CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 13: Model-Based Testing

Gregory Gay
DIT636/DAT560 - March 3, 2025

Models and Software Analysis

- Before and while building products, engineers analyze models to address design questions.
- Software is no different.
- Software models capture different ways that the software ***behaves*** during execution.

Behavior Modeling

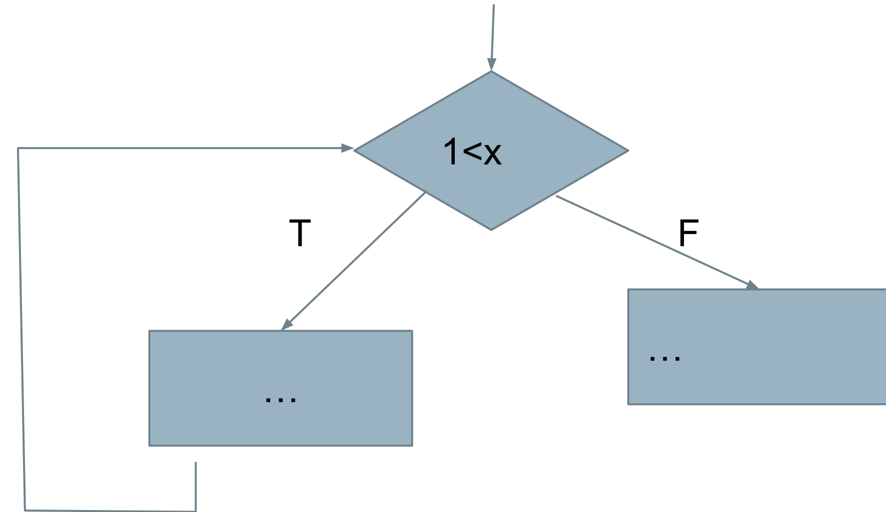
- **Abstraction** - simplify problem by identifying and focusing ***only*** on important aspects.
  - Solve a simpler problem, then apply to the big problem.
- A **model** is a simplified representation of the software-under-development.
  - Ignores all aspects irrelevant to the current task.

Software Models

- Abstractions of system being developed.
  - Only contain details relevant to a particular analysis.
- Can be extracted from specifications, design, code.
  - **Control and Data Flow**
    - Model of how control/data move during execution.
  - **Finite State Machines**
    - Events cause the system to react, changing its internal state.

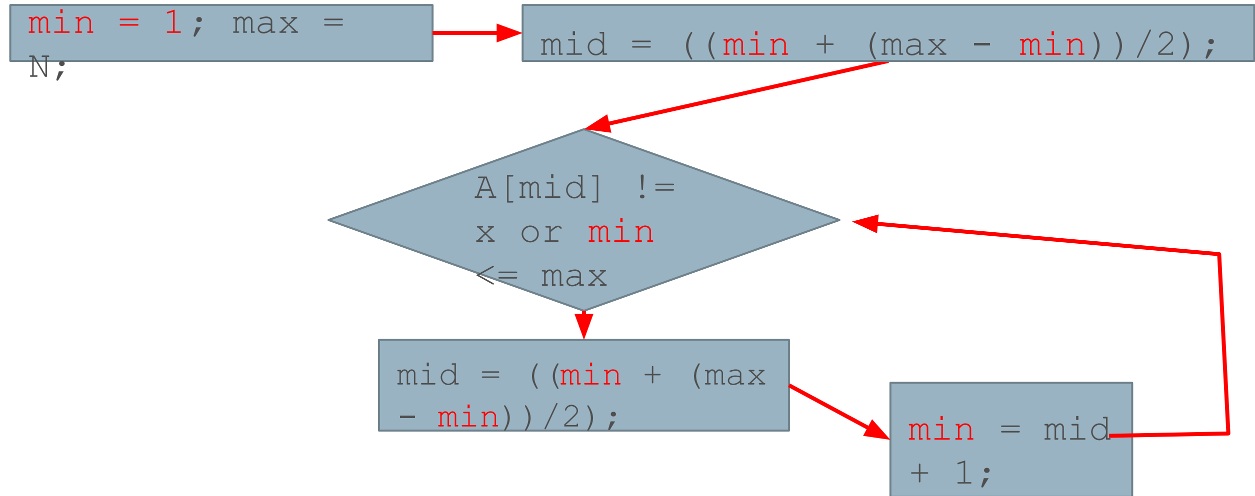
Control Flow Diagrams

- Model of how control flows between basic blocks.
  - Enables analyses and test creation centered around control flow.
- ***Omits all other information about the program.***



Data Flow Diagrams

- Model of how definitions and usages of variables are connected.
- ***Omits all other information about the program.***



Model-Driven Development

- State machine models often created during requirements analysis.
  - Allows refinement of requirements.
  - Can prove that requirements hold over model
    - (**Finite State Verification**)
- Can generate code from state machine models.
  - Used heavily in automotive, embedded.
- **Can create tests using models.**

Model-Based Testing

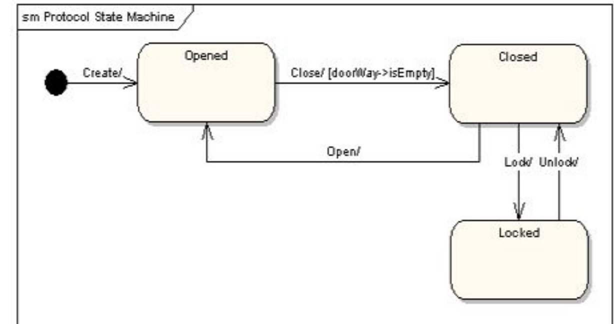
- State machine models describe (abstractly) what happens when input is applied to functionality.
- State machine model structure can be exploited:
    - Coverage criteria used to identify important paths.
    - Steps taken to perform functionality in different ways or to get different outcomes.

Finite State Machines

Finite State Machines

- "State" of software = values assigned to variables.
  - Set of all possible **real** behaviors is often infinite.
  - Called the "**state space**" of the program.
- **Models simplify a "functionality"/component into finite states.**
  - State = simple description or small set of variables
  - Execution modeled as **transitions between states, caused by actions**.

Finite State Machines



- Nodes represent states
  - Abstract description of the current value of an entity's attributes.
- Edges represent transitions
  - Events cause state to change.
  - Labeled `event [guard] / activity`
    - `event`: The event that triggered the transition.
    - `guard`: Conditions that must be true to transition.
    - `activity`: Output behavior when this transition is taken.

Terminology

- **Event** - An input that occurs at a defined time.
  - The user presses a button.
  - The alarm goes off.
- **Condition** - Internal or external property describing a change over time.
  - The fuel level has risen over a threshold.
  - The alarm has been on for ten seconds.

Terminology

- **State** - Abstract description of the current value of the entity’s attributes.
  - (e.g.: “Normal Operating Mode”, “Emergency Mode”)
  - Can also be current value of a set of variables.
    - However, keep that set small!
    - Limit possible variable values (e.g., “<0, 0, >0”, not “any integer”)

States, Transitions, and Guards

- States change in response to events (**transition**).
- When multiple transitions are possible, the choice is guided by the current conditions.
  - Also called the **guards** on a transition.
  - We take the transition that satisfies all guards.

