Testing Object-Oriented Software

Class Testing

Statecharts

- EFSMs (FSMs + variables) + concurrency + hierarchy + communication + real-time
- Plus special features, e.g., history states, pseudo states
- Widely used for specifying real-time embedded HW/SW controllers
- Also used in most of object-oriented methodologies, e.g., UML
- FSMs and EFSMs are in practice not often used. Statecharts are.

A coffee vending machine

Statecharts =

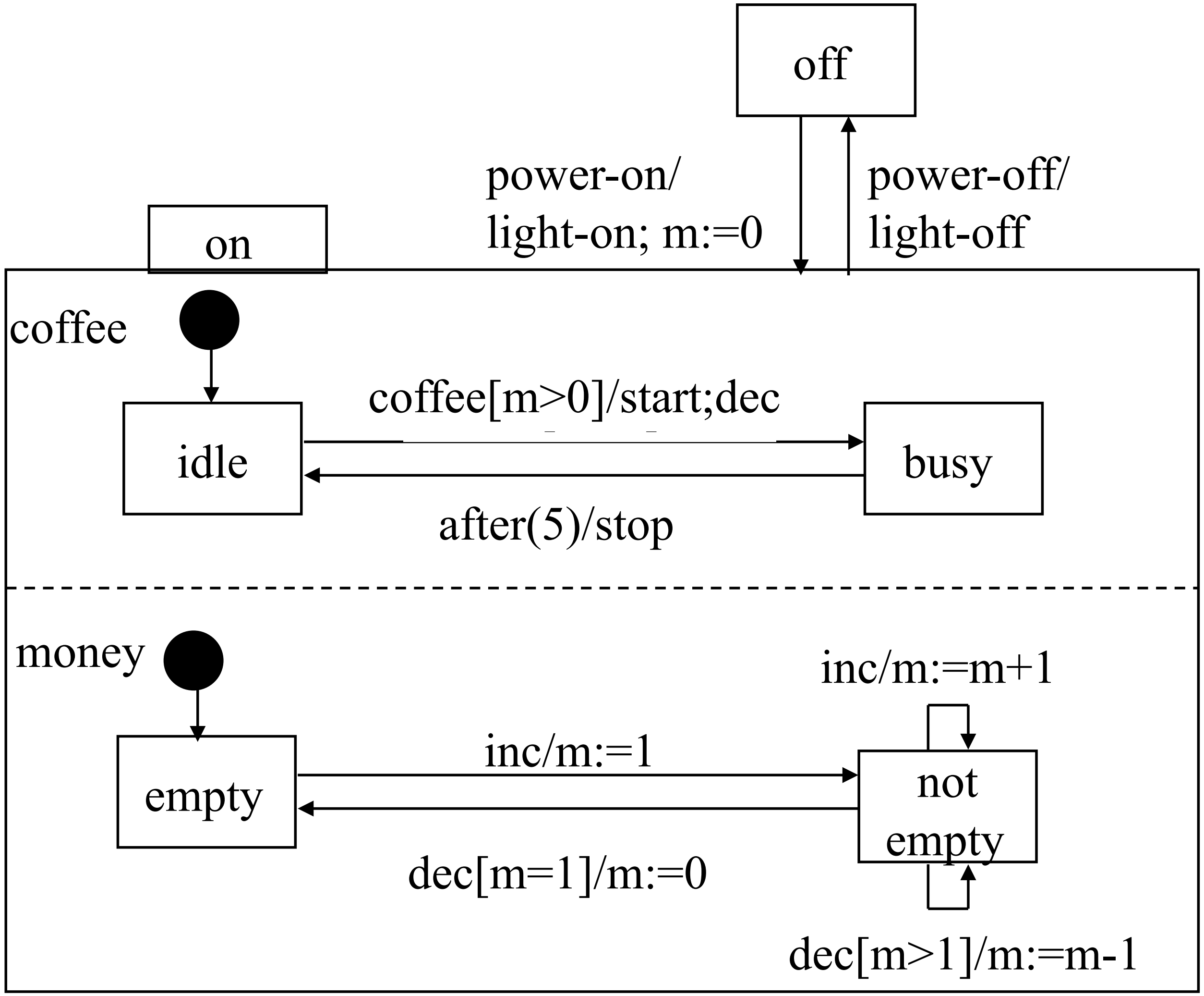
EFSMs +

Hierarchy +

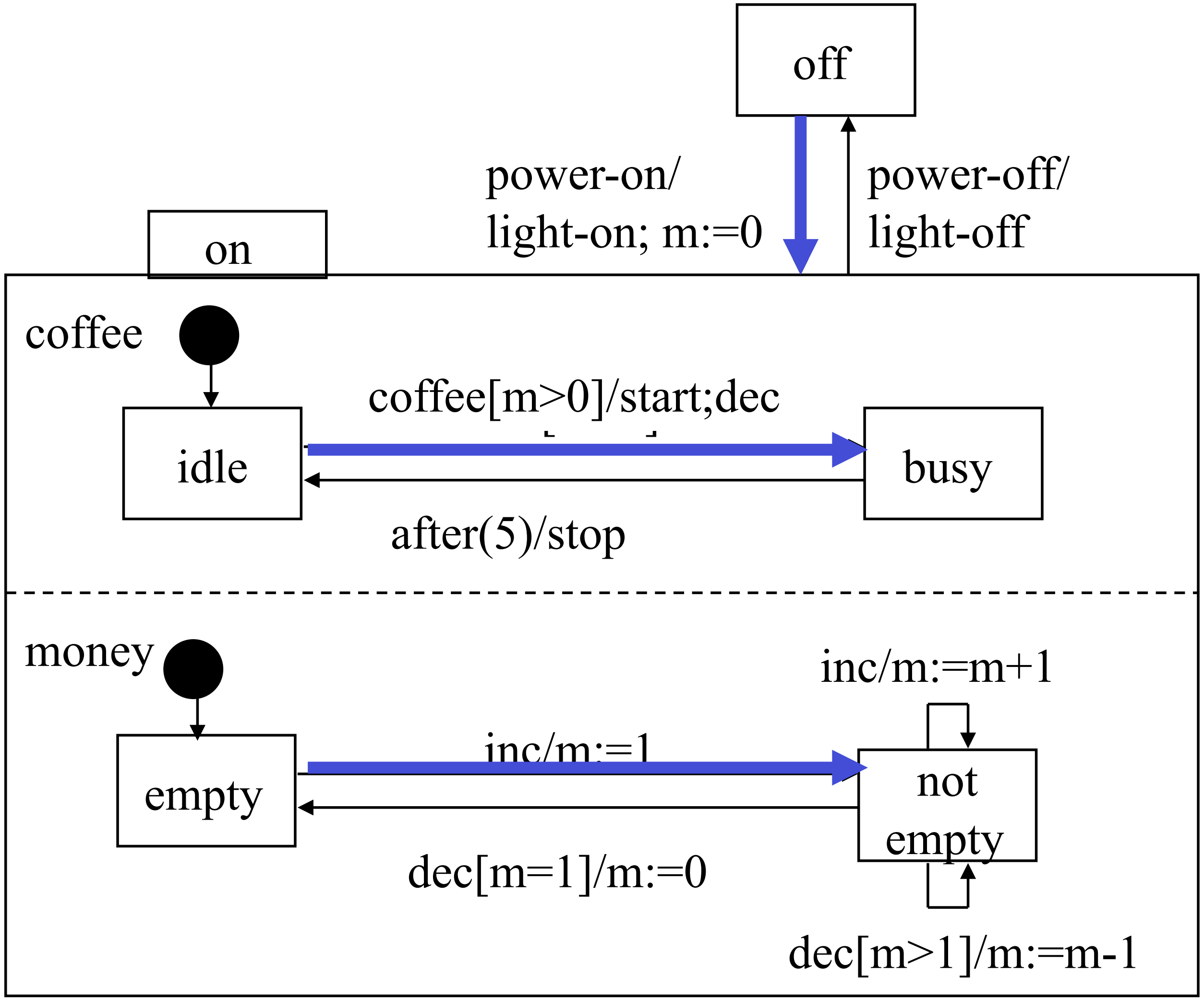
Concurrency +

Communication +

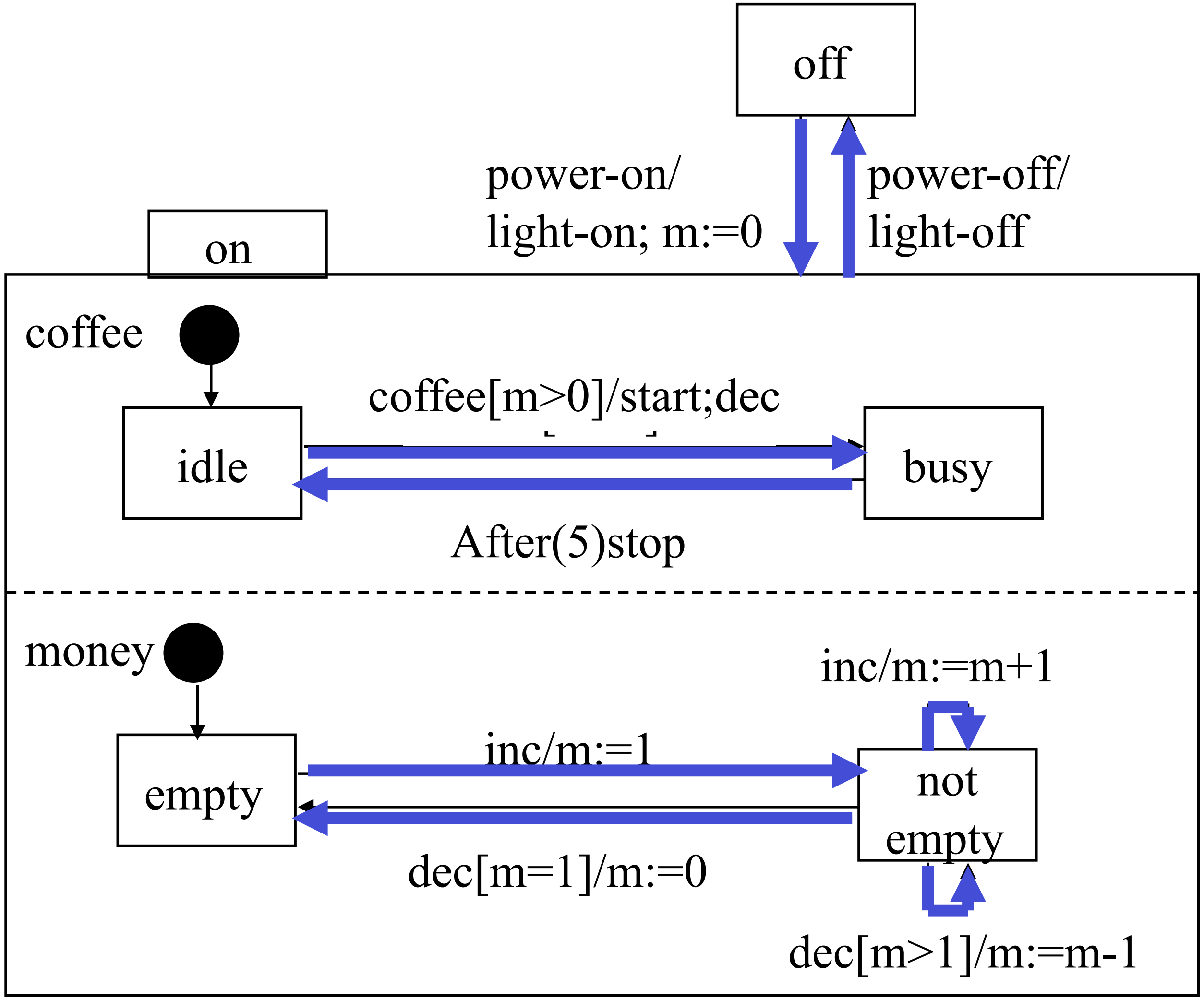
Real-time



State coverage



Transition coverage



Guard Conditions and Transition Pairs (Offutt et al., 1999)

- Offut et al, Criteria for Generating Specifications-based Tests, proceedings of UML'99, 1999, Springer
- *Transition Coverage:* The test suite T causes every transition in the state model to be taken at least once.
- *Full Predicate Coverage:* It is trying to determine whether each clause in a transition predicate (guard condition) is necessary and formulated correctly. For each predicate P on each transition, T must include tests that cause each clause c in P to result in a pair of outcomes (true, false) where the value of P is directly correlated with the value of c.
- *Transition-Pair Coverage:* For each pair of adjacent transitions Si:Sj and Sj:Sk, T contains a test that traverses the pair of transitions in sequence. It tries to exercise interactions between pairs of transitions.
- *Complete Sequence coverage:* The test engineer must define meaningful sequences of transitions on the state model diagram by choosing sequences of states that should be entered. Usually impractical or impossible. We will see how to select a subset of paths next.

Full Predicate Coverage

- This is specific to EFSMs and statecharts, as in UML, not FSMs
- Testers should at minimum provide test cases to test each clause in each guard condition (predicate)
- Same as modified condition decision coverage criterion, but for guard conditions in statecharts, not source code
- Rationale: Check if each clause is necessary and is formulated correctly
- *Clause*: Boolean expression that contains no boolean operators (e.g., AND, OR, NOT)
- *Predicate*: Boolean expression that is composed of clauses and zero or more Boolean operators. A clause may appear more than once in a predicate.

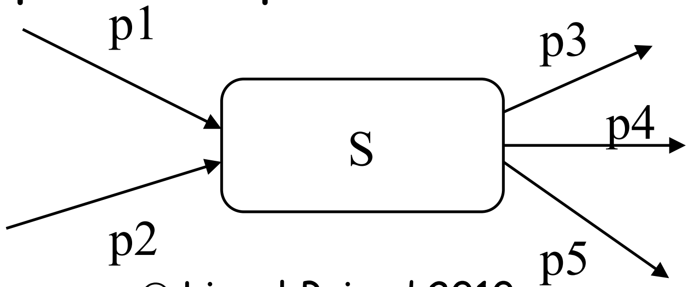
Example

	(A	OR	B)	AND	C
1	**T**		F		T
2	**F**		F		T
3	F		**T**		T
4	F		**F**		T
5	T		T		**T**
6	T		T		**F**

- Significantly increase the cost of transition coverage
- Example: 6 traversals of the transition

Transition-Pair Coverage

- Previous criteria do not test sequences of transitions
- Interactions between transitions (e.g., data flow) should be exercised
- Check for invalid transition sequence allowed or valid sequence not allowed
- Example:
  - Transitions are triggered based on predicates pi
  - Test inputs must satisfy predicate pairs associated with transition pairs
    - to test S, 6 transition pairs are required
    - (p1:p3), (p1:p4), (p1:p5), (p2:p3), (p2:p4), (p2:p5) assuming all pairs are possible …



Complete Sequence Criterion

- Experience and knowledge of the test engineer required
- Select "meaningful sequences" of transitions
- Sequence of state transitions that form a *complete practical use* of the system
- In many cases, the number of possible sequences is too large if not infinite

Empirical Study

- Small cruise control system (400 C lines, 7 functions, 184 blocks, 174 decisions)
- Evaluation Criteria
  - Structural coverage (decision and block coverage)
  - Fault coverage
- Four states: Off, Inactive, Cruise, and Override
- A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," Proc. 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, pp. 416-429, October, 1999.
- A. J. Offutt, Y. Xiong and S. Liu, "Criteria for Generating Specification-based Tests," Proc. 5th International Conference on Engineering of Complex Computer Systems (ICECCS'99), Las Vegas, NV, pp. 119-129, October, 1999.

