state of the state model
  - A *path* here is a *sequence* of transitions: $state_p$, $event_i$, $state_q$, $event_j$, $state_r$, …
  - A *simple path* contains no loop iteration
- Append each sequence with the characterization set (W) or a call to a 'status' / get_state method (assertion)

Procedure for Deriving Tree

- Initial state as the root node of the tree
- An edge is drawn for every transition out of the initial node, with nodes being added as resultant states
- A leaf node is marked as **terminal** if the state it represents has already been drawn or is the final state
- No more transition are traced out of a terminal node
- This procedure is repeated until all leaf nodes are terminal
- The tree structure depends on the order in which transitions are traced (breadth or depth first)
- A depth first search yield fewer, longer test sequences
- The order in which states are investigated is supposed to be irrelevant

UML Statechart for CCoinBox (faulty)



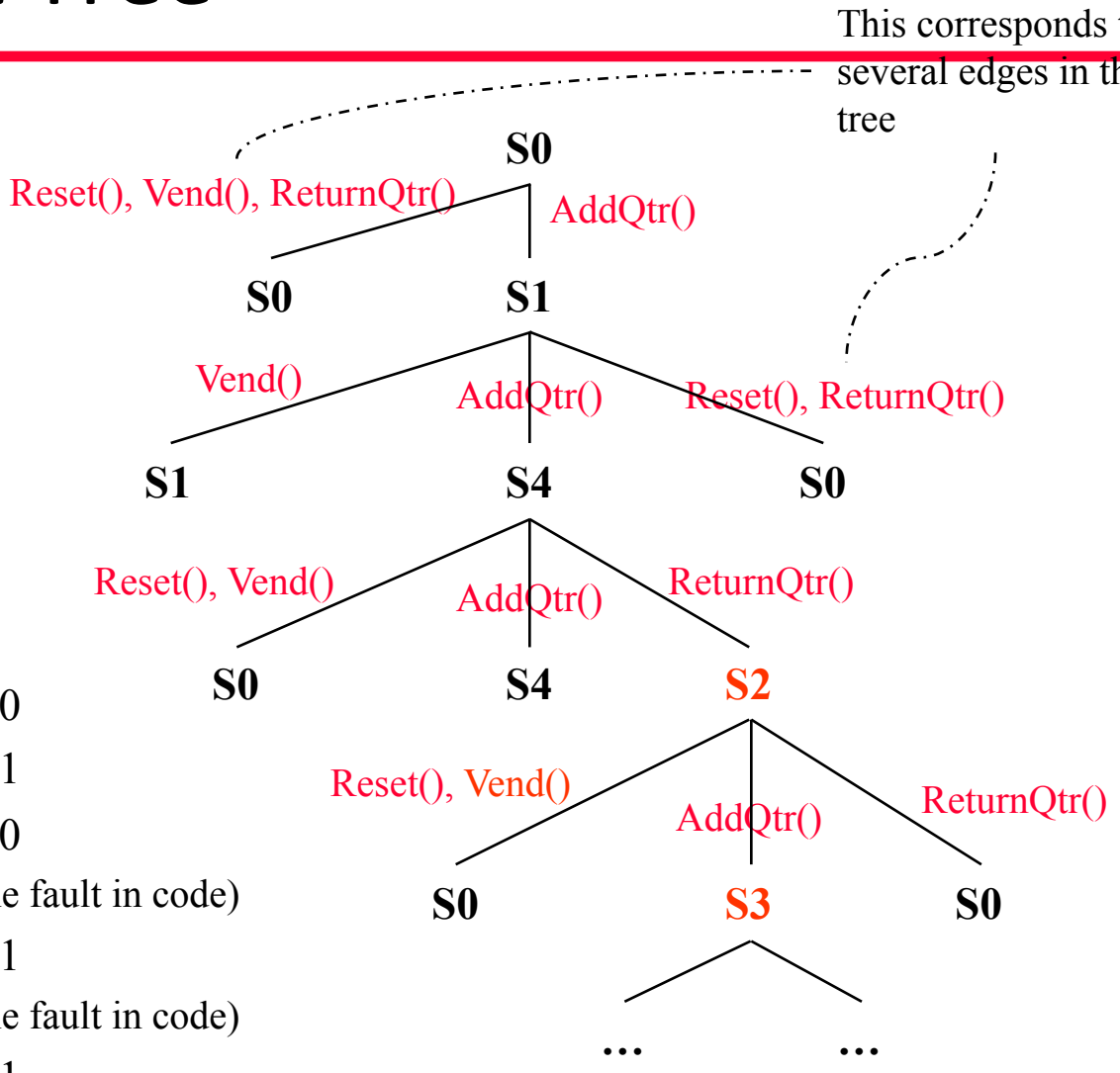
- The statechart is not fully specified to avoid cluttering the diagram
- The state diagram indicates that the scenario:

  `AddQtrs() AddQtrs() ReturnQtrs() Vend()`

  is possible! (=> FREE DRINKS!)