State Transitions

Transitions labeled as:

```
event [guard] / activity
```

- `event`: The event that triggered the transition.
- `guard`: Conditions required to take this transition.
- `activity`: Output when this transition is taken.

State Transitions

```
event [guard] / activity
```

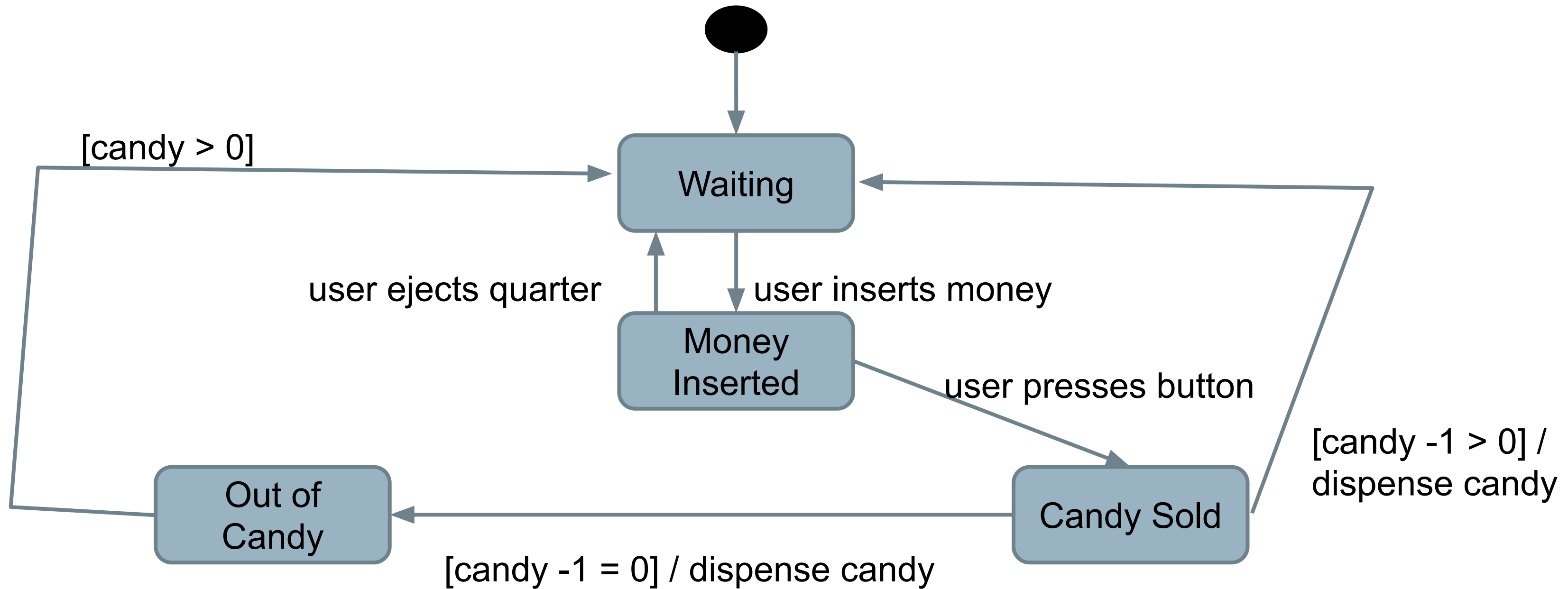
- All three are optional.
  - Missing Activity: No output from this transition.
  - Missing Guard: Always take transition following event.
  - Missing Event: Take this transition immediately after entering preceding state (if guards met).

State Transition Examples

```
event [guard] / activity
```

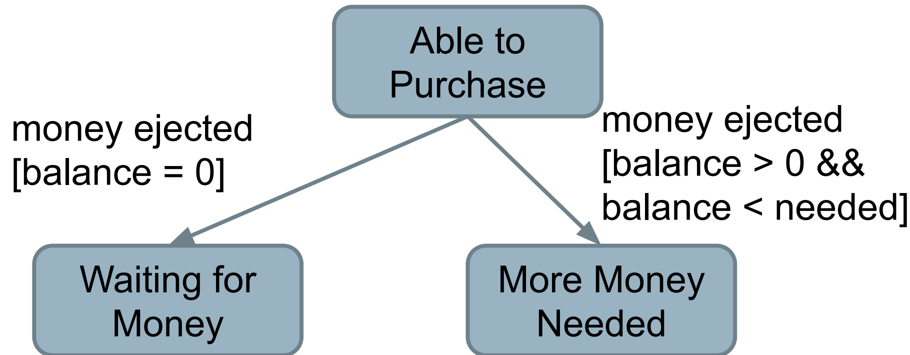
- Controller enters “self-test” mode after test button is pressed, leaves when reset button is pressed.
  - User pressing self-test, reset buttons are `events`.
- The tank enters “too-low” state when fuel level < threshold for N seconds.
  - Fuel level < threshold for N seconds is a `guard`.

Example: Candy Machine



More on Transitions

Guards must be mutually exclusive



If event occurs and no transition is valid, then event is ignored.

Missing transition for:

```
money [balance > 0 &&
balance >= needed]
```

Internal Activities

Can react to events and conditions without transitioning using internal activities.

Typing
entry / highlight all
exit / update field
character entered / add to field
help requested [verbose] / open help page
help requested [minimal] / update status bar

- Special events: **entry** and **exit**.
- Other activities occur each “time step”, until a transition occurs.
  - Entry and exit not re-triggered.

Example: Maintenance Tracking

- Customers send products for maintenance.
- **Maintenance tracking** notes current stage of process.
- **Model only what software tracks and controls!**

MaintenanceTracker
- status - warranty
request() estimateResponse(Bool) transfer() orderParts() return()

Example: Maintenance Tracking

If the product is covered by warranty or maintenance contract, maintenance can be requested through the software.

Waiting

No Warranty

If the product is not covered by warranty, the software informs the customer of the estimated cost. Maintenance starts when the customer accepts the estimate. If the customer does not accept, the item is returned.

Returning

Example: Maintenance Tracking

Under Local Repair

All repairs start at a local station. If the station cannot solve the problem, the product is sent to the main headquarters.

Repair at Main HQ

Maintenance is suspended if some components are not available.