Empirical Settings

- 25 faults were inserted in separate versions of the program (aka mutation testing)
- Most were in the logic that implemented the state machine
- Compare All transition to Predicate Coverage
- Tests were created independently from the faults, by different people (manually)
- Each test case was executed against each faulty version
- As a comparison, 54 test cases were generated randomly

Program and Fault Coverage

- The FP criterion lead to 89% of blocks (stmt) and 95% of decisions (branches) in the code being covered
- Fault detection:

	Random	Transition	Full Pred
# TC	54	12	54
Faults found	15	15	20
Faults missed	9	9	4
Percent (mutation score)	62.5%	62.5%	83.3%

Transition / Test Trees

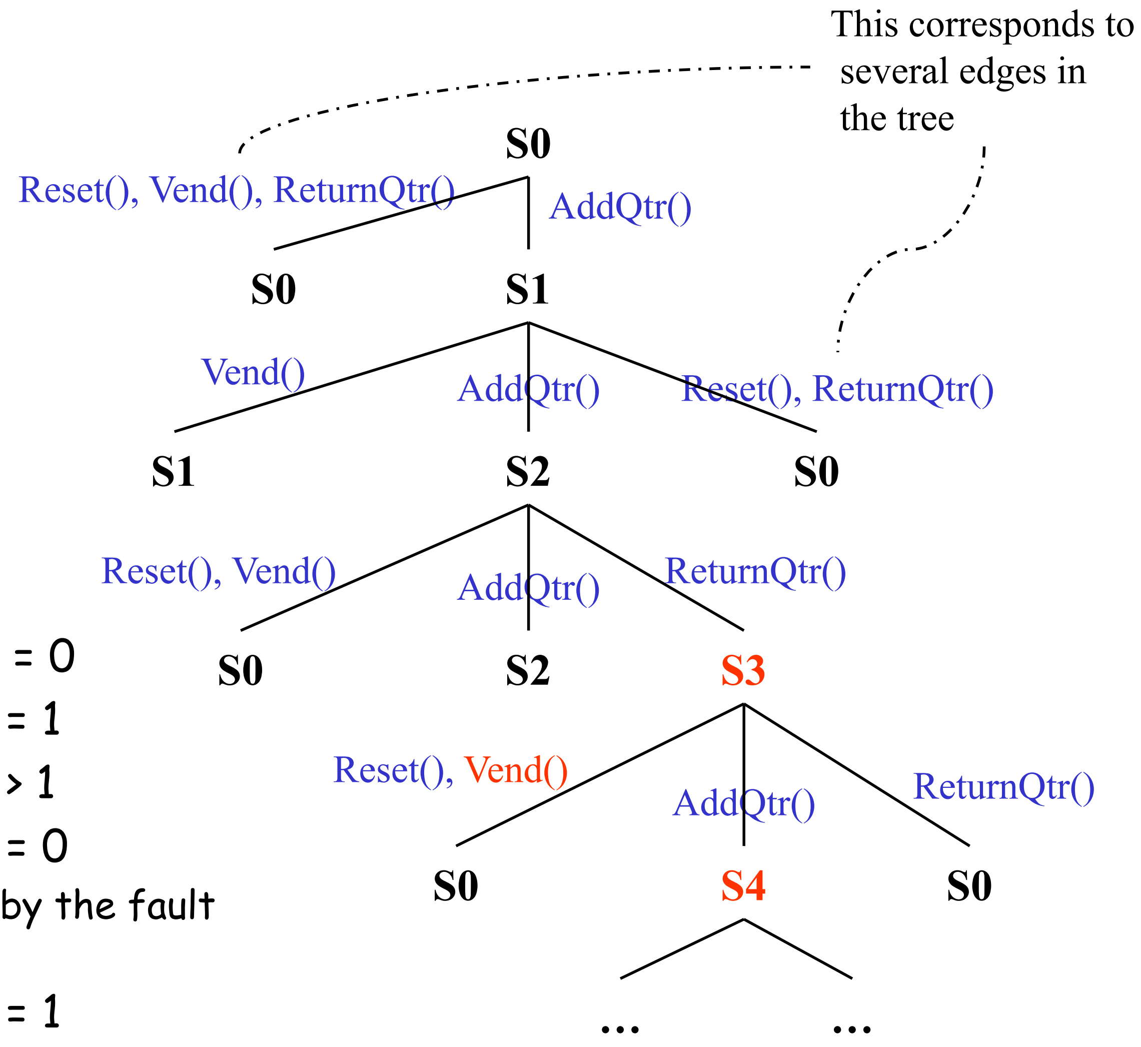
- Offutt defines coverage criteria but does not propose methods to *automate* the determination of state sequences to test
- One of the earliest papers for FSMs is Chow's paper (1978)
- Does not address guard conditions on transitions
- Adapted for (flattened) statecharts by Binder (2000)
- The first step is to generate a *transition or test tree* from the statechart
- The tree paths includes all *round-trip* state-transition paths: 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
  - 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

- Flatten statechart (remove concurrency and hierarchy)
- 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

CCoinBox Test Tree

- **Based on the (faulty) statechart presented earlier**
- Root node = initial state
- S4 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 > 1
- S3: allowVend = 1, curQtrs = 0

  (corrupt state made possible by the fault in code)
- S4: allowVend = 1, curQtrs = 1

  (corrupt state made possible by the fault in code)

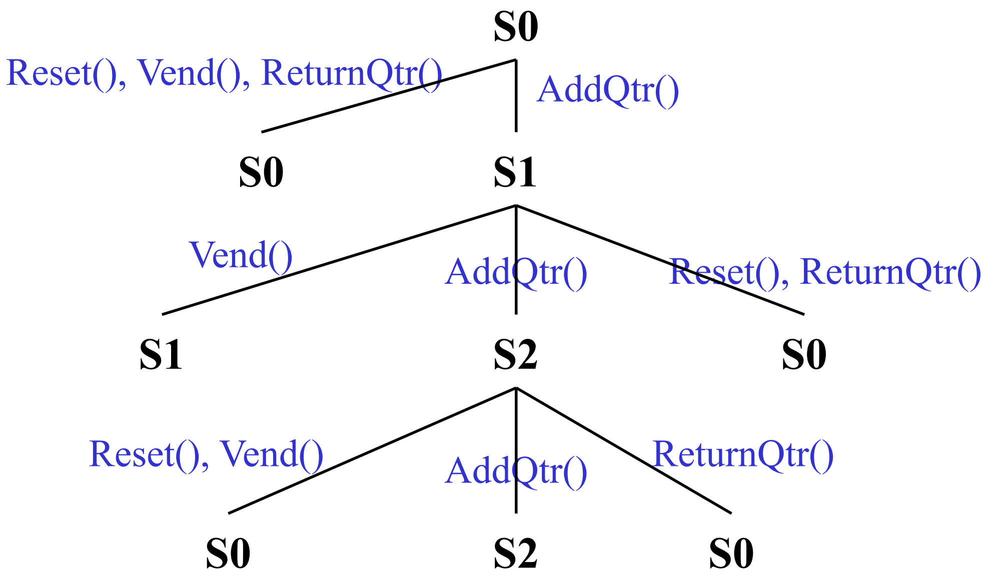
CCoinBox Test Tree (correct)

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 & check the state of `CCoinBox` or use distinguishing/UIO sequences after each leaf (we attempt to vend, then add a quarter and vend).

From Test Tree to Test Cases

- Each test sequence 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) – assuming states can be "observed".
- 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 *intermediary* states, *final* state, and outputs (e.g., logged)

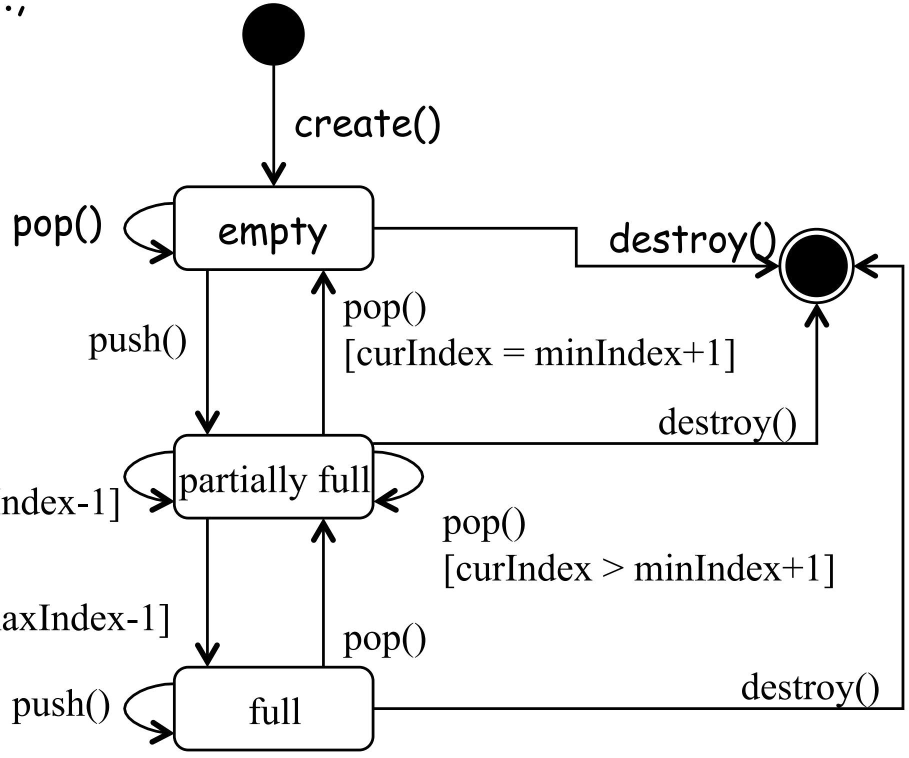
Guard Conditions

- The guard is a simple Boolean expression or contains only logical `and` operators: one *true* combination
- The guard is a compound Boolean expression containing at least one logical `or` operator. One transition is required for each *true* combination.
- 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 (see sneak paths)
- Alternative: use Offutt's predicate coverage, which is more complex – difficult to say which one is more cost-effective

BoundedStack Example

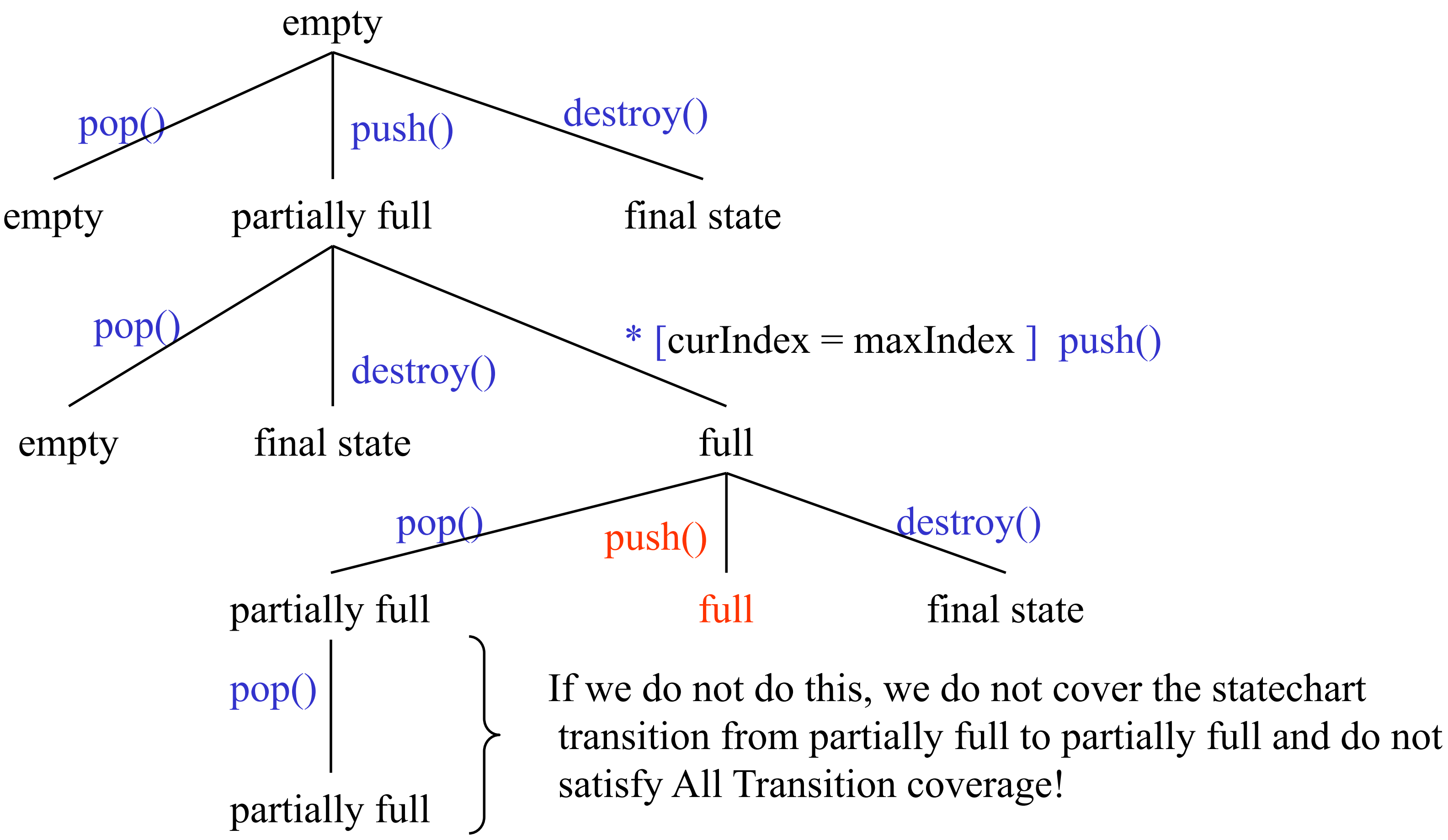
Assume that three data members are defined in the class:

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



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)
};
```

Types of Faults Detected

- Incorrect or missing transitions
- Incorrect or missing actions leading to incorrect output or system state
- Missing states and some corrupt states
- Some extra transitions (sneak paths) cannot be detected unless state model completely specified (unlikely)
- How to improve testability by improving the *observability* of corrupt states: Class invariant check.

Detecting Sneak Paths

- Covering all *round-trip* paths shows *conformance* to the explicitly modeled behavior
- When state machines are incompletely specified (and they usually are), we have to test for *sneak paths*
- They are unexpected transitions
- A sneak path is possible for each unspecified transition and for guarded transitions that evaluate to false
- We need to test all state's illegal events (no need to check for sneak paths traversing two or more states) – this guarantees to reveal all sneak paths
- Detecting sneak paths may be particularly important for safety-critical systems
- One has to check that the appropriate action is taken: exception handling, error message …

Sneak Path Testing Procedure

- Place the object in the corresponding state (possibly using a built-in *set* method)
- Apply illegal event by sending message or forcing the virtual machine to generate the desired event
- Check that the actual response matches the specified response (e.g., raise exception, error message)
- Check that the resultant state is unchanged

Subsumption Hierarchy

- *Piecewise*: All states, all events, all actions
- *All (explicit) transitions*: Every *specified* transition is exercised
  - Maximum #test cases is #events*#states
  - Catch all operation errors, but may not catch all transfer / transition errors
- *All (explicit) n-Transitions Sequences*: Every *specified* transition sequence of *n* events is exercised at least once
- *All Round-Trip Paths*
- *Exhaustive*

=> Overall, despite some theory, very little experience exists with these coverage criteria

Problems with Binder's Approach

- Covering certain transitions requires to traverse specific paths to satisfy the guard condition. We cannot simply follow the test tree algorithm when the statechart contains guard conditions. How do we automate the generation of the tree?
  - Potential solution: Dynamic test generation
- Sometimes, as a result of the above problem, using the tree algorithm does not lead to covering all transitions
- The test tree covers round trip paths in a piecewise manner - it does not execute the round trip paths per se. Therefore, we cannot say that the round trip path technique subsumes the n-transitions sequence criterion

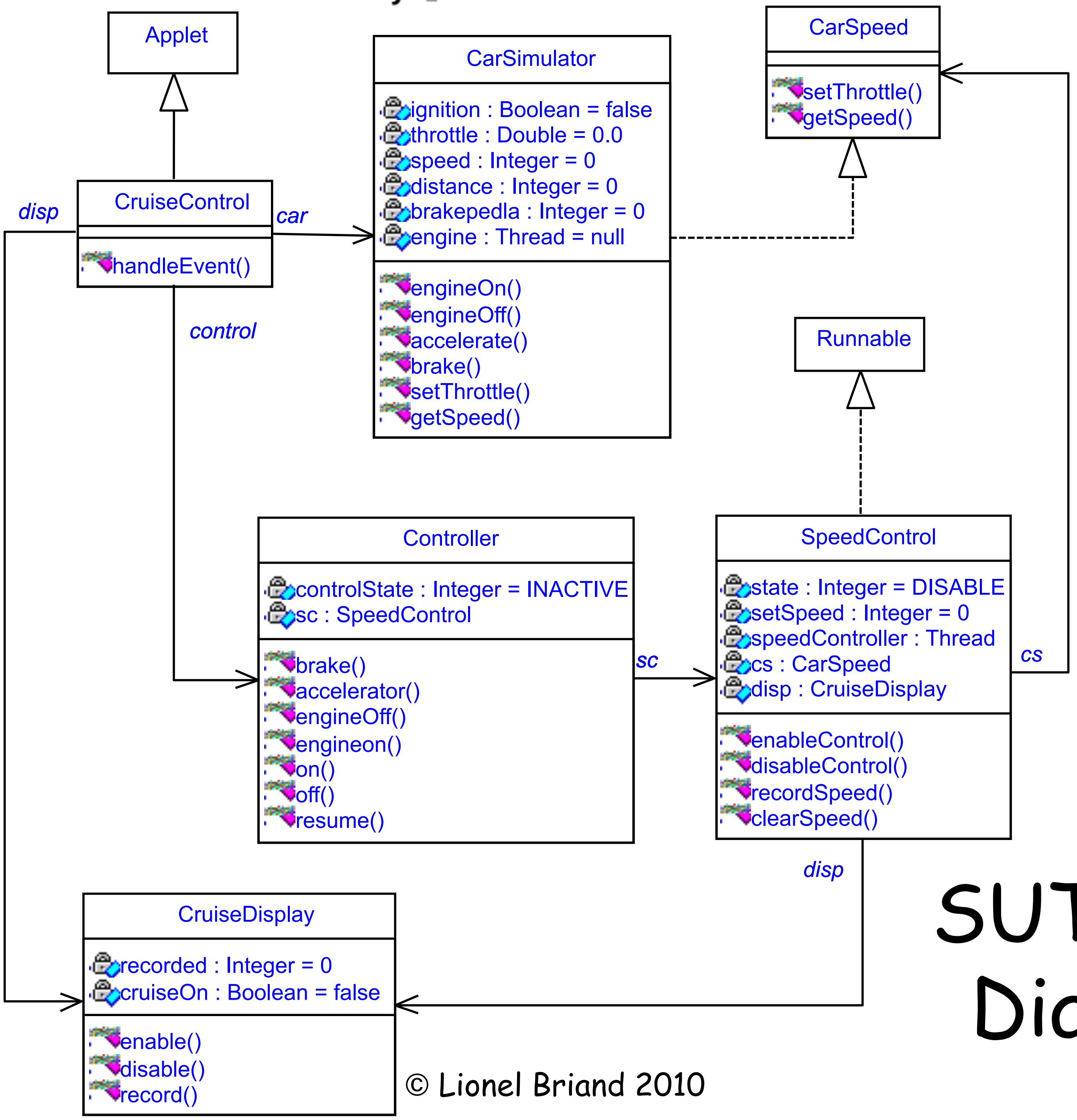
Object-Oriented Class Testing

- 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

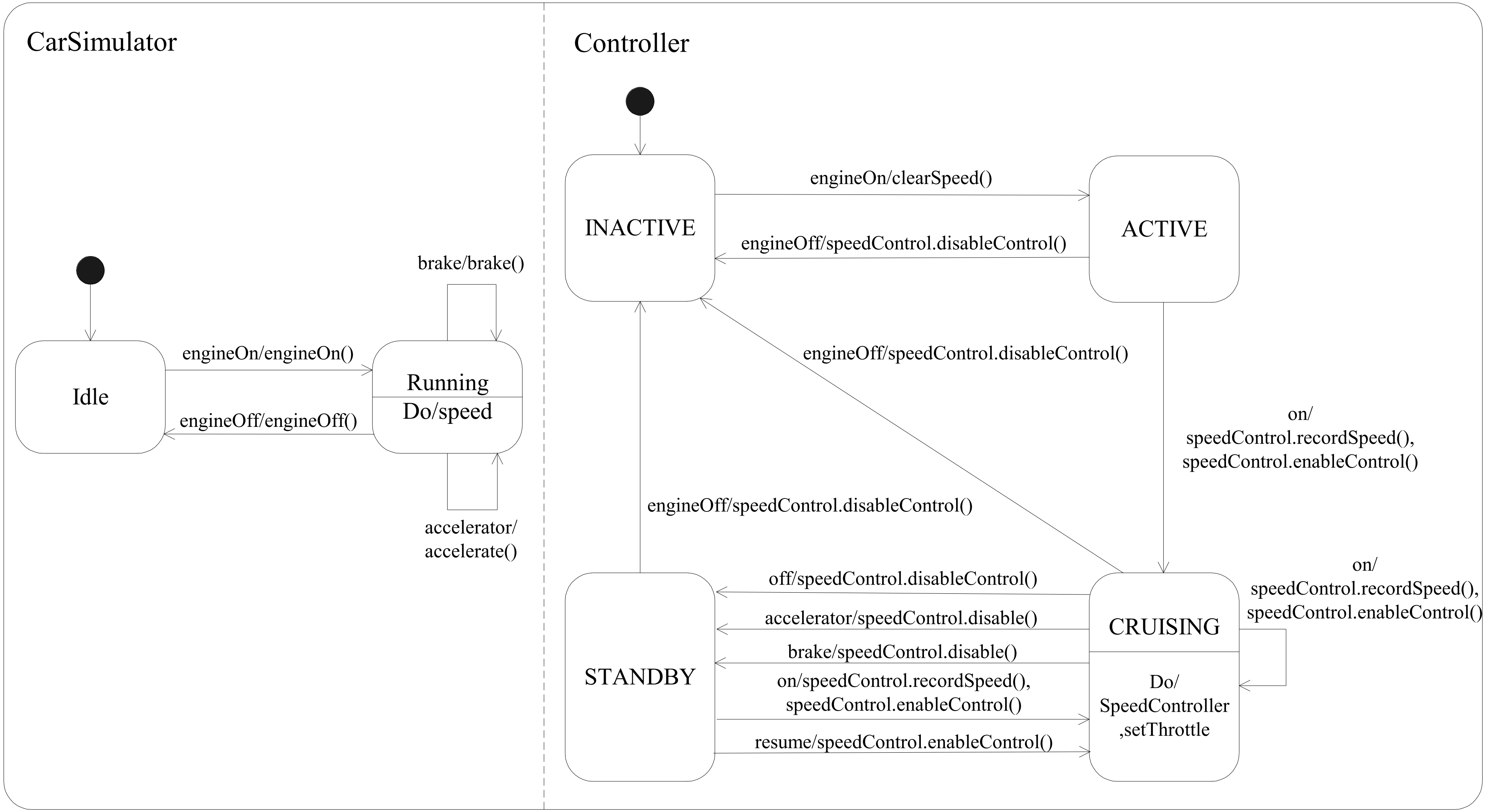
Motivations

- How do we assess the cost-effectiveness of various state-based coverage criteria?