CCoinBox Test Tree

- **Based on the (faulty) statechart presented earlier**
- Root node = initial state
- S3 was not in the statechart (missing corrupt state) and is not a terminal node



- S0: allowVend = 0, curQtrs = 0
- S1: allowVend = 0, curQtrs = 1
- S2: allowVend = 1, curQtrs = 0
  (illegal state made possible by the fault in code)
- S3: allowVend = 1, curQtrs = 1
  (illegal state made possible by the fault in code)
- S4: allowVend = 1, curQtrs > 1

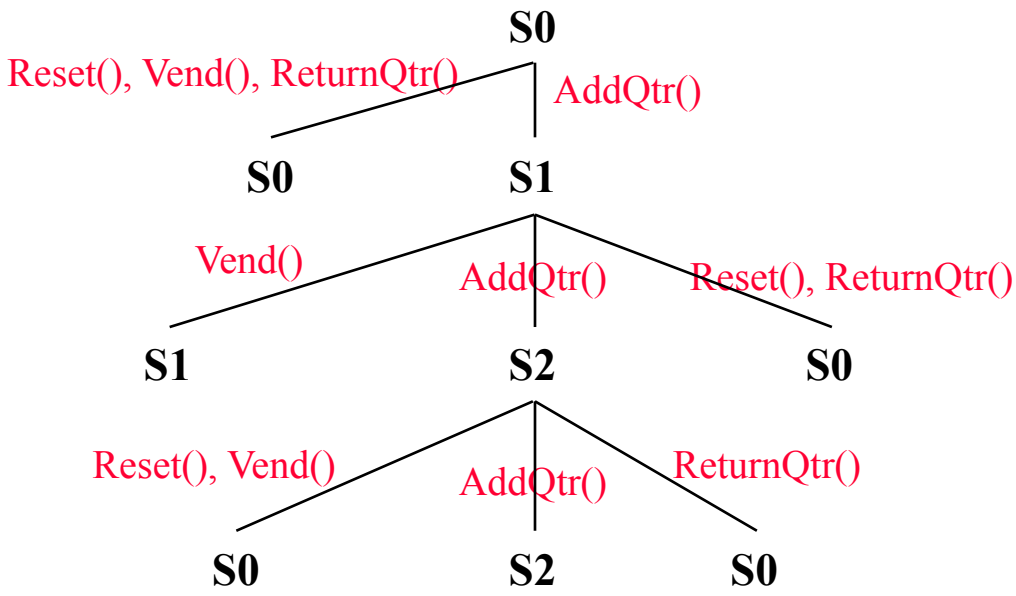
CCoinBox Test Tree (correct)

Based on the correct statechart, modeling how the code should behave

S0: allowVend = 0, curQtrs = 0

S1: allowVend = 0, curQtrs = 1

S2: allowVend = 1, curQtrs > 1



Using this transition tree, with the faulty `CCoinBox` program, `ReturnQtr()` on S2 would not lead to S0 but to a corrupt state – however it would not be observable unless we have a way to directly access the state of `CCoinBox`, or if we attempt to vend.

From Test Tree to Test Cases

- Each test case begins at the root node and ends at a leaf node
- The expected result (Oracle) is the sequence of *states* and *actions* (outputs, other objects' change of state)
- Test cases are completed by identifying *method parameter values* and *required conditions* to traverse a path
- We run the test cases by setting the object under test to the *initial state*, applying the sequence, and then checking the *intermediate* states, *final* state and outputs (e.g., logged)