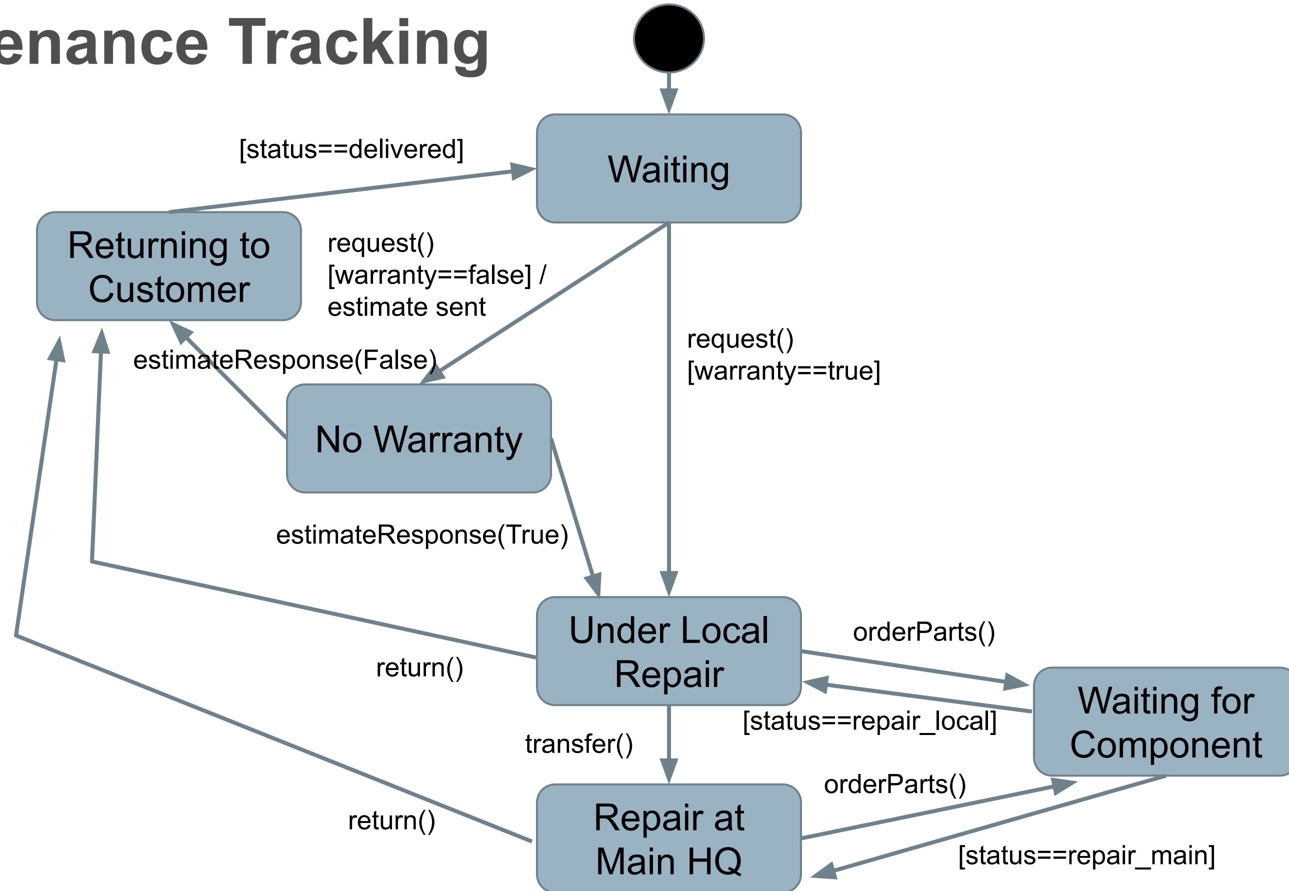
Waiting for Component

Once repaired, the product is returned to the customer.

Returning

Example: Maintenance Tracking

MaintenanceTracker
- status - warranty
request() estimateResponse(Bool) transfer() orderParts() return()



Example - Computer Model

- Many classes have stateful behavior.
  - States = class variables
  - Transitions = method calls
  - Derive model from class and create tests.
- We sell computers on our website. Model class represents a model of computer.
  - Models have slots for components (e.g., CPU, memory, video card).

Model
ModelID Slots
selectModel(modelID) deselectModel addComponent(slot, component) removeComponent(slot) isLegalConfiguration()

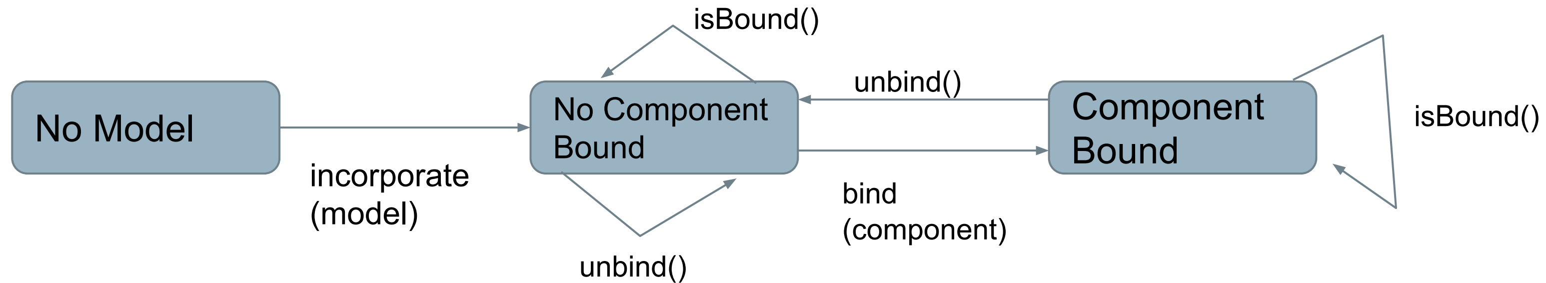
Slot
Model Component Required
incorporate(model) bind(component) unbind() isBound()

Slot Specification

Slot represents a configuration choice in all instances of a particular model of computer. A given model may have zero or more slots, each of which is marked as required or optional. If a slot is marked as required, it must be bound to a suitable component in all legal configurations. Slot offers the following methods:

- **Incorporate:** Make a slot part of a model, and mark it as either required or optional. All instances of a model incorporate the same slots.
- **Bind:** Associate a compatible component with a slot.
- **Unbind:** The unbind operation breaks the binding of a component to a slot, reversing the effect of a previous bind operation.
- **IsBound**: Returns true if a component is currently bound to a slot, or false if the slot is currently empty.

Slot State Machine



- Do not derive too many states.
  - Map variables to abstract values, not a state for each possible combination of values.
- Model how a method affects a class.
  - States only need to capture interactions between methods and the class state.

Example - Model

Model represents the current configuration of a model of computer.

- A given model may have zero or more slots, each of which is marked as required or optional.
- Each slot may contain a single component.
- To be a legal model, the model ID must exist in the ModelDB, each slot marked as required must be filled, the configuration must match that of the ModelDB entry for the model ID, and the optional components must match those allowed for that model in the ModelDB.