- This cannot be done in an analytical manner (beyond subsumption).
- Empirically, we have several options:
  - Controlled experiments
  - Case studies
  - Simulation
- Simulation allows us to perform more comprehensive studies, but is challenging: (1) automation of the simulation process, (2) external validity of results.

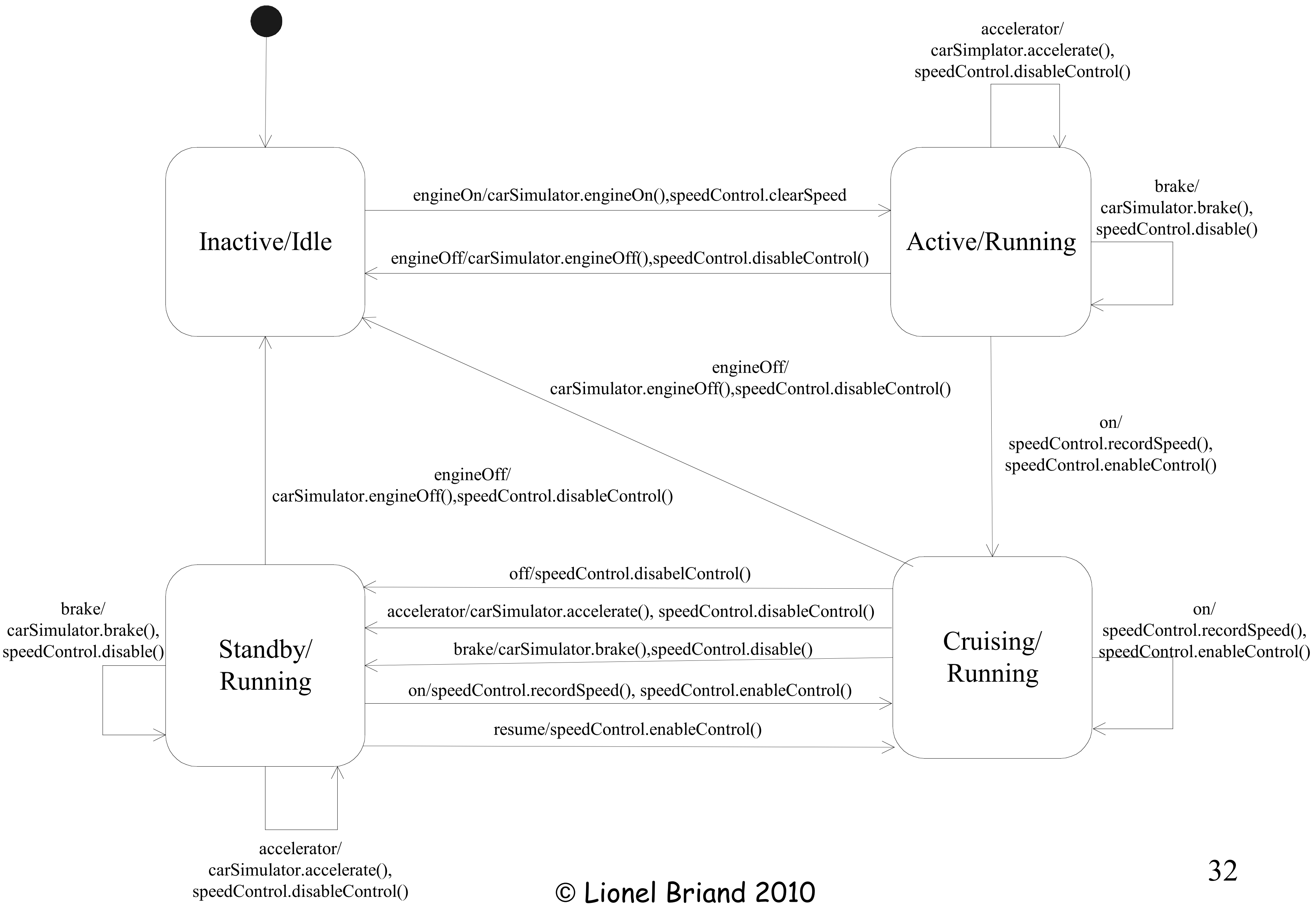
SUT Class Diagram



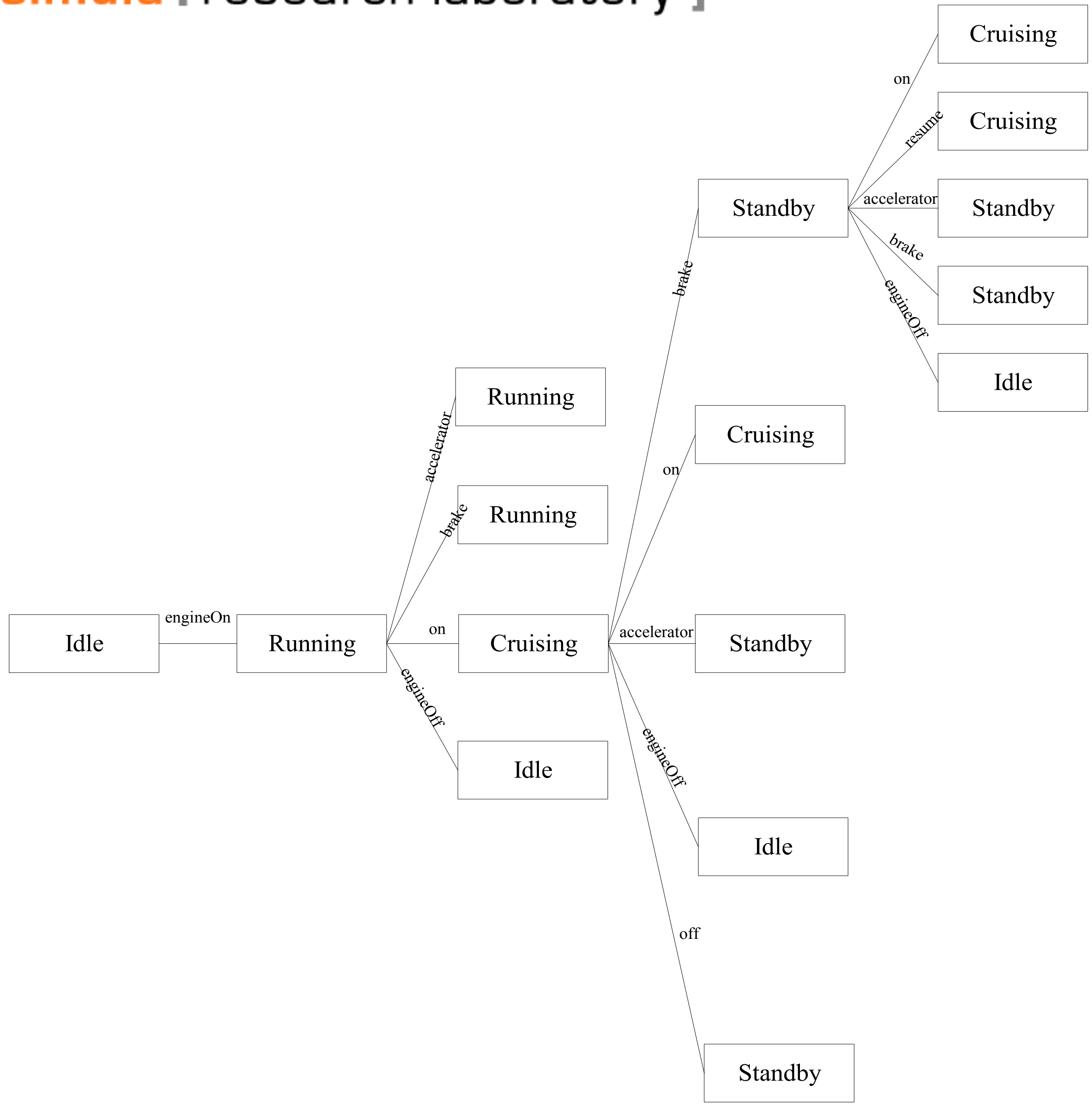
Concurrent Statechart


CarSimulator
Idle
engineOn/engineOn()
engineOff/engineOff()
Running
Do/speed
brake/brake()
accelerator/
accelerate()
Controller
INACTIVE
ACTIVE
engineOn/clearSpeed()
engineOff/speedControl.disableControl()
engineOff/speedControl.disableControl()
on/
speedControl.recordSpeed(),
speedControl.enableControl()
engineOff/speedControl.disableControl()
STANDBY
CRUISING
Do/
SpeedController
,setThrottle
off/speedControl.disableControl()
accelerator/speedControl.disable()
brake/speedControl.disable()
on/speedControl.recordSpeed(),
speedControl.enableControl()
resume/speedControl.enableControl()
on/
speedControl.recordSpeed(),
speedControl.enableControl()

Flattened Statechart

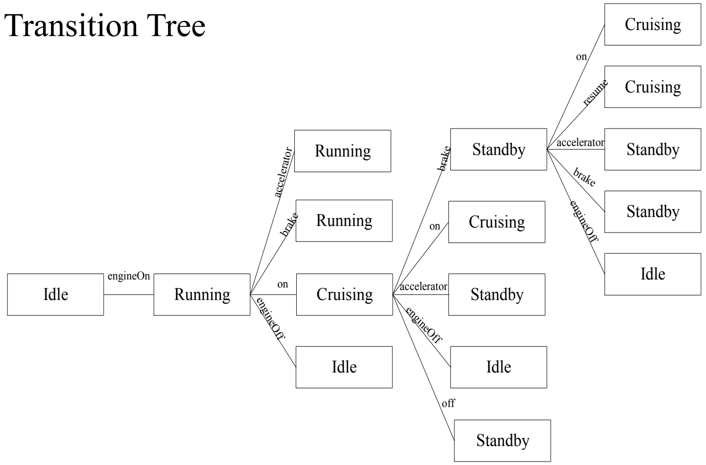

accelerator/
carSimplator.accelerate(),
speedControl.disableControl()
Inactive/Idle
engineOn/carSimulator.engineOn(),speedControl.clearSpeed
engineOff/carSimulator.engineOff(),speedControl.disableControl()
Active/Running
brake/
carSimulator.brake(),
speedControl.disable()
engineOff/
carSimulator.engineOff(),speedControl.disableControl()
on/
speedControl.recordSpeed(),
speedControl.enableControl()
engineOff/
carSimulator.engineOff(),speedControl.disableControl()
off/speedControl.disabelControl()
accelerator/carSimulator.accelerate(), speedControl.disableControl()
brake/carSimulator.brake(),speedControl.disable()
on/speedControl.recordSpeed(), speedControl.enableControl()
resume/speedControl.enableControl()
brake/
carSimulator.brake(),
speedControl.disable()
Standby/
Running
Cruising/
Running
on/
speedControl.recordSpeed(),
speedControl.enableControl()
accelerator/
carSimulator.accelerate(),
speedControl.disableControl()

Example Transition Tree

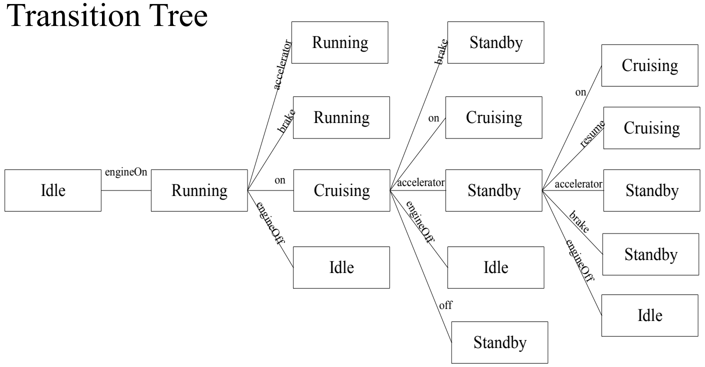


3 Transition Trees

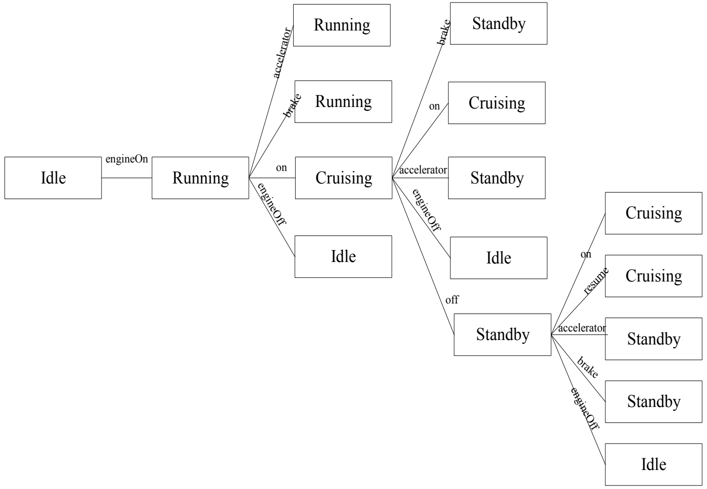
(a) Transition Tree 1:



(b) Transition Tree 2:



(c) Transition Tree 3:



Mutation Operators I

Class mutation operator	Description
LOR (Language Operator Replacement)	Replace a language operator with other legal alternatives.
LCO (Literal Change Operator)	Increase/decrease numeric values or swap boolean literals.
SSO (Swap Statement Operator)	Swap case block statement in a switch statement, swap first and second contained statements in if-then-else statement, and so on.
CFD (Control Flow Disrupt operator)	Disrupt normal control flow: add/remove break, continue, return.
VRO (Variable Replacement Operator)	Replaces a variable name with other names of the same type and of the compatible types.

Mutation Operators II

Class mutation operator	Description
