CISC/CMPE 327 Software Quality Assurance

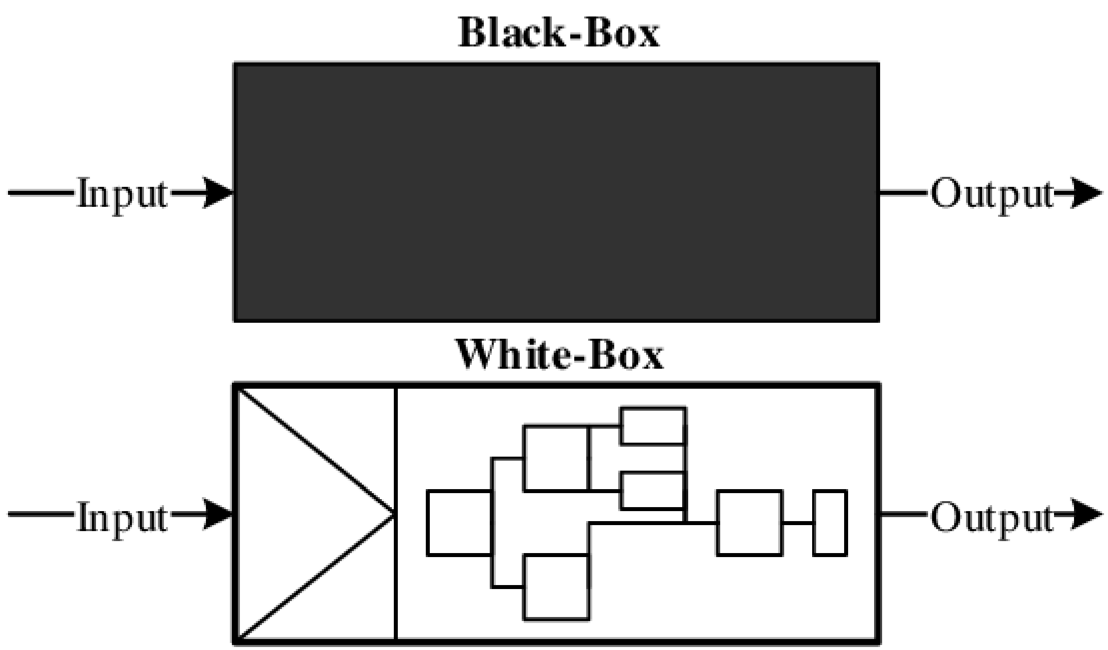
Queen's University, 2019–fall

Lecture #11 Black Box Testing – Output Coverage

Black Box Testing

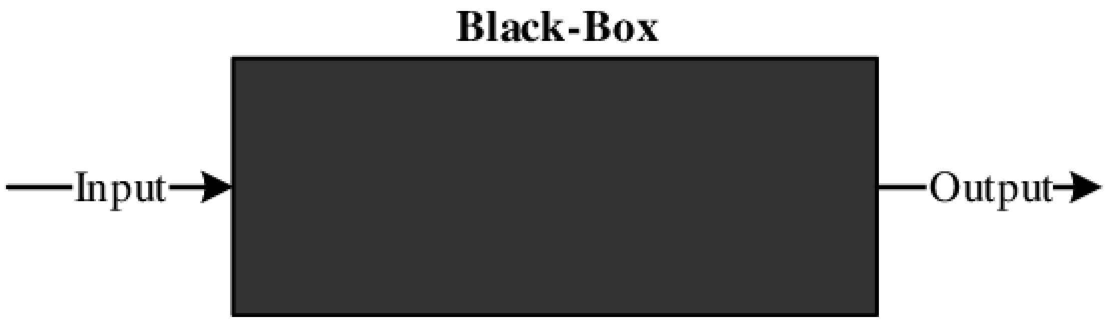
- Outline
  - input coverage testing
  - output coverage testing
  - Output coverage methods
    - exhaustive
    - output partitioning
  - Testing multiple input or output streams
  - "Gray box" testing
  - Model-based testing

Black Box Testing



Output?

- all the possible outputs specified in the functional specification (requirements)



Output Coverage Testing

1. Analyze all the possible outputs
2. Create tests to cause each one

output -> input

"Given as input two integers x and y, output all the positive numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Output: 3, 6, 9, 12

x: ???

y: ???