Example - Model

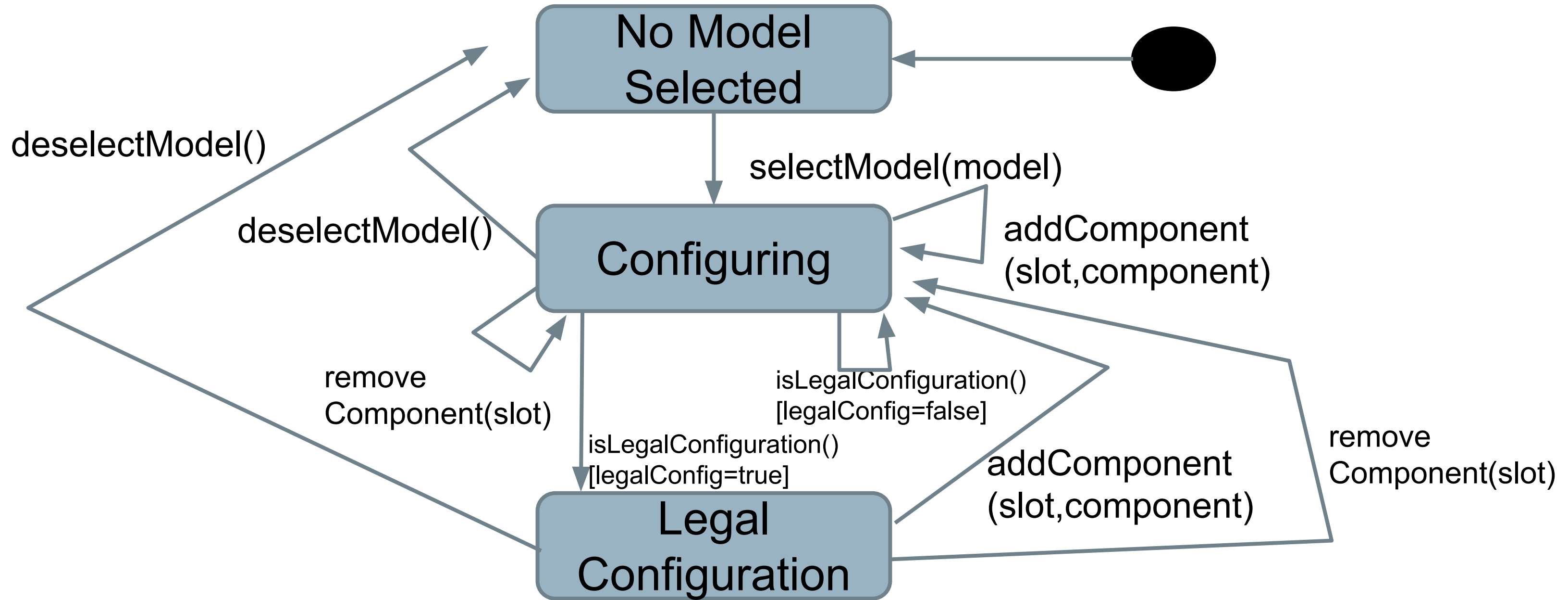
- **selectModel(modelID)**: Sets the model ID to the value passed in, as long as the model ID is set to “no model selected”. A model ID must be set before any other services are requested.
- **deselectModel():** Sets the model ID to “no model selected”. If the configuration was previously judged to be legal, it is no longer legal.
- **addComponent(slot, component):** Adds the selected component to the selected slot. If the configuration was previously judged to be legal, it is no longer legal.
- **removeComponent(slot)**: Removes the selected component to the selected slot. If the configuration was previously judged to be legal, it is no longer legal.
- **isLegalConfiguration():** Compares the current configuration to the entry in ModelDB. If the configuration is valid, the Model’s isLegal field is set to “true”.

Choosing States

No Model Selected | Configuring | Legal Configuration

- What does the class represent?
  - e.g., a computer model.
- What causes method results to differ?
  - e.g., whether the model is legal or illegal.
- Can the class be in any other states?
  - e.g., we may not have set the model yet, we could still be making decisions and have not determined legality.

Choosing Transitions and Initial State



Let’s Take a Break

Activity - Safe Lock Controller

You must design a state machine for a class that controls the lock on a safe.

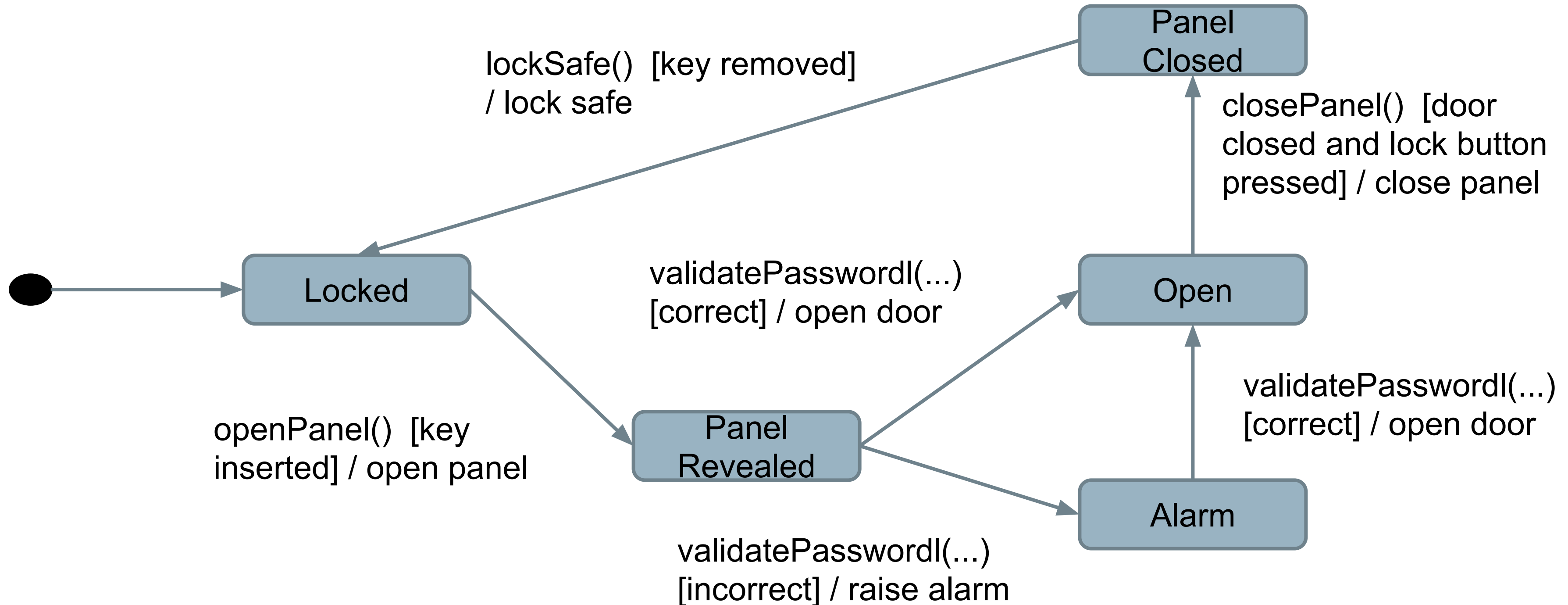
- To unlock the safe, a user must first insert a physical key. The software will then issue a command to open a panel, where a user will then enter a password.
  - If the password is correct, the lock will be released and the safe will open.
  - If the password is incorrect, an alarm will be raised. To stop the alarm, the user must enter the correct password.
- To relock the safe, the user must close the door and press the “lock” button on the keypad. The panel will close. The user may then remove their key. This will complete the locking process.

Activity - Safe Lock Controller

You must design a state machine for a class that controls the lock on a safe.