CRT (Compatible Reference Type replacement)	Replace a class type with compatible types.
ICE (Instance Creation Expression changes)	Change an instance creation expression with other instance creation expressions of the same and/or compatible class types.
POC (Parameter Order Change)	Change method parameter order in method declarations.
VMR (oVerloading Method Removal)	Remove the declaration of an overloading method.
AOC (Argument Order Change)	Change method argument order in method invocation expressions.

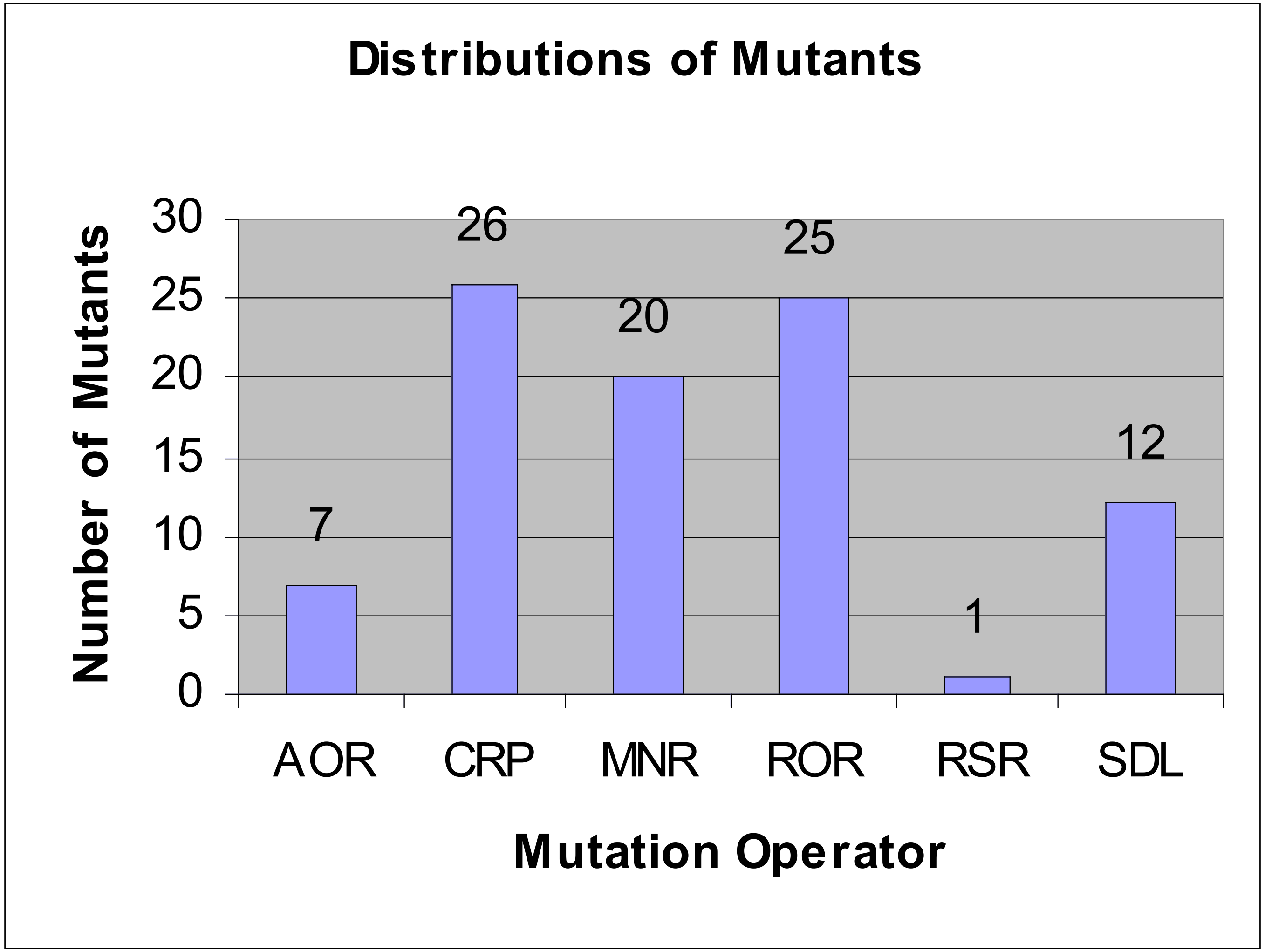
Mutation Operators III

Class mutation operator	Description
AND (Argument Number Decrease)	Decrease arguments one by one.
HFR (Hiding Field variable Removal)	Remove a field variable declaration when it hides the variable in superclasses.
HFA (Hiding Field variable Addition)	Add a field variable of the same name as the inherited field variable.
OMR (Overriding Method Removal)	The declaration of an overriding method is removed
AMC (Access Modifier Change)	Replace an access modifier with other modifiers.

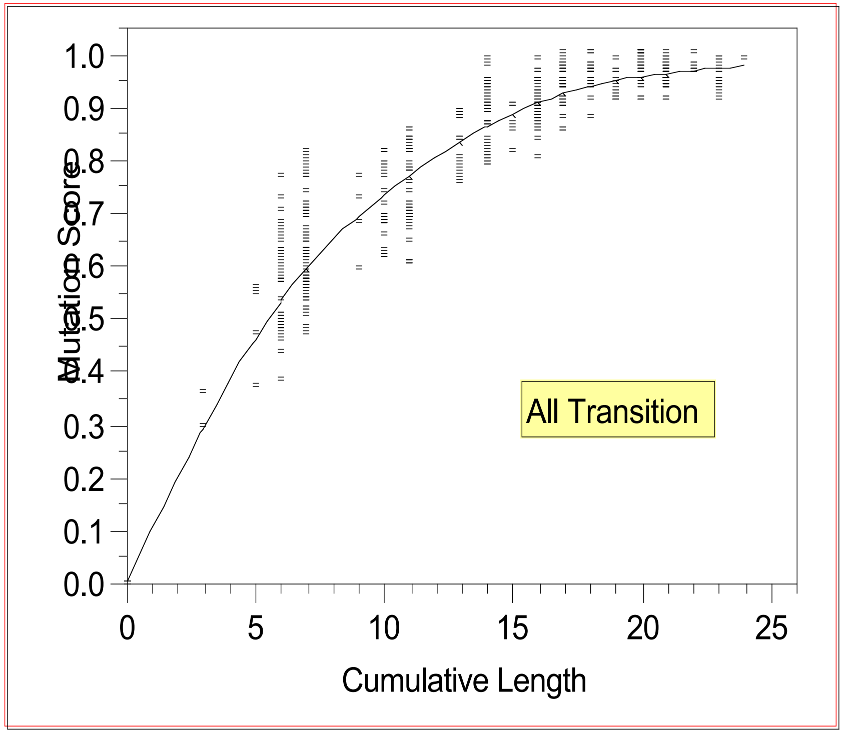
Mutation Operators IV

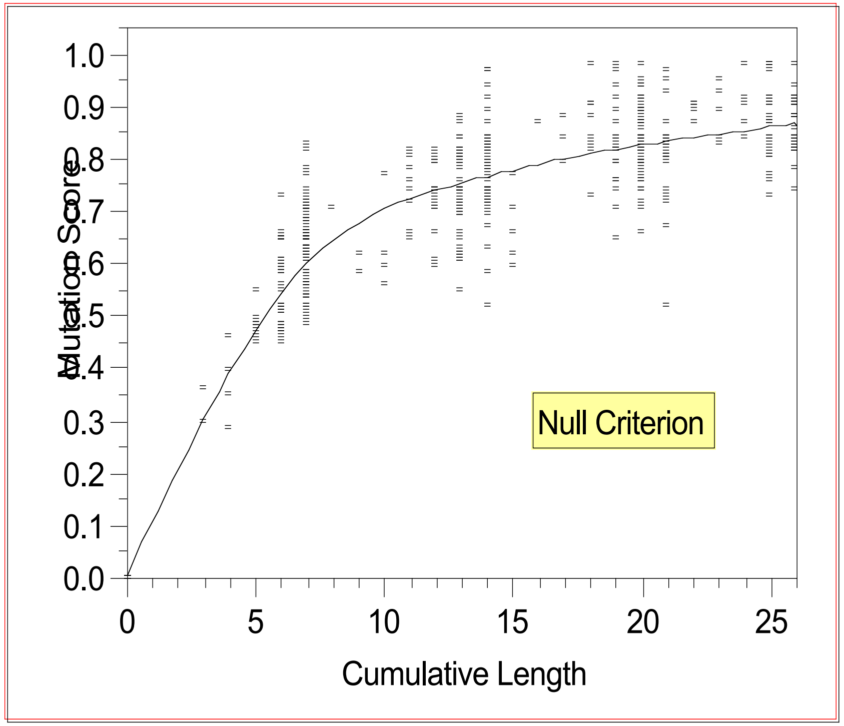
Class mutation operator	Description
SMC (Static Modifier Change)	Add or remove a static modifier.
HER (Exception Handler Removal)	Remove exception handlers one by one.
EHC (Exception Handling Change)	Change an exception handling statement to an exception program statement, and vice versa.

Mutant Distribution

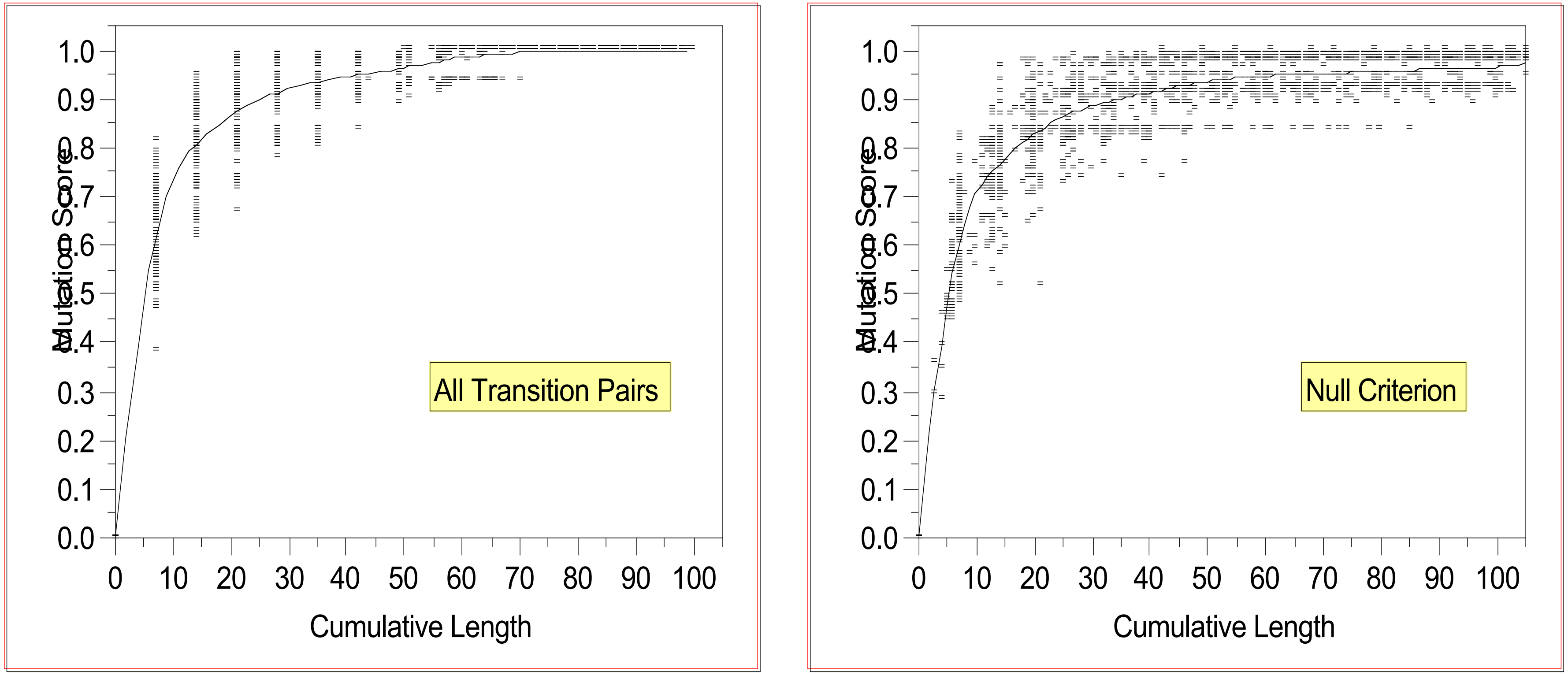


All Transitions

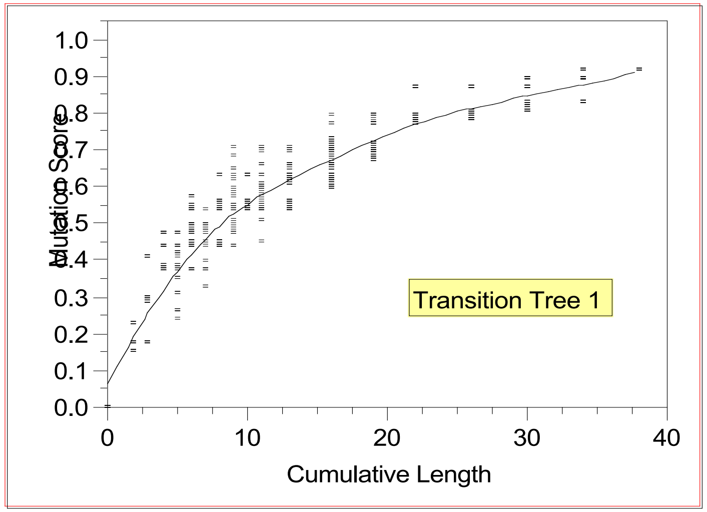


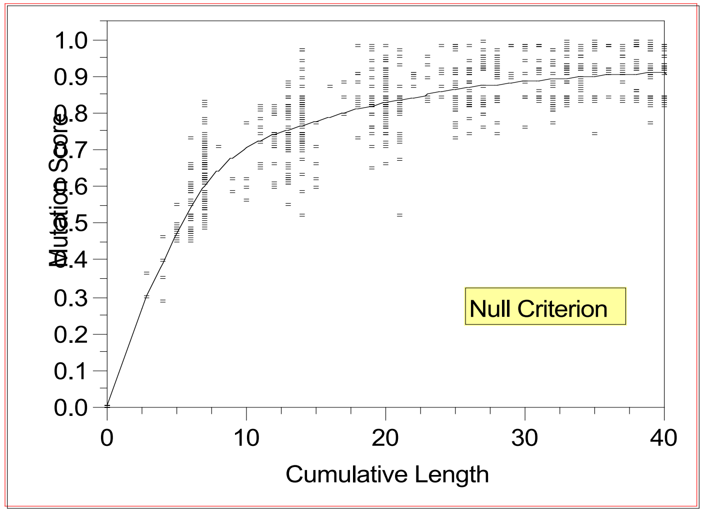


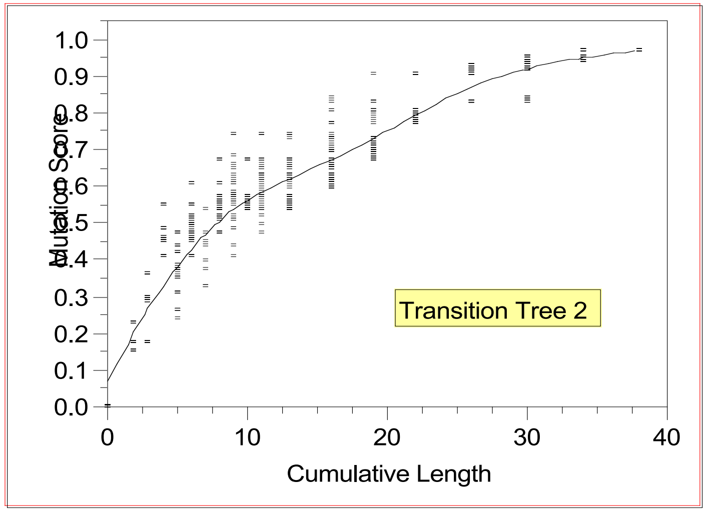
All Transition Pairs

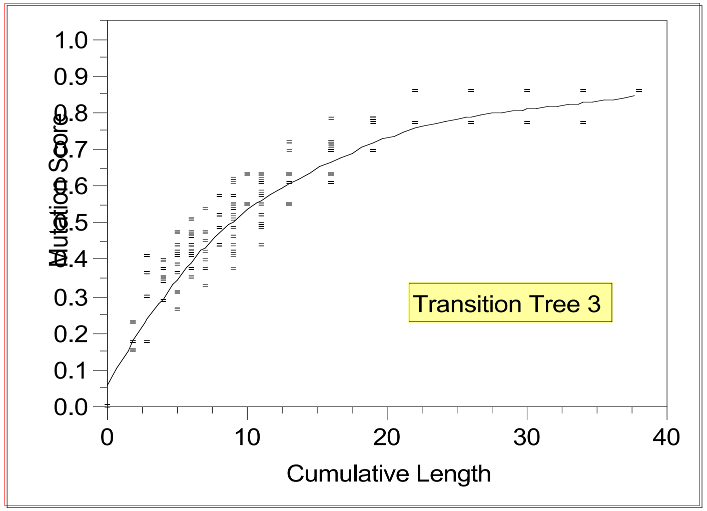


Transition Tree

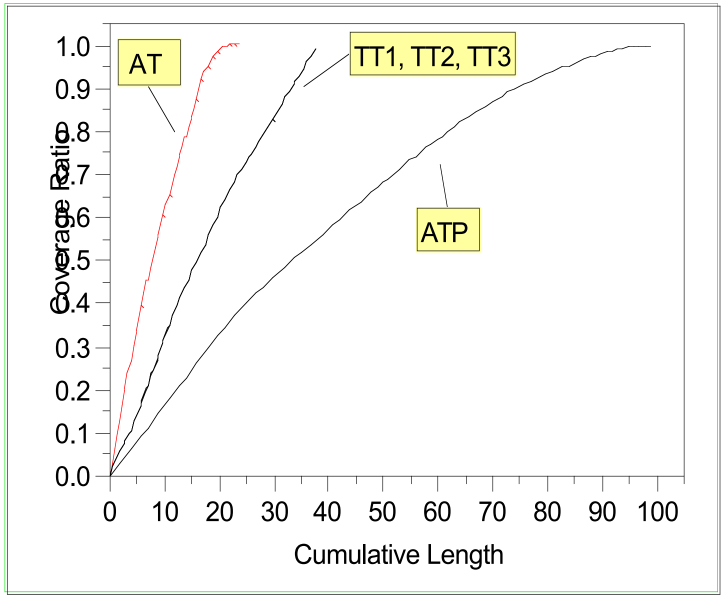


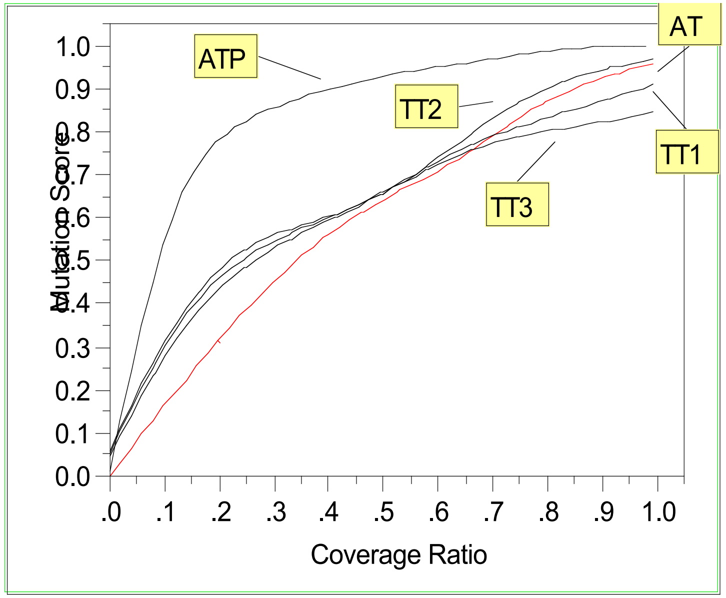


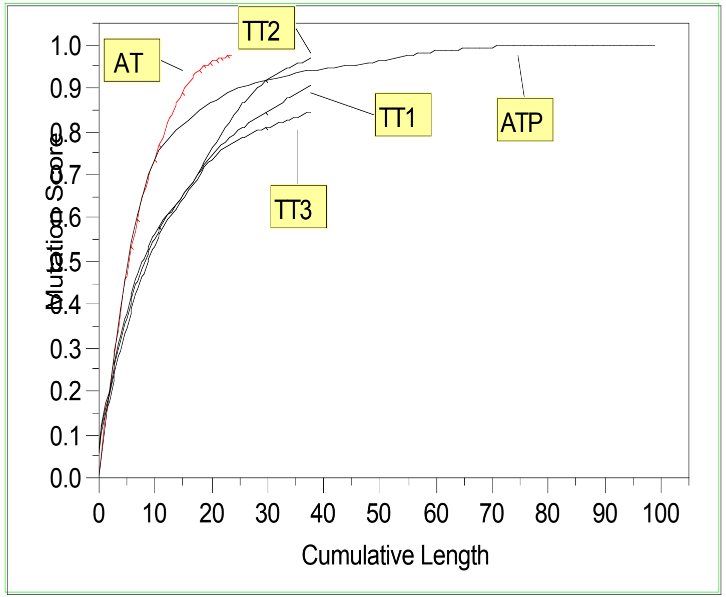




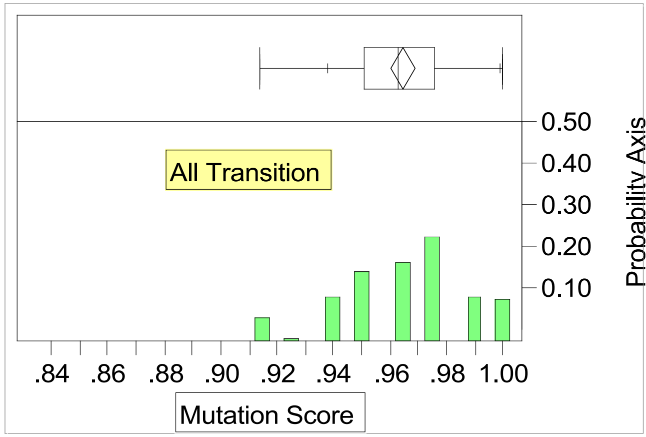
Comparing Criteria

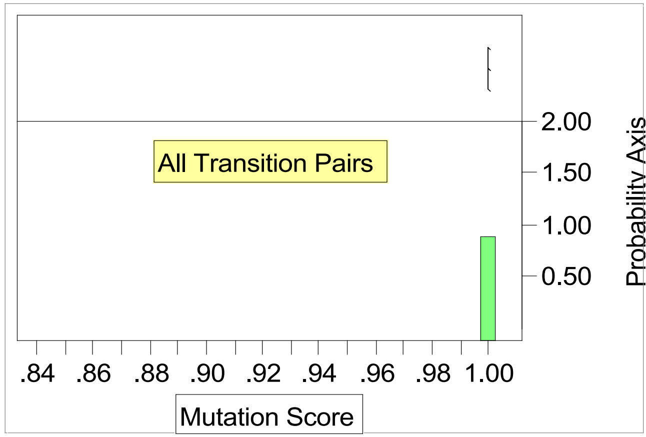


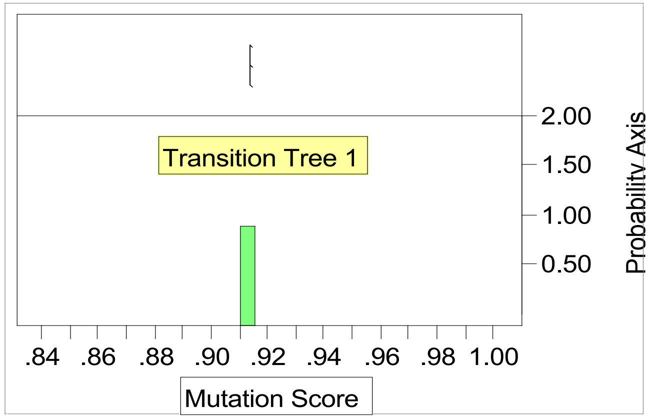


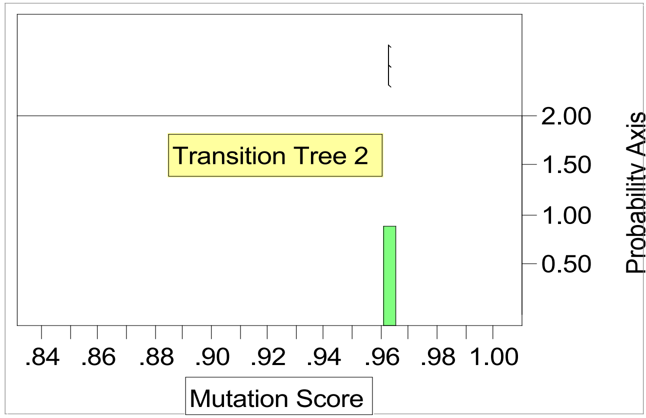


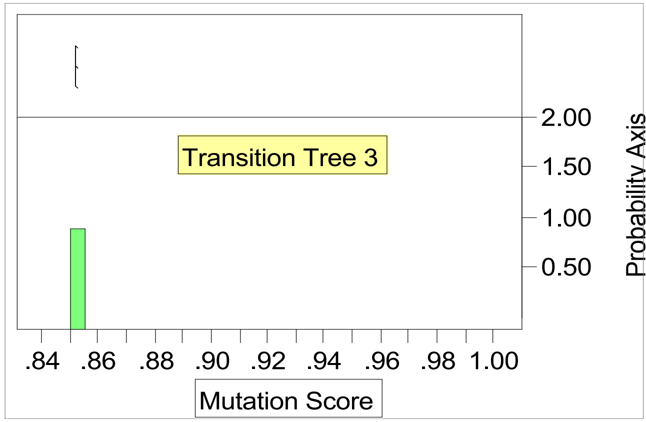
Adequate Test Sets











First Case Study: Conclusions

- All criteria, as expected, definitely perform better than their respective null criterion
- TT is a compromise between AT and ATP in terms of cost
- It is unclear whether it is more effective than AT (in the best case, same average but less variance)
- More studies needed
- When several transition trees are possible, it is important to choose carefully the one that exercise common usage scenarios
- Here, all the tests should not be performed on a stationary car (accelerator event must have been received)
- Use code coverage to select tree? Change definition?

Summary of Results

- AT probably not sufficiently reliable for an indicator of fault detection
- ATP highly reliable but also substantially more expensive
- TT gets mixed results, depending on the statechart properties: guard conditions, etc.