Output Coverage Testing

- More difficult than input coverage
- Effective:
  - > finding problems
  - > develop deep understanding of the requirements

Exhaustive Output Testing

- Test them all !
- requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"

Exhaustive Output Testing

- Test them all !
- requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"

- Only two test cases:
  - Output 1, output 0

V. S. Input Coverage - Exhaustive

- Test them all !
- requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"

V. S. Input Coverage - Partition

- Test them all !
- requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"

V. S. Input Coverage - Partition

- Test them all !
- requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"

  - Numbers equal, numbers not equal, first number zero / positive / negative, second number zero / positive / negative

Exhaustive Output Testing

- More practical than Exhaustive Input Testing?
  - Exhaustive output testing makes one test for every possible output
  - Practical more often than input testing
  - But still impractical in general
    - an infinite number of different possible outputs

Output Partitioning

- Partition all the possible outputs into a set of equivalence classes with something in common

Output Partition Testing

"Given as input two integers x and y, output all the positive numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

– The output is a list of integers, so we might partition into the following cases:

Number of integers in output

output values	zero	one	many
all positive	P1	P2	P3

Output Partition Testing: Designing Inputs

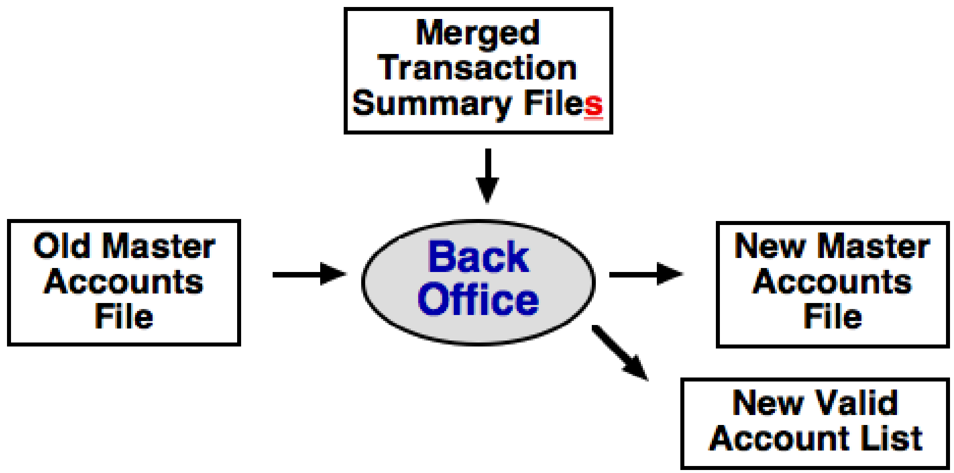
- Design inputs to cause outputs in each partition
- This is difficult and time-consuming
  - The biggest drawback to output coverage testing!
- We cannot find such an input
  - This implies an error or oversight in either the requirements or in the partition analysis

Multiple Input or Output Streams

- A Separation of Concerns
  - Multiple inputs (variable, file, socket etc.)
  - Must create separate coverage tests for each one
  - Effectively, what we do is treat each separate file or stream as a pre-made input or output partition, within which we make a separate set of smaller partitions

Multiple Input or Output Streams

- A Separation of Concerns



  - create separate output partition test sets for each
- Partitioning system in general
  - we assume that each class of input or output is independent of the others