Method	Description
openPanel()	Checks that the key is inserted and opens the panel if it is.
validatePassword (password)	Checks whether the password is correct.
closePanel()	Closes the panel, as long as the door is closed and the lock button has been pressed.
lockSafe()	Locks the safe, as long as the panel has been closed.

Activity Solution



Model Coverage Criteria

Test Creation

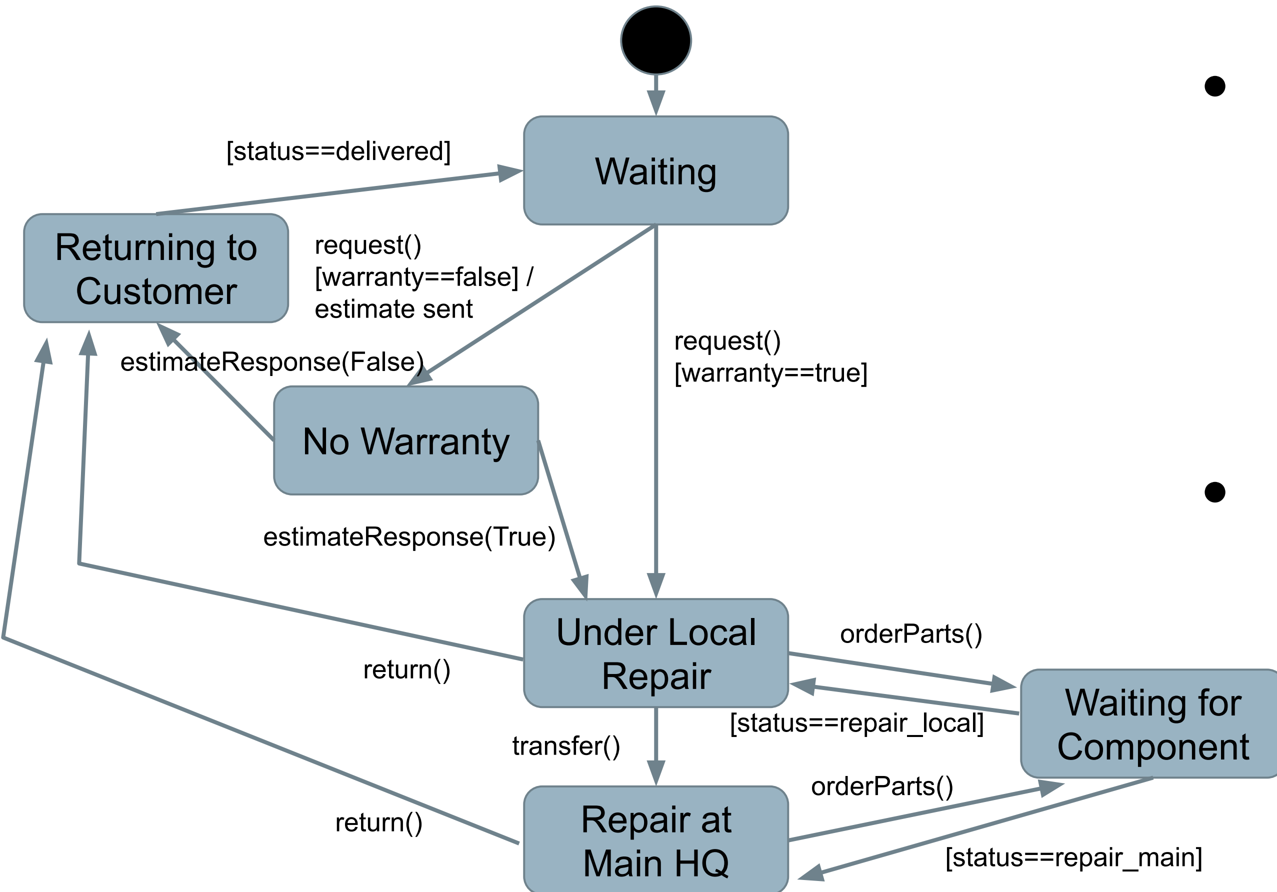
- Tests from models can be applied to the program.
  - Events translated into method/API calls.
  - Program output (abstracted) should match model output.
- Model coverage maps to requirements coverage.
  - Tests should be effective for verification.
  - Exercises stateful behavior thoroughly.
  - Coverage criteria based on states, transitions, paths.

State Coverage

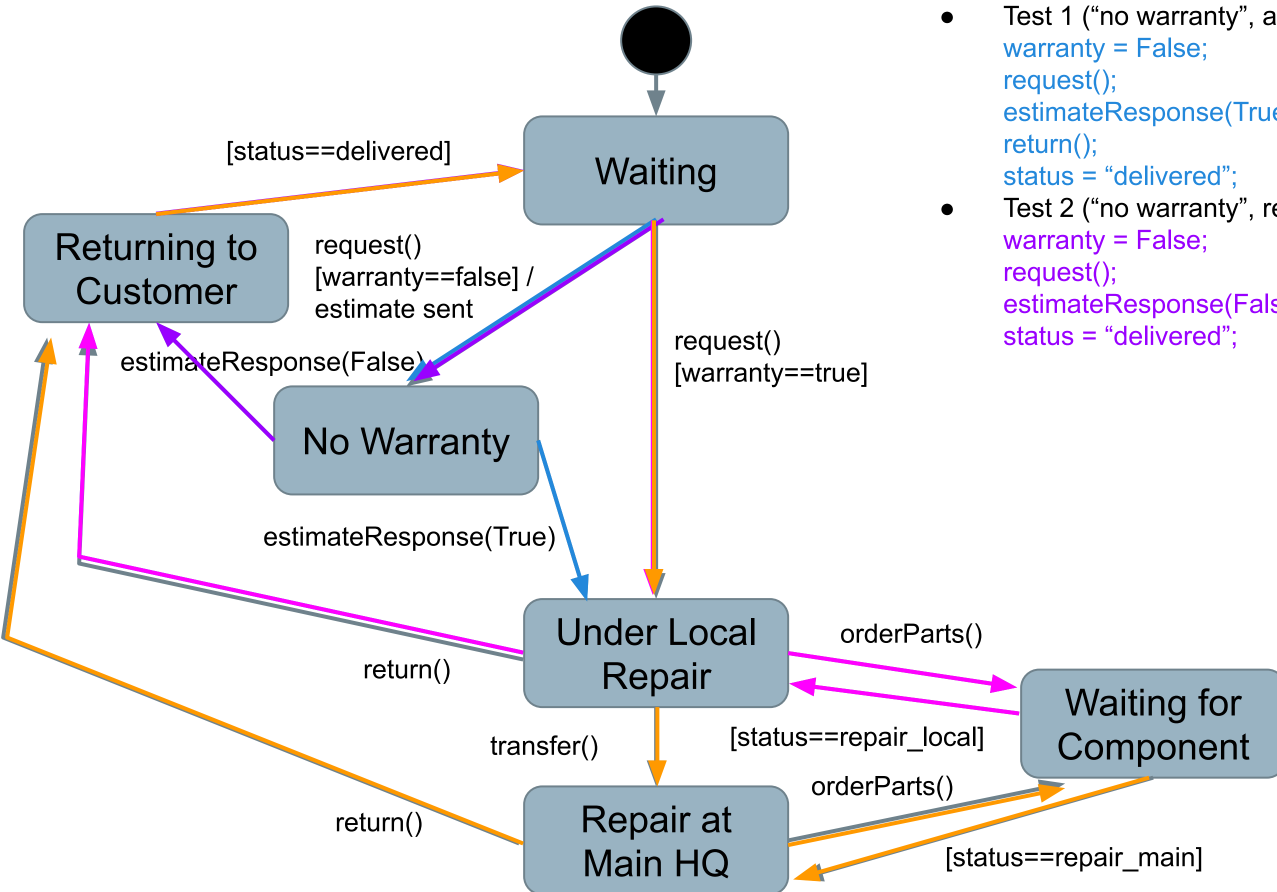
- **Each state must be reached by test cases.**
  - **Num. of Covered States / Number of States**
- Easy to understand and obtain, but low fault-revealing power.
  - Software takes action during transitions
  - Most states can be reached through multiple transitions.