Guard Conditions

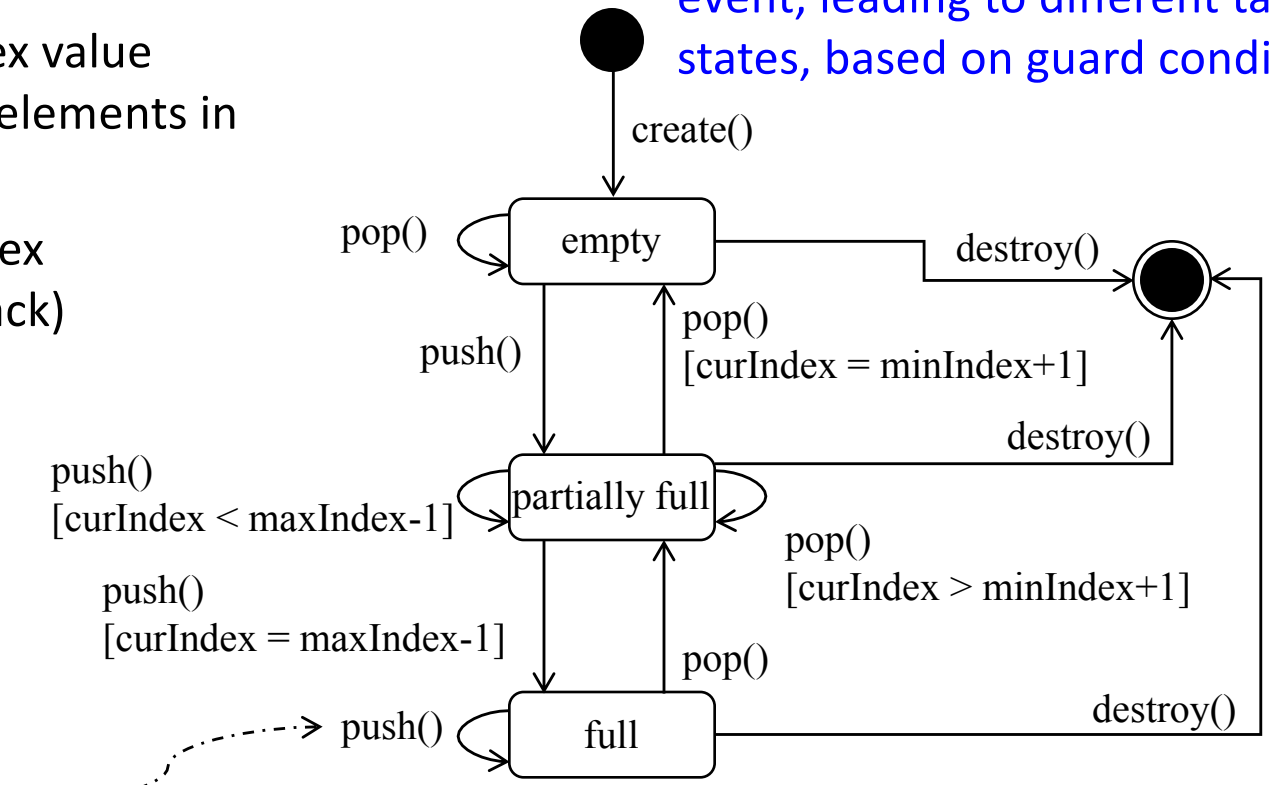
- When the guard is a simple Boolean expression or contains only logical `and` operators: one *true* combination
- When the guard is a compound Boolean expression containing at least one logical `or` operator. One transition is required for each *true* combination
- When the guard specifies a relationship that occurs only after repeating some event several times: single arc annotated with * for the transition
- At least one false combination in all cases

BoundedStack Example

Assume that three data members are defined in the class:

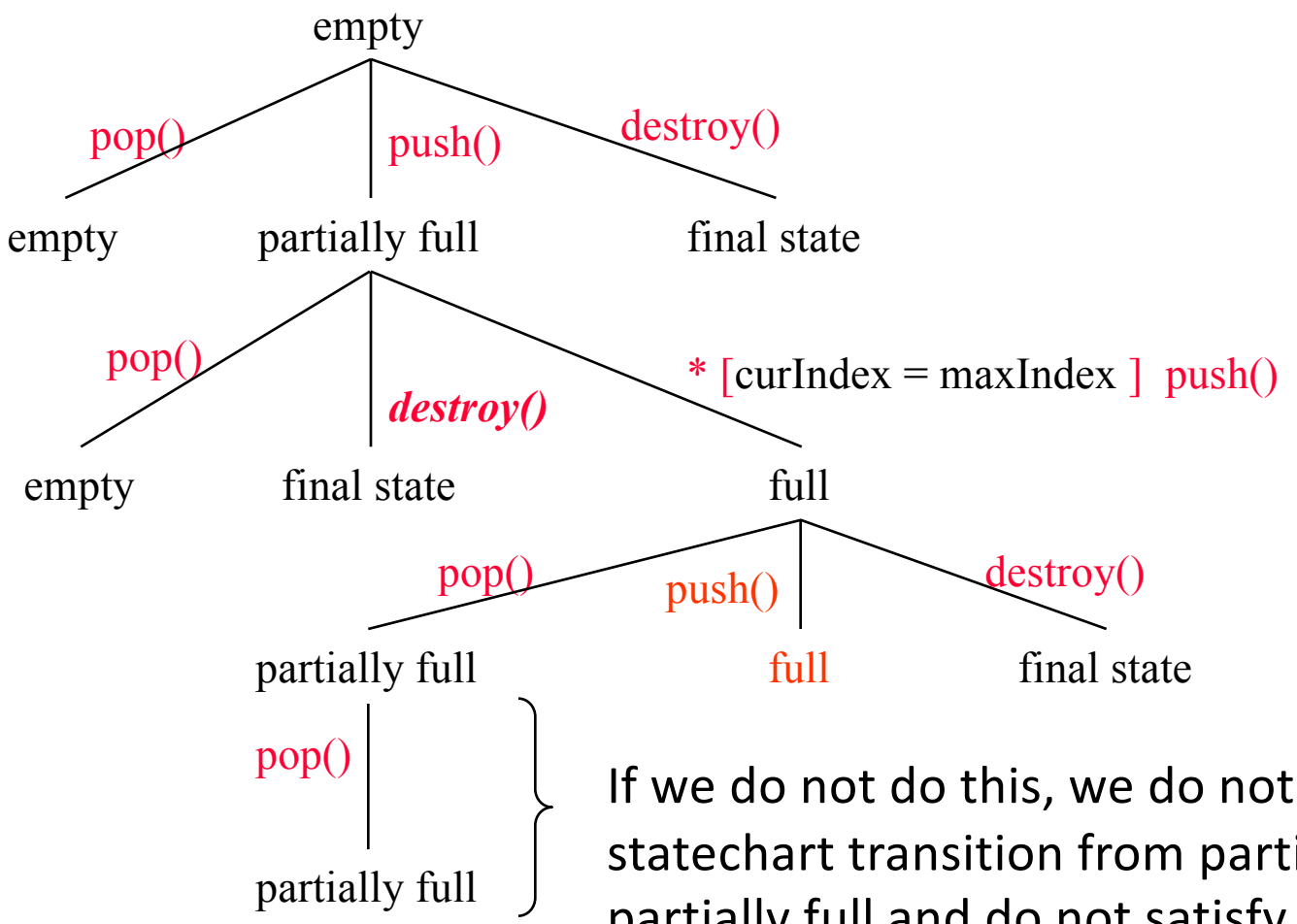
- curIndex: current index of last element introduced in the stack
- minIndex: minimum index value (e.g., 0 minimum size of elements in stack)
- maxIndex: maximum index (maximum size of the stack)

How do we handle multiple transitions from the same source state, with the same event, leading to different target states, based on guard conditions?



Assume, for example, that there is a bug in the way `push()` handles a full stack

BoundedStack Transition Tree



If we do not do this, we do not cover the statechart transition from partially full to partially full and do not satisfy All Transition coverage!

BoundedStack Test Driver

```
int boundedStack_test_driver() {
   BoundedStack stack(2);
   stack.push(3);    // push() when empty
   stack.push(1);    // push() when partially-full
   try {Stack.push(9);} // push() when full
   catch (Overflow ex) {} // expected to throw
   stack.pop();        // pop() when full
   stack.pop(); // pop() when partially-full
   try {stack.pop();}       // pop() when empty
   catch (Underflow ex) {} // expected to throw
   BoundedStack stack2(3);
   stack2.push(6);   // stack2 is partially-full
   stack2.push(5);   // stack2 is still partially-full
   BoundedStack stack3(1);
   stack3.push(6);          // stack3 is full
   // destructors called implicitly at end of block for
   // stack (empty), stack2 (partially-full) and stack3
   (full)
};
```