- TT good compromise when many guard conditions. Slightly more expensive but significantly more effective than AT.
- Where alternative transition trees are possible, one should be careful to select the one that exercise the code in the most realistic and complete manner
- MTT is an interesting alternative but more studies are needed
- FP not particularly cost-effective when numerous, complex guard conditions

Test Evaluation based on Mutant Programs

- Take a program and test data generated for that program
- Create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
- The test data are then run through the *mutants*
- If test data detect differences in mutants, then the mutants are said to be *dead,* otherwise live. Oracles are based on both a comparison of outputs and state invariant checking.
- A mutant remains live either because it is equivalent to the original program (functionally identical though syntactically different – *equivalent mutant*) or the test set is inadequate to kill the mutant
- For the automated generation of mutants, we use *mutation operators* , that is predefined program modification rules (I.e., corresponding to a fault model)

Guidelines to Perform Studies

- Seed mutants (possibly randomly) before devising test cases to avoid bias
- Devise a sample of mutants as large as possible and automate the execution of drivers on mutants
- Use adequate mutation operators (depends on programming language)
- Focus on mutations that correspond to faults to be detected by testing (e.g., interface faults for integration testing)
- Examine why a mutant was left alive – could it have been killed, how, what were the chances?

Object-Oriented Class Testing

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

Testability of UML Statecharts

- UML does not prevent the definition of untestable statecharts - It is therefore important to follow some rules to develop testable statecharts
- In order to apply the techniques presented, we need:
  - Unambiguous and testable definition of state (e.g., state invariant) => Oracles
  - Guards should be expressed in an unambiguous syntax (e.g., OCL) => Exercise guards, e.g., FP criterion
  - Unambiguous definitions of events and responses (actions) (e.g., through post-conditions) => Oracles
  - Built-in test support (e.g., get, set, status methods) => Decrease the cost of testing

Built-in Test Support

- *Get state functions:* The simplest case is a function that evaluates the state invariant and returns a Boolean indicating whether an object is in that state.
  - During OO analysis and design, each state is defined by a *state invariant*, e.g., set of data member values for S0 = (allowVend=0, curQrts=0) in *CCoinBox*
  - Each state invariant is associated with an *executable assertions* in the code, e.g., isS0()
- *Set state functions*: It may be hard to reach a state in which a test sequence starts, so we may need built-in methods to set objects in certain states that are difficult to reach.

“Get State” for BoundedStack