Transition Coverage

- A transition specifies a pre/post-condition.
  - “If system is in state S and sees event I, then after reacting to it, the system will be in state T.”
  - Faulty system could violate (pre, post-condition) pairs.
- Every transition must be covered by test cases.
  - **Num. Covered Transitions / Number of Transitions**

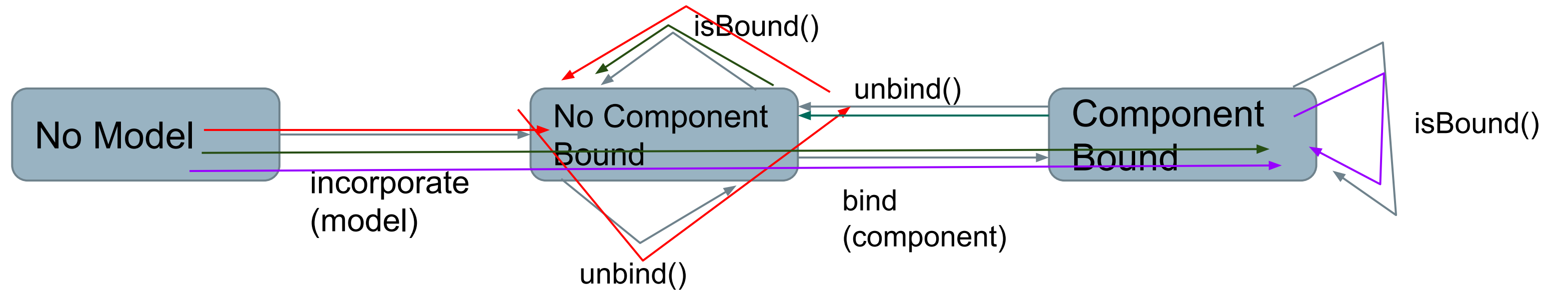


- If no “final” states, we could achieve transition coverage with one large test case.
    - Smarter to target sections in different test cases.
- Map input to method calls or variable assignments.



- Test 1 (“no warranty”, accept)
  warranty = False;
  request();
  estimateResponse(True);
  return();
  status = “delivered”;
- Test 2 (“no warranty”, reject)
  warranty = False;
  request();
  estimateResponse(False);
  status = “delivered”;
- Test 3 (Local Repair)
  warranty = True;
  request();
  orderParts();
  status = “repair_local”;
  return();
  status=”delivered”;
- Test 4 (Main HQ Repair)
  warranty = True;
  request();
  transfer();
  orderParts();
  status = “repair_main;
  return();
  status=”delivered”;

Example - Slot

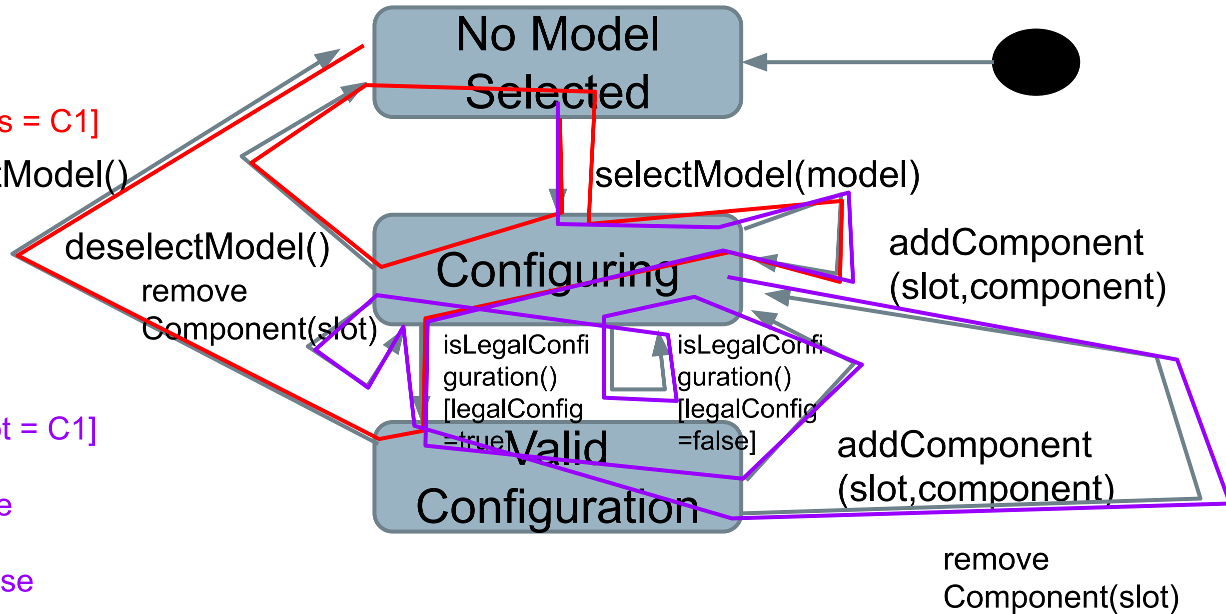


- incorporate(model), isBound(), unbind()
- incorporate(model), bind(component), isBound()
- incorporate(model), bind(component), unbind(), isBound()

Example - Model

TC1:
selectModel(M1) [M1, 1 slots = C1]
deselectModel()
selectModel(M1)
addComponent(S1,C1)
isLegalConfiguration() //true
deselectModel()

TC2:
selectModel(M1) [M1, 1 slot = C1]
addComponent(S1,C1)
isLegalConfiguration() //true
addComponent(S2,C2)
isLegalConfiguration() // false
removeComponent(S2)
isLegalConfiguration() // true
removeComponent(S1)