Black Box Testing at Different Levels

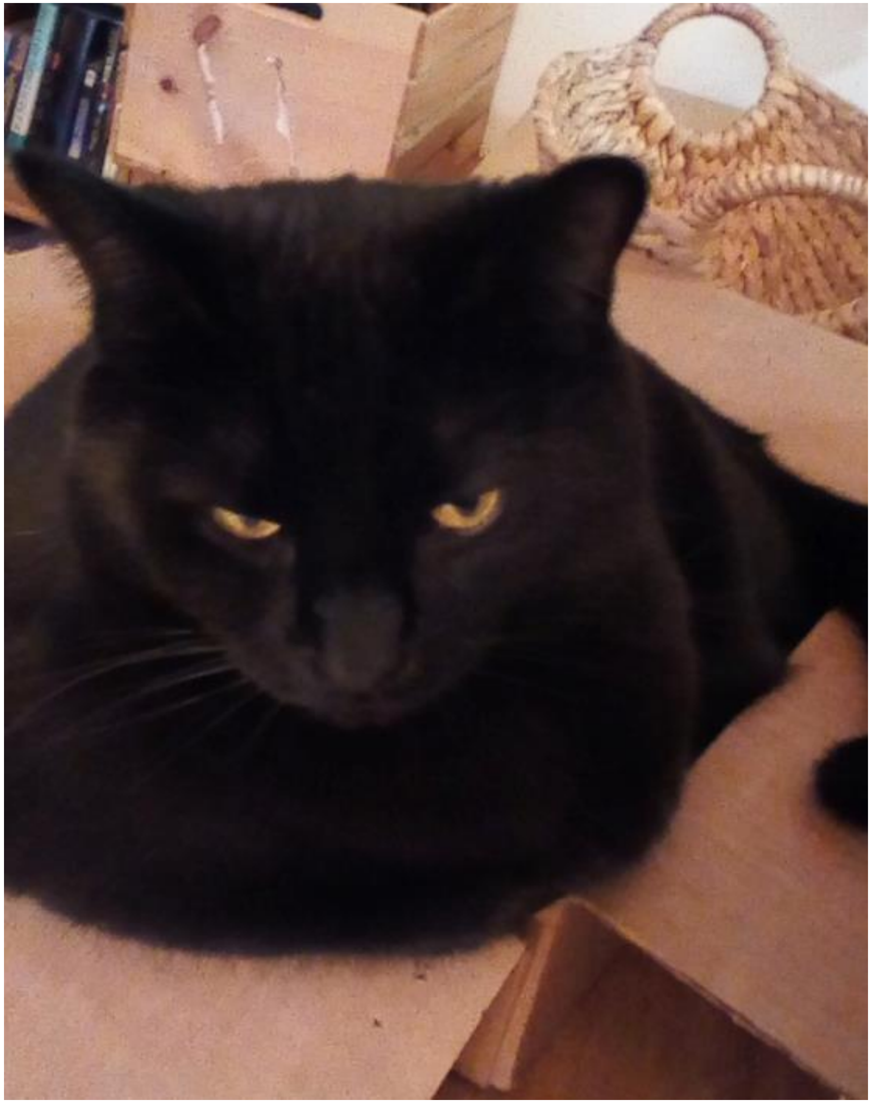
- levels of testing
  - Unit/Integration/System

- In particular, black box testing of all kinds can be used at every level of software development

Black Box Testing at Different Levels

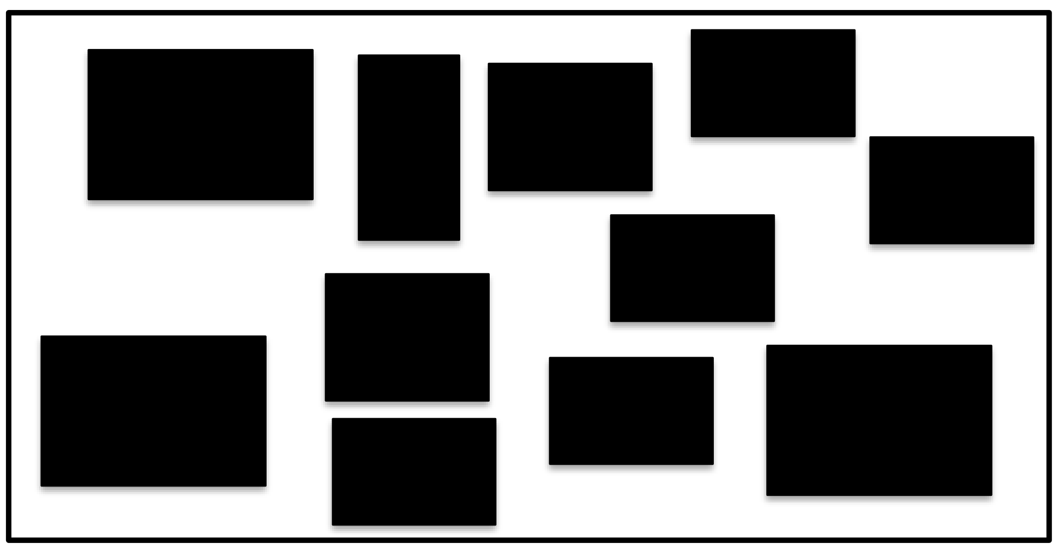
- system testing
  - Functional coverage
  - Input coverage
  - output coverage
  - tests for functional specification
    - (the requirements for the software)
  - This is pure black box testing

“I do not understand why everything in this ~~script~~ course must inevitably ~~explode~~ be a box.”