```
Class BoundedStack {
        public:
         …

        private:
          const char* get_BoundedStack_state () const {
            if (cur_index == 0) return “empty”;
            else if (cur_index == max_index)
                 return “full”;
            else return “partially-full”;
          }
};
```

- Only test drivers should be allowed to use these operations
- Get/Set operations can be private and test drivers can be *friend classes* in C++ or they can be protected and inherit from the classes under test

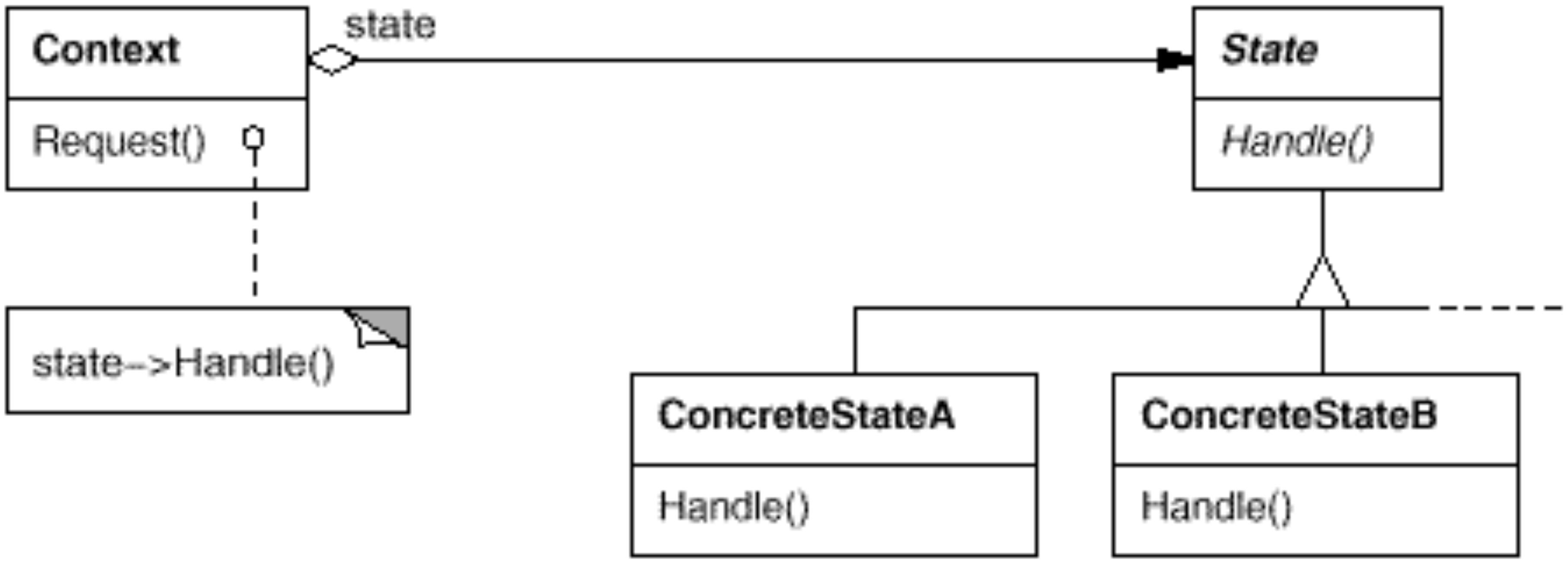
Using the State Design Pattern

- The state design pattern facilitates the implementation of get/set operations (enforces it and simplifies it)
- If we use the *state design pattern*, the *Get/Set state operations* can be coded as methods in the subclasses representing the states.
- Can be part of a *design standard* in an organization
- This is an example of how design decisions relate to testability

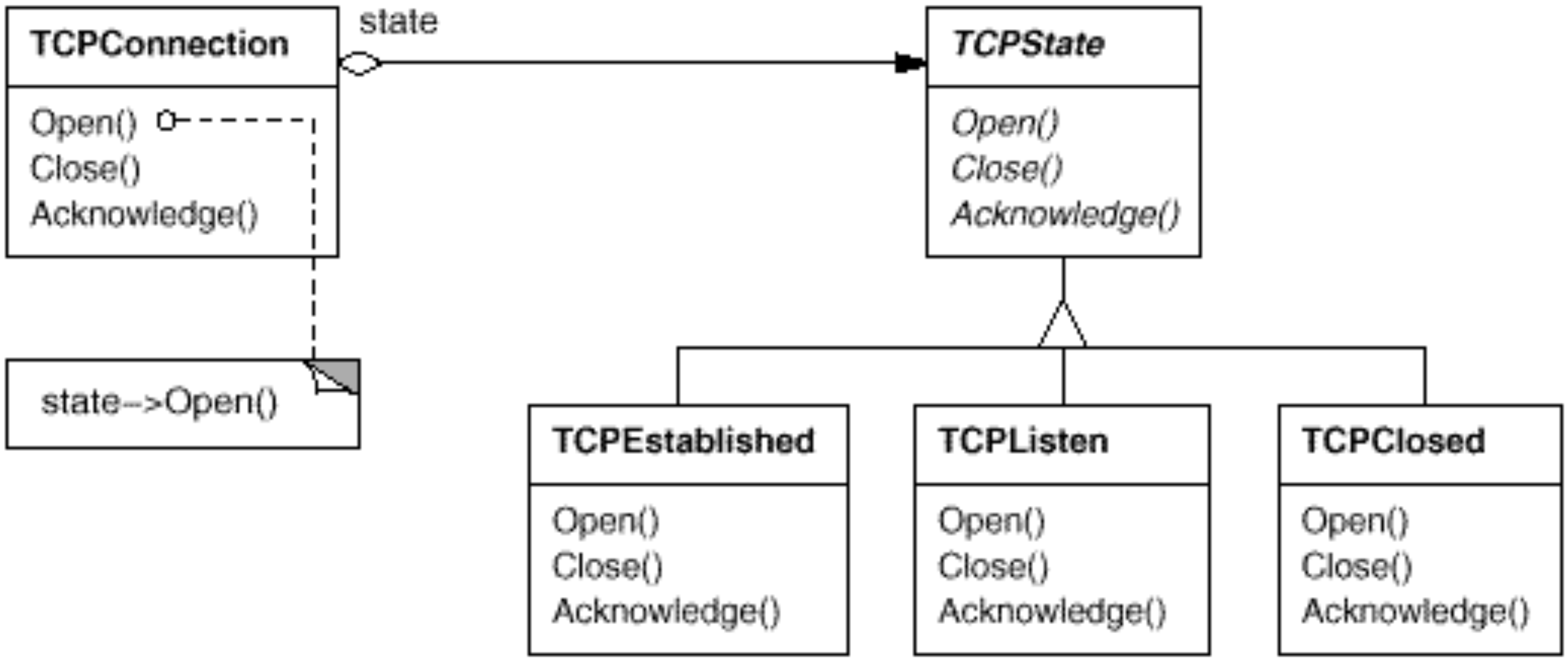
State Pattern

- *Intent:* Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- *Applicability:* An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- *Consequences*:
  - It localizes state-specific behavior and partitions behavior for different states.
  - It makes state transitions explicit (object changes subclass)
  - Ease the addition of new states

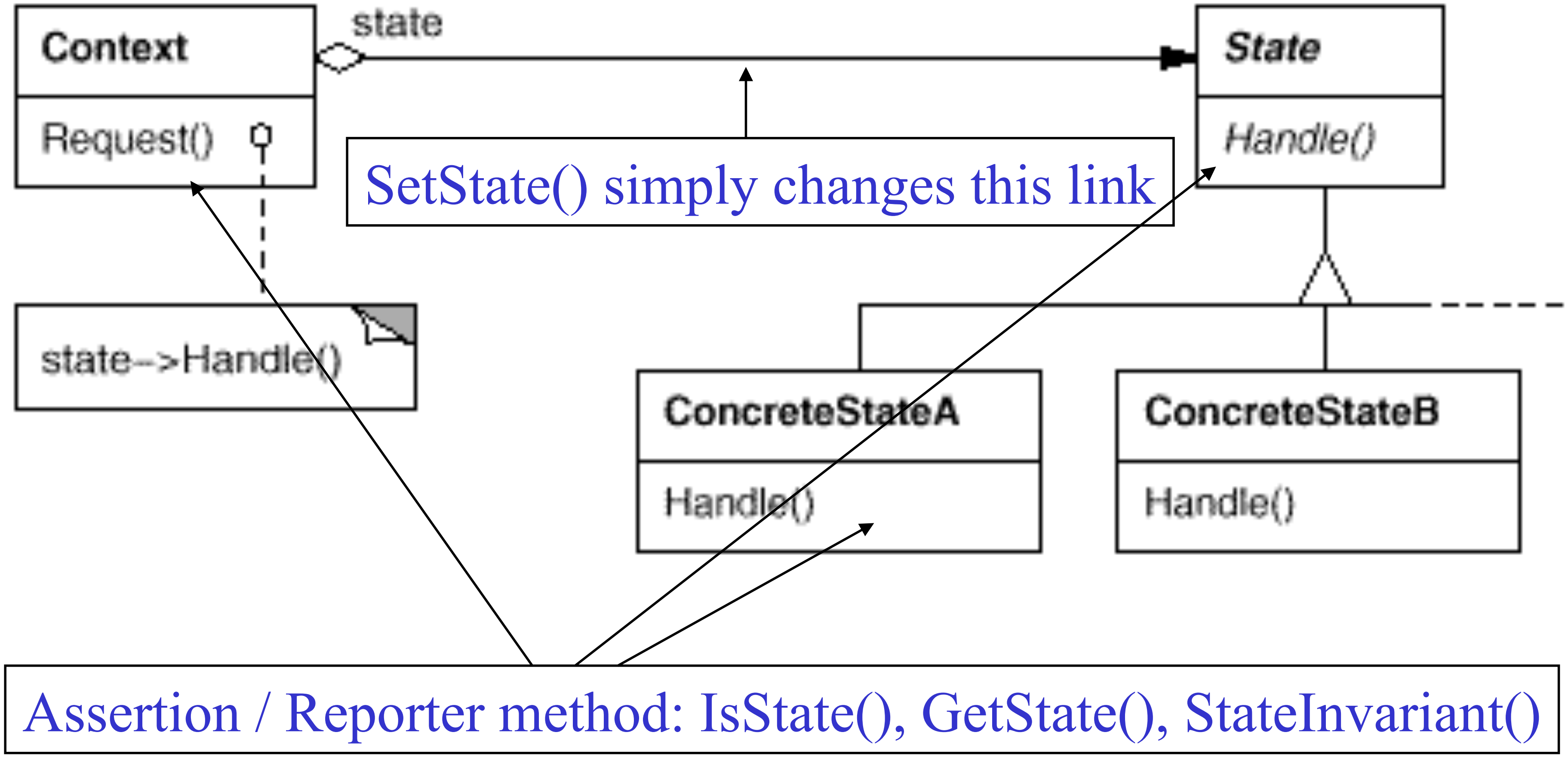
Structure


Context
Request()
state
state->Handle()
State
Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

State Pattern: Example



Built-in Test Support



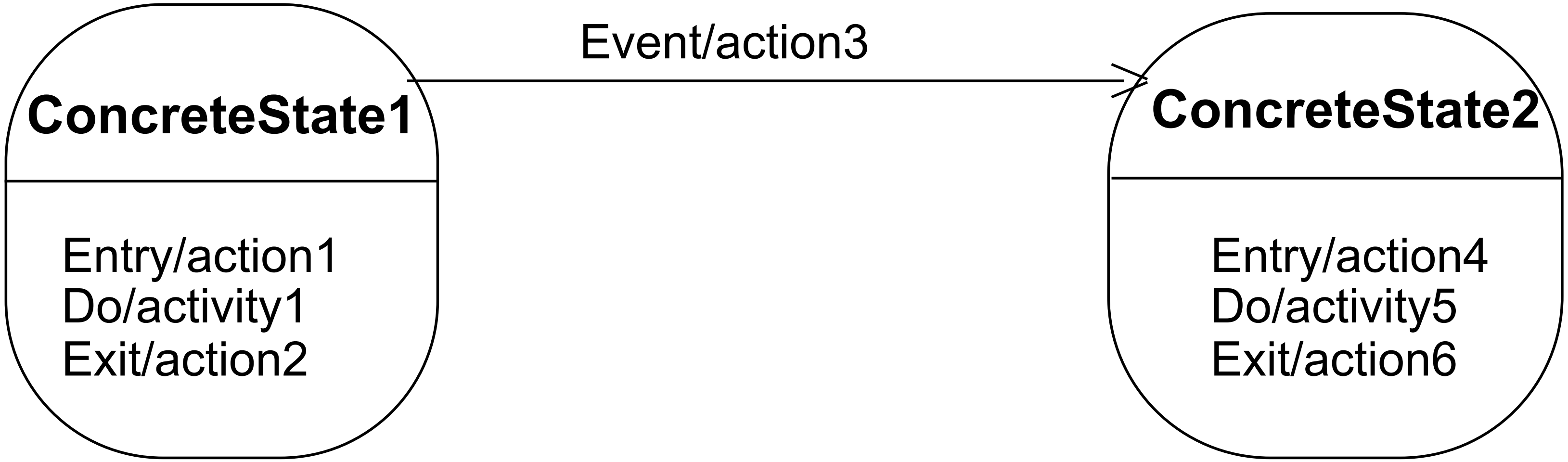
SetState() simply changes this link

Assertion / Reporter method: IsState(), GetState(), StateInvariant()

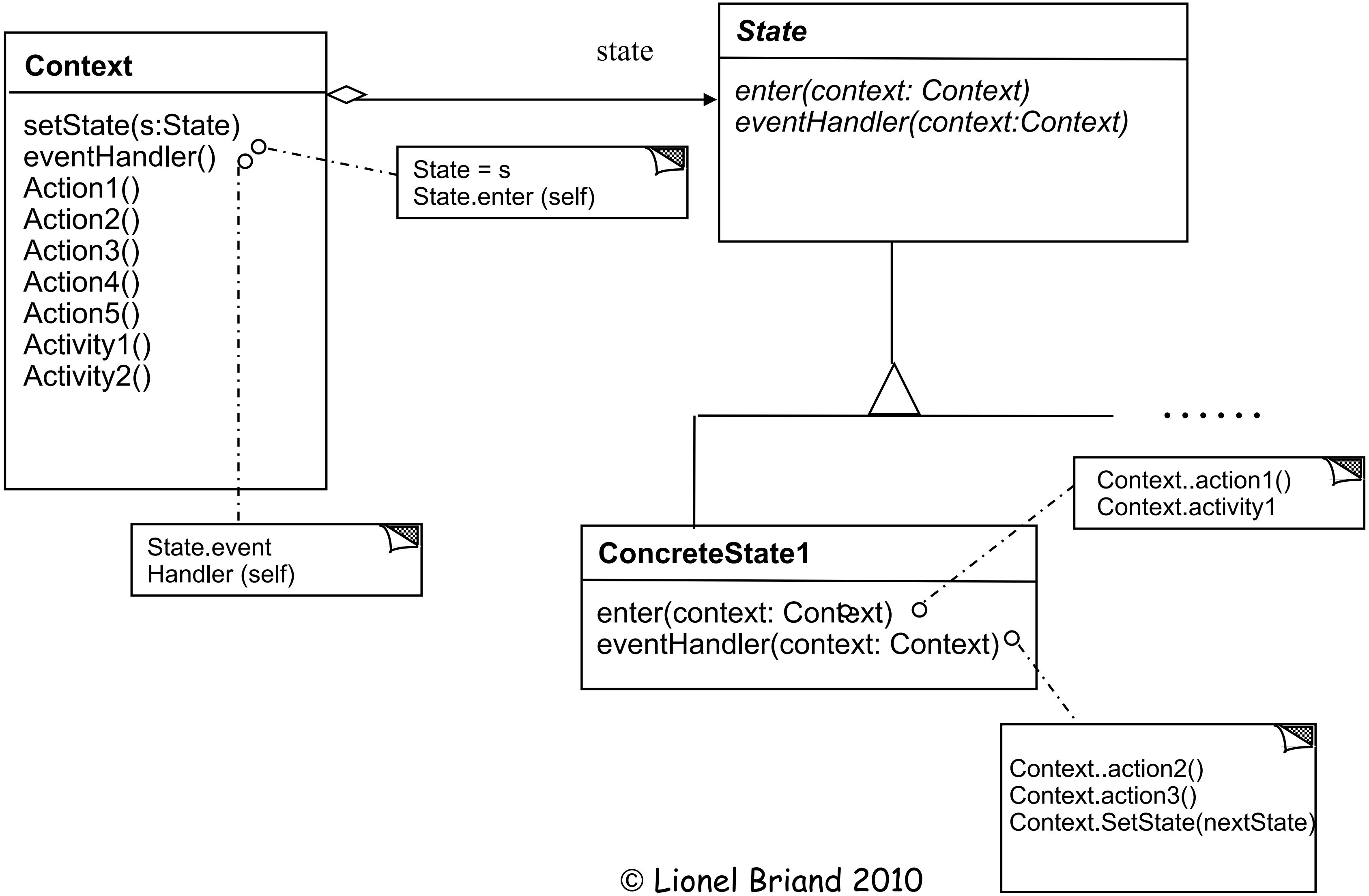
State Pattern Deficiencies

- The state pattern does not address UML statechart concepts
- Actions on transitions
- Entry and exit actions associated with states
- Activities performed while in states
- The state pattern needs to be expanded

Extending the State Pattern



Extending the State Pattern II



Context

setState(s:State)
eventHandler()
Action1()
Action2()
Action3()
Action4()
Action5()
Activity1()
Activity2()

state

State

enter(context: Context)
eventHandler(context:Context)

State = s
State.enter (self)

State.event
Handler (self)

……

ConcreteState1

enter(context: Context)
eventHandler(context: Context)

Context..action1()
Context.activity1

Context..action2()
Context.action3()
Context.SetState(nextState)

Discussion

- When no *Set state operations* are used, then sequences of public methods must be executed by the test driver to reach the desired state for testing – this may be complex and error-prone
- When *Set state operations* are used, coding or generating a test driver becomes easier, but at the price of additional code to be provided by the developers
- One needs to ensure that *set state operations* are not misused in the application code – it is exclusively for the usage of test drivers
- *Get state operations* enables an easier detection of corrupt states, eliminate the need for characterization input sequences (Chow), but again require additional code to be provided by the developers
- Checking *state invariants* may not be enough to detect corrupt states – checking the *class invariant* may be required
- All the above is easier to implement when using the state design pattern

Object-Oriented Class Testing

- Introduction
- Accounting for Inheritance
- Testing Method Sequences
- State-Based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Drivers

- Assume you have a `diff` operation that computes the difference between two ordered sets (i.e., elements that are in the first set but not in the second)

```
// Java code chunk
OrdSet s1, s2, s3;
…
s3 = s1.diff(s2);         // s3=s1-s2, e.g.,{1,5}=
  {1,4,5}-{4}
System.out.println(s3);
```

- How to execute the test cases you devised using a black-box or white-box technique?
  - How do you build the test driver that executes your test cases?

Drivers - First solution

```
public class Driver {
public static void main(String argv[]) {
    // The test case consists in s1={1,4,5} and s2={4}
    OrdSet s1 = new OrdSet();
    OrdSet s2 = new OrdSet();
    OrdSet s3 = new OrdSet();
    s1.add(1); s1.add(5); s1.add(4);   // adding elements to
                                          sets
    s2.add(4);
    s3 = s1.diff(s2);
    System.out.println(s3);
   }
}
```

- Testing technique → many test cases
- Solution: writing all the test cases in function main in the driver.

➢ Can't selectively run test cases (regression testing), can't have different test sets for a class

Drivers - Second Solution

- One static method per test set (namely TS1, TS2, …)
- One static method per test case (namely TC1, TC2, …)
- Test sets can share test cases

```
public class Driver {
public static void main(String argv[]) {
     TS1(); // Test set 1
}
public static void TS1() {
     TC1();
     TC2();
     …
}
public static void TS2() {
     TC5();
     TC2();
     …
}
```