Path Coverage Criteria

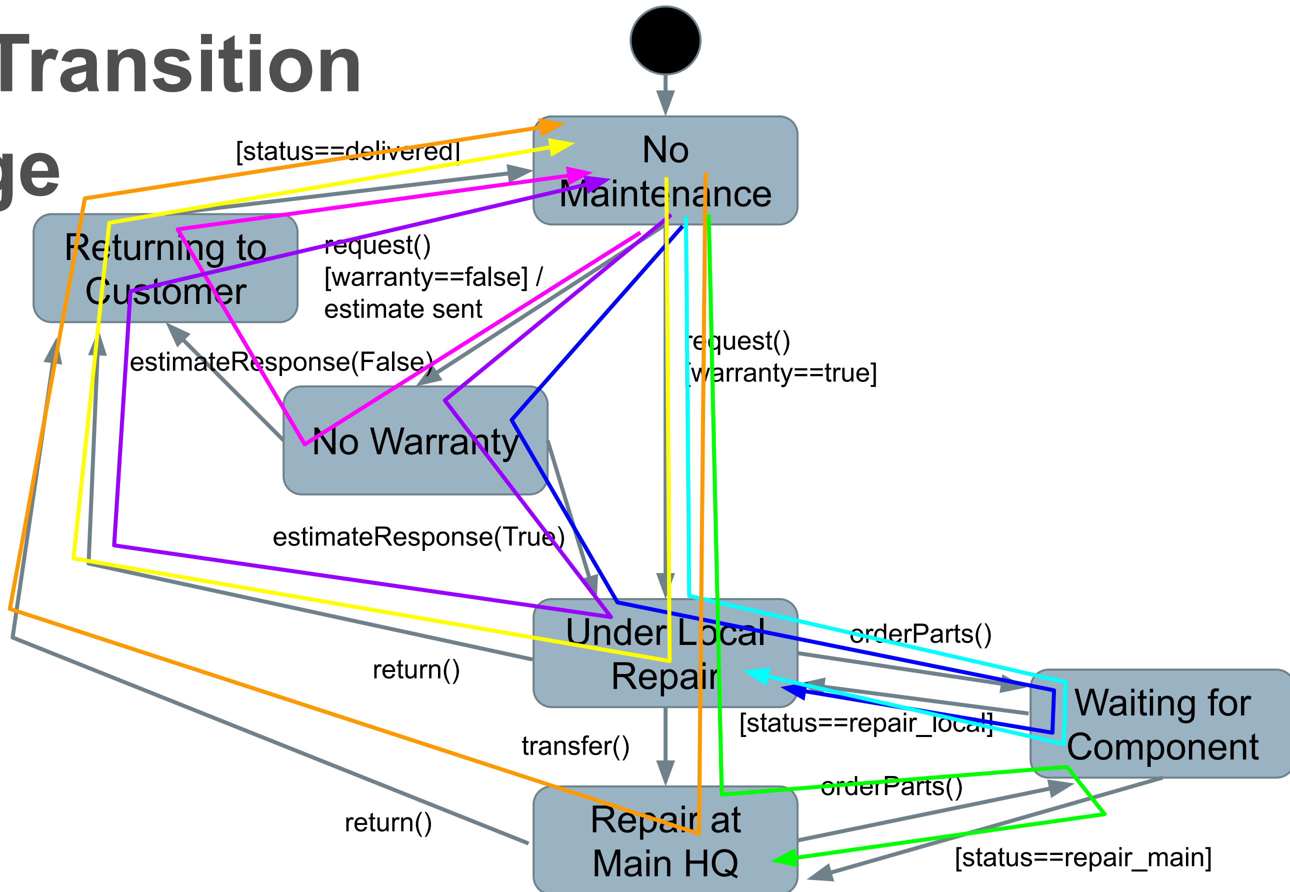
- Transition coverage based on assumption that transitions are independent.
- Many machines exhibit "history sensitivity".
  - Transitions available depend on path taken.
    - "wait for component" in Maintenance Tracking example.
- Path-based metrics can cope with sensitivity.

Path Coverage Metrics

- Single State Path Coverage
  - Requires that each subpath that traverses states at most once to be included in a path that is exercised.
- Single Transition Path Coverage
  - Requires that each subpath that traverses a transition at most once to be included in a path that is exercised.
- Boundary Interior Loop Coverage
  - Each distinct loop must be exercised minimum, an intermediate, and a large number of times.

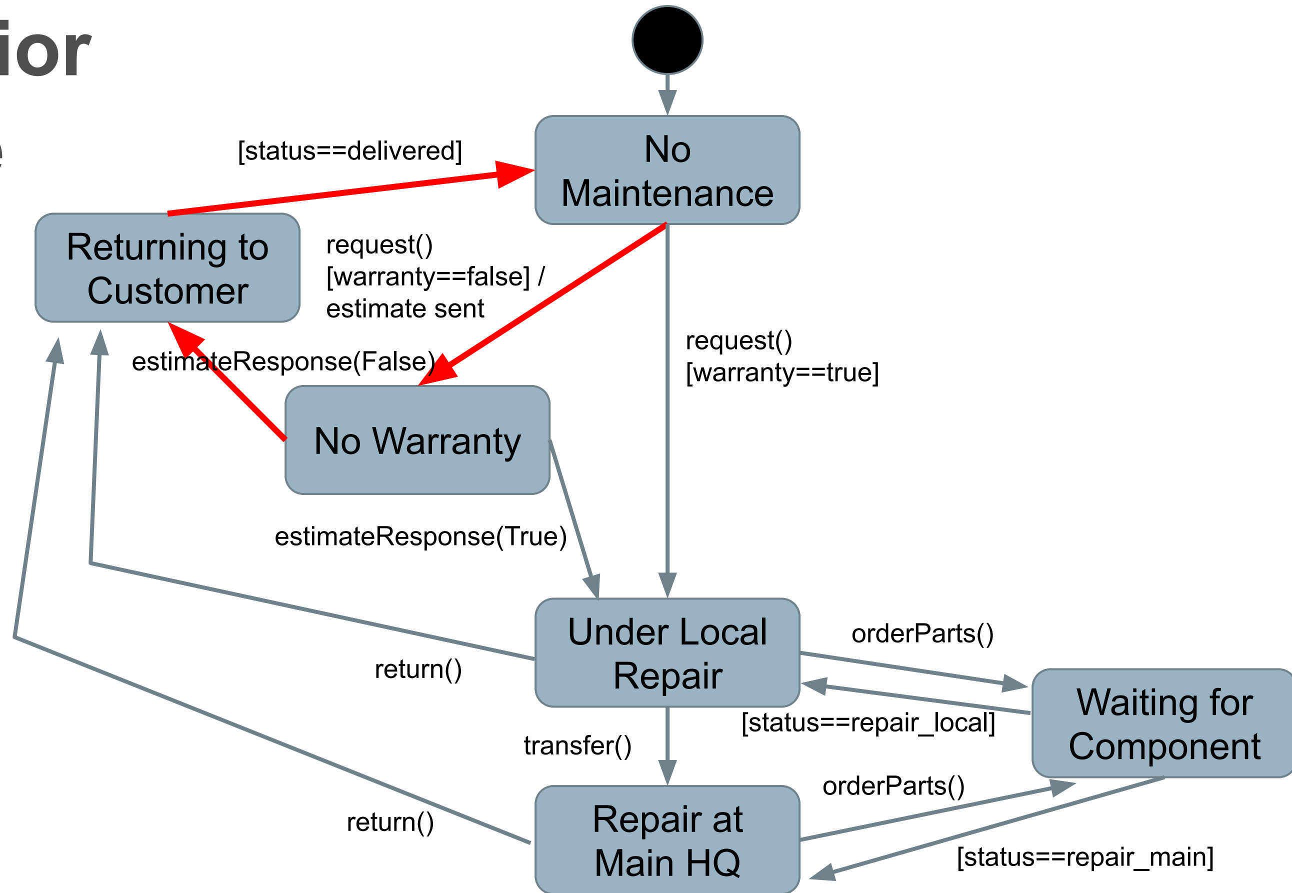
Single State/Transition Path Coverage

- Each subpath that traverses a state (or transition) **at most once** must be exercised.