"Gray" Box Testing

- If we already have a design...
  - Visible: an architectural (class level) design, or even a detailed (method level) design
  - Test each of those



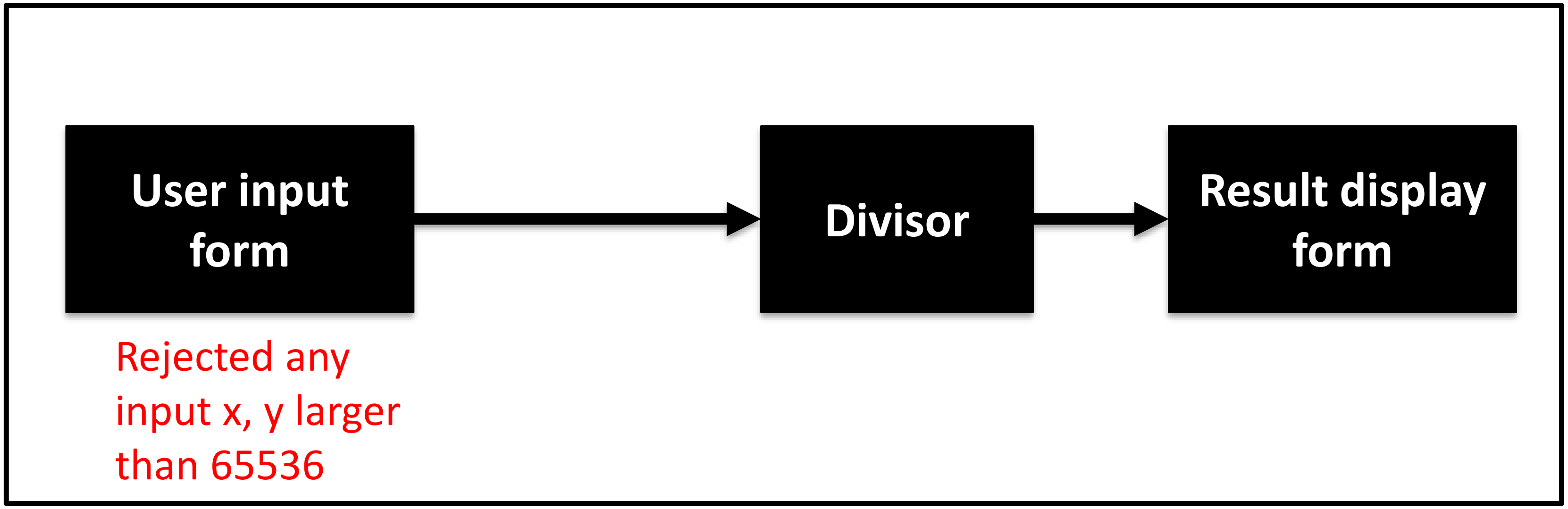
"Gray" Box Testing

- If we already have a design...
  - apply the same black box coverage techniques to the interface of each class to create class level black box tests (a.k.a. interface tests)

  - If we know how a software code fragment is written, we can design tests with that in mind