```
public static void TC1() {
     OrdSet s1 = new OrdSet();
     OrdSet s2 = new OrdSet();
     OrdSet s3 = new OrdSet();
     s1.add(1); s1.add(5);
     s1.add(4);
     s2.add(4);
     s3 = s1.diff(s2);
     System.out.println(s3);
}
public static void TC2() {…}
…
}
```

Discussion

- All the static methods that execute test cases have (usually) the same structure
  - Here: creating sets s1 and s2, and calling the diff
- A testing technique can produce hundreds of test cases
- What happens if we want to add test cases?
  1. Add new TCxxx and TSxxx static methods
  2. Add calls to these methods in the main
  3. Compile the driver
  4. Execute the driver

➢ We create different tests for different purposes, e.g., successive regression test sets

➢ We can change one statement in the `main()` to execute different test sets

Drivers - Third solution

- The main function in the driver reads a text file:
  - 1 test set per text file
  - No need to recompile when changing test cases
  - The driver must read the text file, create the objects according to its contents and execute the diff

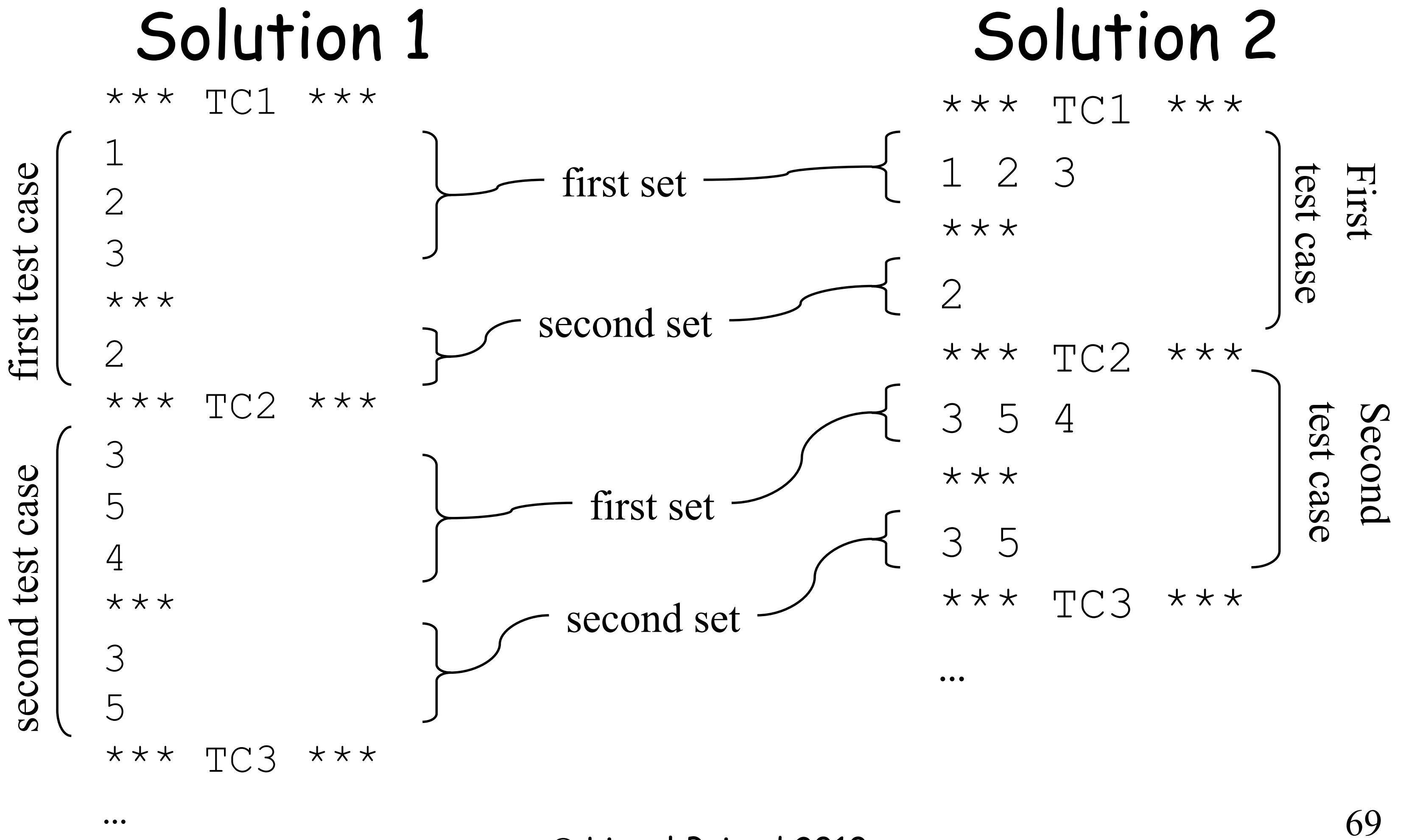
```
public class Driver {
public static void main(String argv[]) {
   OrdSet s1, s2, s3
   // open file and read test set name …
   while (not at the end of file) {
   // read a test case (elements in sets)
   // instantiate s1 and s2
   // execute the diff, assign result to s3
   // output the test case number and the result
   }
}
```

Discussion

- What is the format of the text file?
  - Many test cases in the text file
- The driver (the main) needs to:
  - Identify test cases:
    - Beginning and end of a test case in the file
    - Number/Name of the test case
  - Know how to read the data in test cases
    - Separation between different fields in the test case (e.g., the two sets)

➢Different strategies: must be standardized

Test Set File Format



Solution 1

```
*** TC1 ***
1
2
3
***
2
*** TC2 ***
3
5
4
***
3
5
*** TC3 ***
...
```

first test case

second test case

first set

second set

first set

second set

Solution 2

```
*** TC1 ***
1 2 3
***
2
*** TC2 ***
3 5 4
***
3 5
*** TC3 ***
...
```

First test case

Second test case

More Complex Drivers

E.g., test cases are different (different kinds of executions/data)

Solutions

1. Several different drivers (main functions) with different execution flows (i.e., with different file format)
   - Driver (main) one: file format 1
   - Driver (main) two: file format 2
   - ➢ Implementation depends on the programming language (difference between Java and C/C++)
2. One driver with several possible inputs (e.g., command line arguments, or in the text file)
   - Driver (main) with input 1 (command line argument): file format 1
   - Driver (main) with input 2 (command line argument): file format 2
   - ➢ Implementation language independent (we pass information on the command line)

More Complex Drivers (in Java)

- In Java, we can have different driver classes, each having a main function.
  - In the same directory we would have files OrdSet.java, Driver1.java, Driver2.java, …
  - And Driver1.java, Driver2.java, …, all have a main function
- We just choose which main (driver class) we want to execute when executing the Java Virtual Machine
  - java Driver1 […]
  - java Driver2 […]
  - …
- Solution can be used even if we test a program (not a class), that already has a main function
  - Drivers can be considered as additional "entry-points" in the program

More Complex Drivers (in C++)

- Conditional compilation in C/C++
  - Enables the programmer to control the execution of preprocessor directives and the compilation of program code.
  - Using `#define`, `#ifdef`, `#ifndef`, and `#endif` preprocessor directives, that:
    - Determine whether symbolic constants are already defined
    - Determines whether parts of the code are compiled (and thus executed)
- Two solutions:
  - Remove/Add the preprocessor directives
    - Requires modifying source files and compilation
  - Use compiler options
    - Requires compilation only
- Solutions can be used even if we test a program (not a class), that already has a main function
  - Drivers can be considered as additional "entry-points" in the program

More Complex Drivers (in C++)

```
// File: ClassTester.h
#include "ClassTested.h"
class ClassTester {
public:
   ClassTester();
   void runTestSuite();
private: …
};
```

```
// File: MainProg.cpp
#define DRIVER
#ifndef DRIVER
#include "ClassTested.h"
class MainProg {
public: …
private: …
};
#endif
```

```
// File: ClassTester.cpp
#include "ClassTester.h"
ClassTester::ClassTester() {…}
void ClassTester::runTestSuite(){…}
// code to run and report test cases
```

```
// File: ClassTesterMain.cpp
#define DRIVER
#ifdef DRIVER
#include "ClassTester.h"
int main() {
   ClassTester tester;
   tester.runTestSuite();
}
#endif
```

- These lines make the driver execute
- Removing them make the main program execute

More Complex Drivers (in C++)

```
// File: MainProg.cpp
#ifndef DRIVER
#include “ClassTested.h”
class MainProg {
public: …
private: …
};
#endif
```

```
// File: ClassTesterMain.cpp
#ifdef DRIVER
#include “ClassTester.h”
int main() {
   ClassTester tester;
   tester.runTestSuite();
}
#endif
```

```
// File: ClassTester.cpp
#ifdef DRIVER
#include “ClassTester.h”
ClassTester::ClassTester() {…}
void ClassTester::runTestSuite(){…}
// code to run and report test cases
#endif
```

- Executing the Main of the program
  - the main in file MainProg.cpp

```
cc -c MainProg.cpp
cc -c ClassTester.cpp
cc -c ClassTesterMain.cpp
cc MainProg.o ClassTester.o
   ClassTesterMain.o -o Exe
```

- Executing the Main of the driver
  - the main in file ClassTesterMain.cpp

  (compiler option `-D` defines constants `DRIVER`)

```
cc -DDRIVER -c MainProg.cpp
cc -DDRIVER -c ClassTester.cpp
cc -DDRIVER -c ClassTesterMain.cpp
cc MainProg.o ClassTester.o
   ClassTesterMain.o -o Exe
```

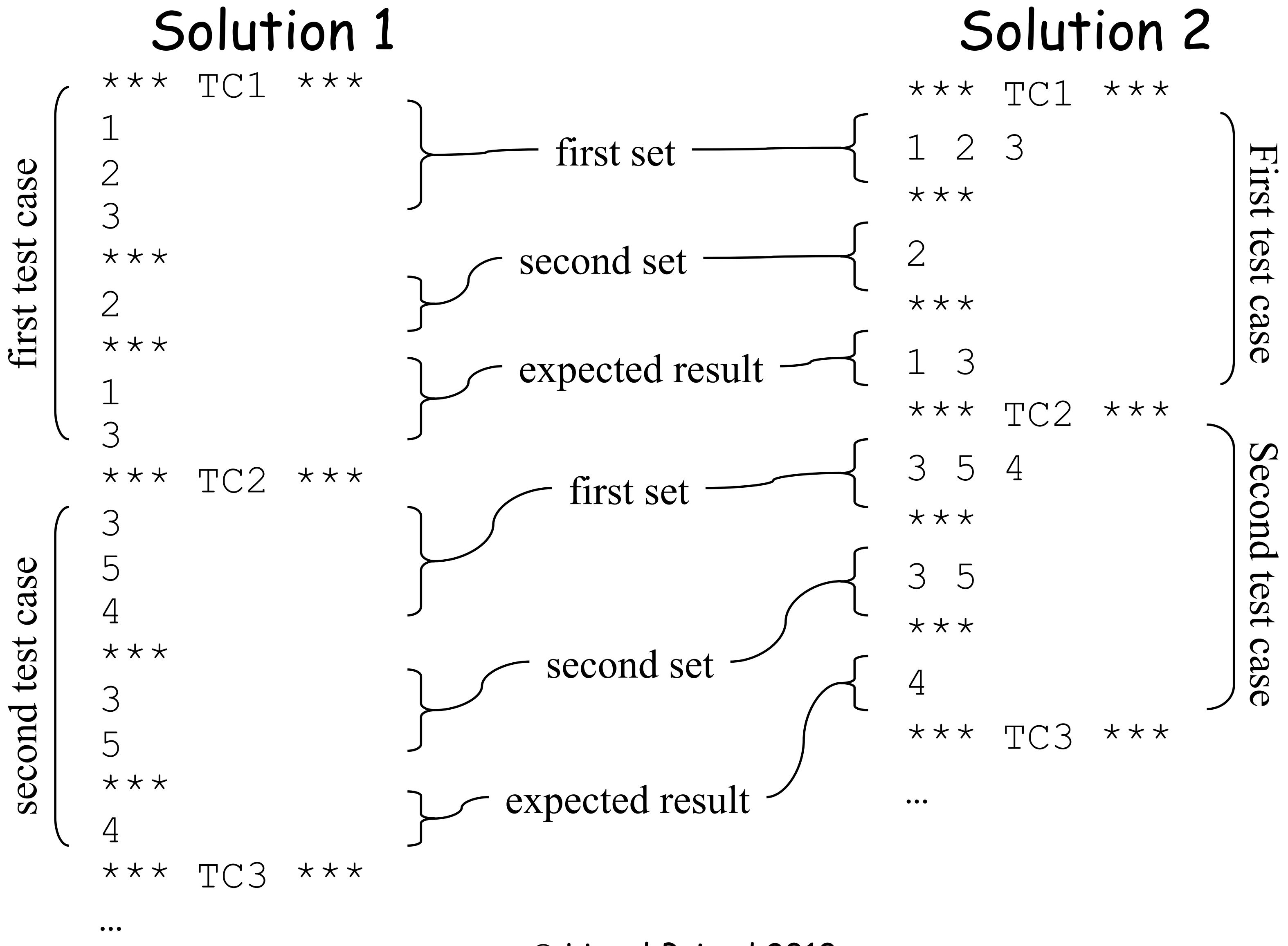
The Oracle

- In the previous approaches for the implementation of the driver, the driver reports the observed output.
- Deciding which test cases failed (a fault has been revealed) is done manually.
  - Comparing the expected output with the observed one, for each test case
- Automating the oracle, i.e., the comparison requires that:
  - We know exactly the expected output (which is the case in the `diff` example)
    - However, sometimes, we do not know exactly what is the result, as (for instance) doing (i.e., implementing) the comparison would correspond to building the function we test
    - This is when using assertions to check class and state invariants can come handy
  - Then the driver outputs the test case number when a failure has been detected
    - We only need to know which test cases have failed

The Oracle

- The oracle is included in the driver
  - The expected output must be found in the input text file
    - Format?
- The oracle:
  - gets the output produced by the system under test
    - In our example, the `toString` method of class `OrdSet` is called (format of returned string?)
  - gets the expected output from the input text file
  - Performs the comparison
    - In our example, a simple String comparison

The Oracle



Solution 1

```
*** TC1 ***
1
2
3
***
2
***
1
3
*** TC2 ***
3
5
4
***
3
5
***
4
*** TC3 ***
...
```

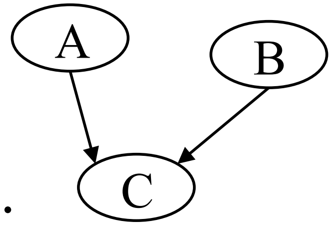
Solution 2

```
*** TC1 ***
1 2 3
***
2
***
1 3
*** TC2 ***
3 5 4
***
3 5
***
4
*** TC3 ***
...
```

Stubs

- Need for stubs:
  - Parts of the system that are not yet unit tested or even available (i.e., the code is not ready)
  - Simulation of hardware devices
- Example 1:
  - Modules A and B use services provided by module C.
    - Here, modules can be functions, classes, sub-systems
  - But: A uses only part of the services in C, say C_A

    B uses only part of the services in C, say C_B
  - Then:
    - When testing A, we create a stub simulating C_A's behavior
    - When testing B, we create a stub simulating C_B's behavior
- Comment: "If the stub is realistic in every way, it is no longer a stub but the actual routine" [Beizer]



Stubs

Example 2:

- An Automated Teller Machine (ATM) has a keyboard (i.e., the hardware device) and a `Keyboard` class.
- During development and testing, a stub is developed for class `Keyboard` that is not "connected" to any hardware device.
  - Rather, it is used by the driver to feed the ATM system with a PIN number, an amount, …
- Typical test case:

  1. Enter the PIN, 2. Select the account, 3. Enter an amount

  - Each time, the ATM system invokes methods of the `Keyboard` stub.
  - So the `Keyboard` stub must return the correct value according to the test case.
  - The driver must set different values during the execution of a test case.
    - The driver must "stop" the ATM in order to set the next value (threads?)
    - Class `Keyboard` knows where (e.g., a file) to read the different inputs.

Other Considerations

- Non-determinism
  - What if the behavior of the system under test is not deterministic?
  - Concurrent systems
  - Impact on test cases, drivers, stubs, oracles
- Infinite loop because of a fault
  - How do we decide (when automating the execution of test cases) that a test case failed?
- The system under test has a GUI
  - The GUI is removed to facilitate automation during (unit/integration/system) testing, I.e., the GUI needs to be substituted with the driver at compilation time.
  - Testing a GUI is a separate task (http://en.wikipedia.org/wiki/GUI_Testing)
    - It's not functional testing
- There are many tools to facilitate the coding and execution of drivers, e.g., JUnit for Java