"Gray" Box Testing

- Imagine that our divisors example program was used in an interface

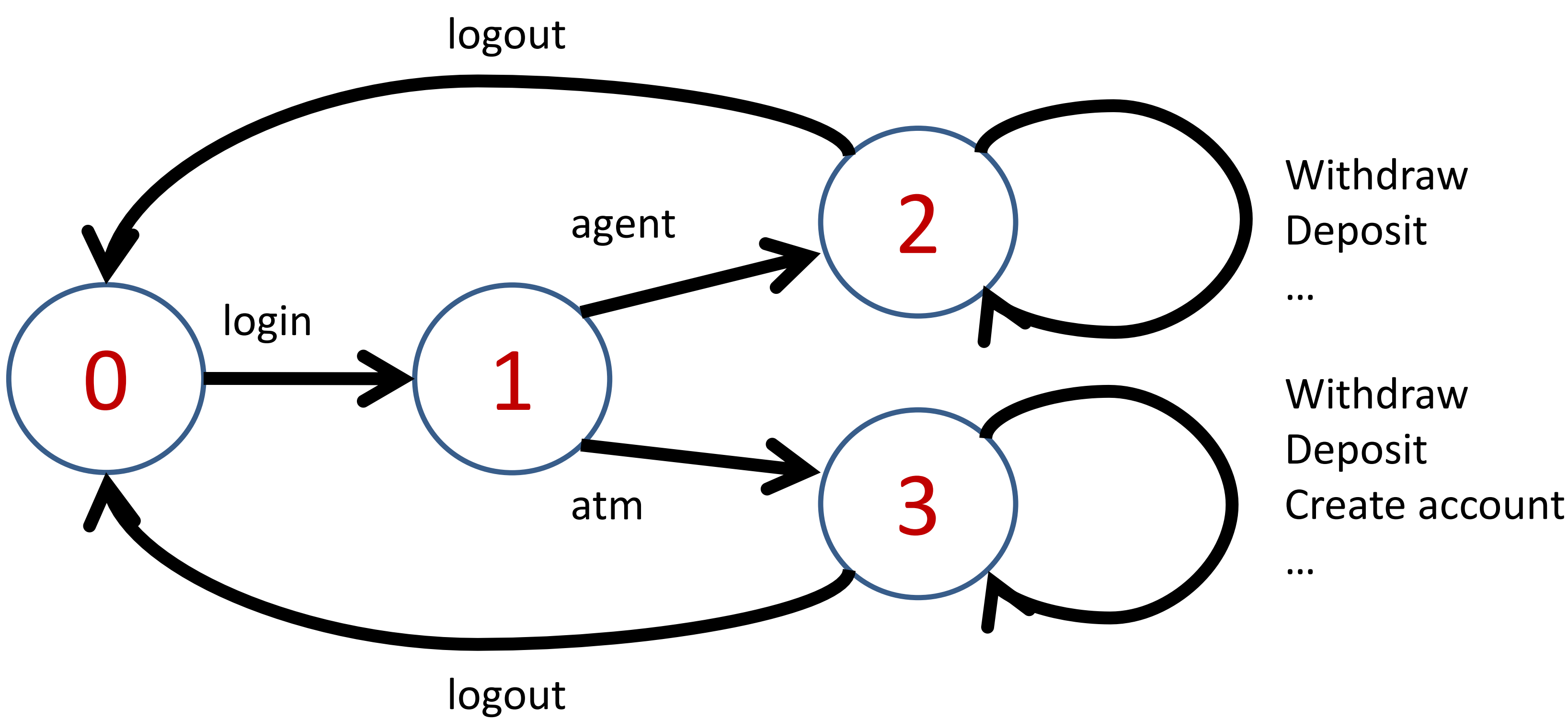


Model-Based Testing

- Model-Driven Engineering (MDE)
  - A modern new black-box method is model-based testing, part of MDE
  - Model-based testing does not use a specification, but rather a formal state model of the process implemented by the program
  - State models are high-level abstractions (simplifications) of the program's intent, usually expressed at the level of the problem domain rather than the computer
  - State models ignore implementation details, but retain essential states of the process

Model-Based Testing

- Model-Driven Engineering
  - For example, the following might be a state model of the login aspect of the Front End



Model-Driven Engineering

- Models are formal (mathematical) specifications of the process to be implemented
- Formal models can be used in several ways
  - To verify that the model (formal specification) is itself correct, using model checking (NASA, Airbus) (CISC 422)
  - To generate some or all of the implementation from the formal model, if it is detailed enough (General Motors)
  - To test that the implementation is consistent with the formal model (model-based testing)

Model-Based Testing

- The basic idea of model-based testing is that the model is smaller and simpler than the code, so we can generate far fewer tests to cover it than we would for the implementation
  - For example, white-box testing
- The model also encodes the entire specification, so we know that if we make a set of tests to cover the model, every essential requirement is tested

Model-Based Testing

- Because the model is formal, we can automatically generate the tests, then run them against the implementation to verify that it correctly implements the model (which was itself verified using model checking)
  - Of course, this is the ideal situation
  - In practice, models may be partial or may address only some aspect of the requirements

Model-Based Testing

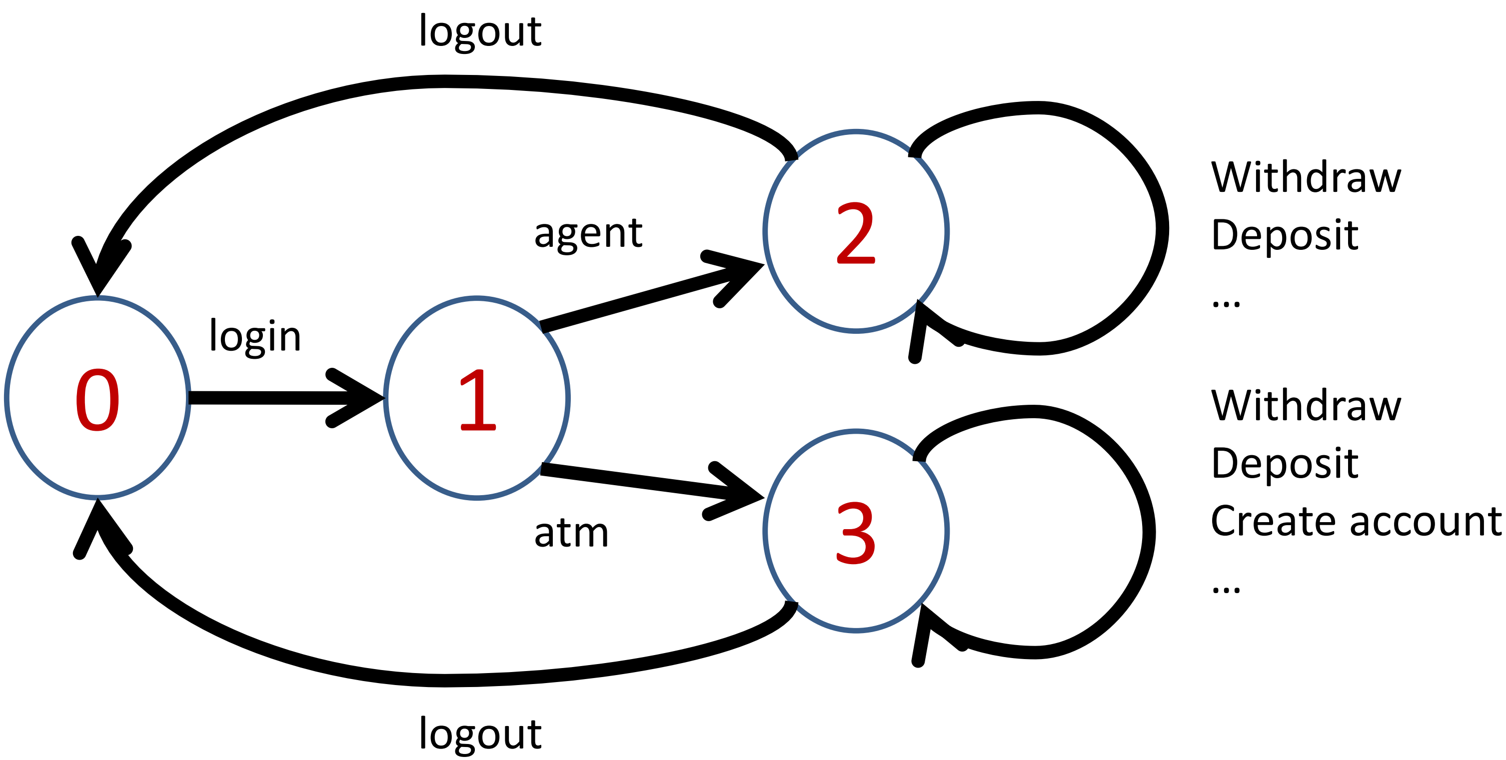
- We can generate tests to cover every state in the model, every state transition in the model, every path in the model, or so on
- In essence, this uses white-box coverage methods, but for the model rather than the code, automatically yielding complete, high-quality functionality tests

Model-Based Testing

- Example: We can cover all the states of our example model for login using only two tests:

T1: login
agent
logout

T2: login
atm
logout



Model-Based Testing

- Advantages:
  - Automatic test generation
  - Tests against a formal specification (the verified model)
  - Covers all essential behaviour
  - Still a black box method, with all its advantages
    - Requires only the model, not the code
  - Yields high confidence in the correctness of the final code

Model-Based Testing

- Disadvantages:
  - Heavyweight test method, probably only practical for safety-critical and security-critical applications (aerospace, automotive, etc.)

A1 Advice

- Make sure you include (ideally in your table of test cases, but if not there, in a file) the **actual terminal input**:

  login

  atm

  logout
- not just “Log in as planner”

A1 Advice

- Make sure you include (ideally in your table of test cases, but if not there, in a file) the **actual expected Transaction Summary File**, including the EOS mentioned in the requirements
- For "error cases" (negative tests), you **still** need to give the expected Transaction Summary File (usually, a file with only the EOS "transaction")
  - don't put "Error output" as the transaction summary; that's not what the requirements say to do

Summary…

- Black Box Testing
  - Output coverage methods analyze the set of possible outputs specified and create tests to cover them
  - Exhaustive output testing and output partitioning are similar but distinct from input coverage methods
  - Multiple input or output streams / files are handled by treating them as a predefined partitioning boundary

…Summary.

- Black Box Testing
  - We can also apply black box methods at lower levels of testing, if we have the architecture or detailed design
  - Model-driven engineering (MDE) can assist to automatically generate high quality tests using model-based testing