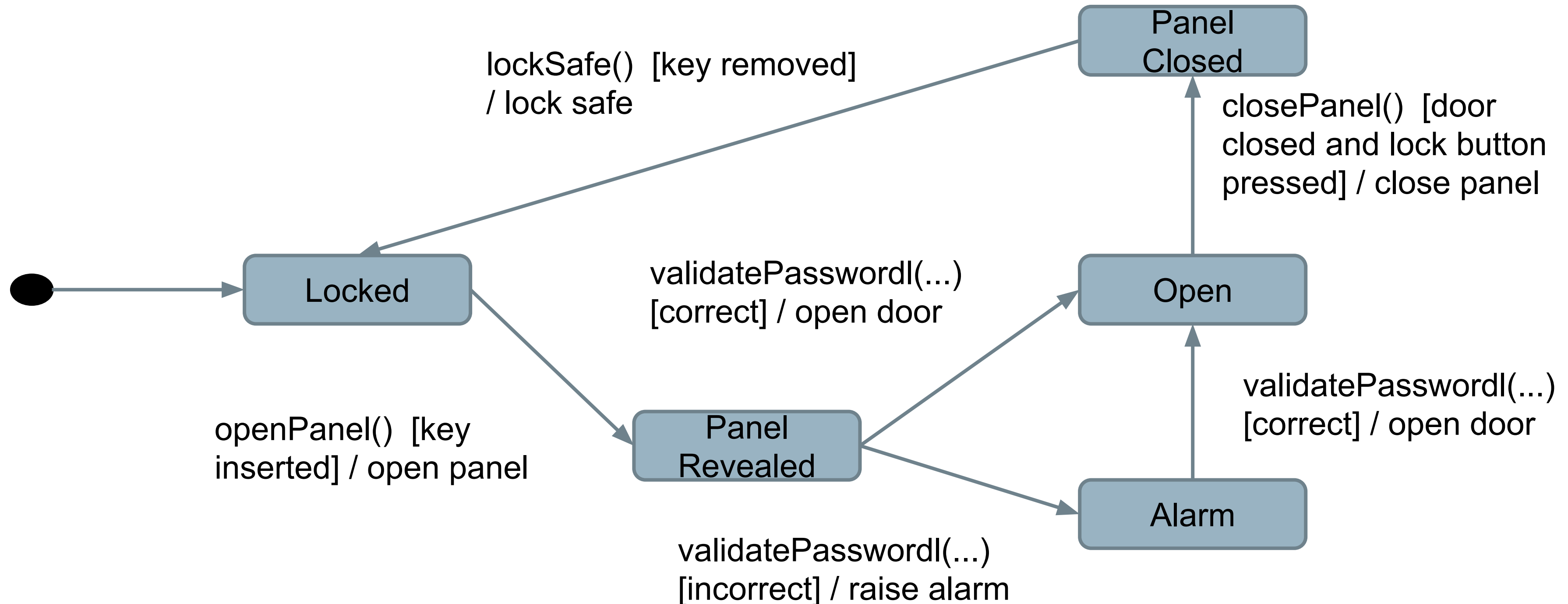
Boundary Interior Loop Coverage

- Each loop must be exercised 1, 2, N times.
- (N = some higher number)



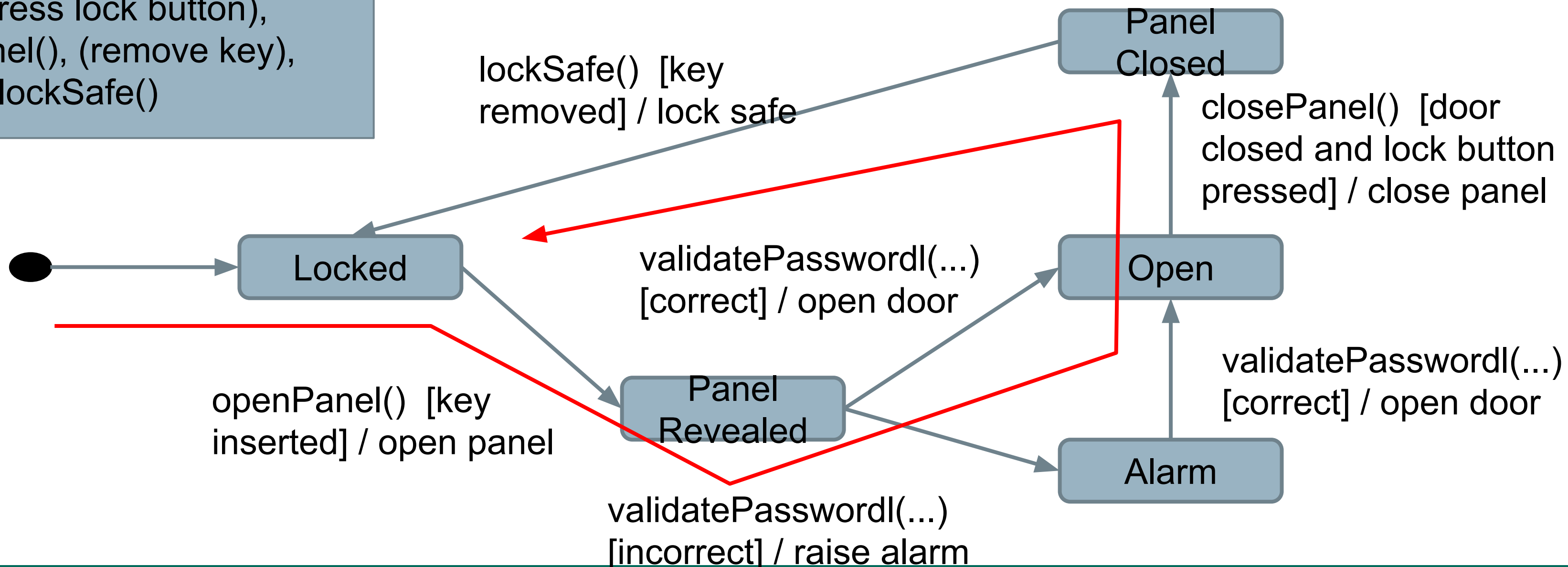
Activity

For the safe lock model, derive test suites that achieve state and transition coverage.



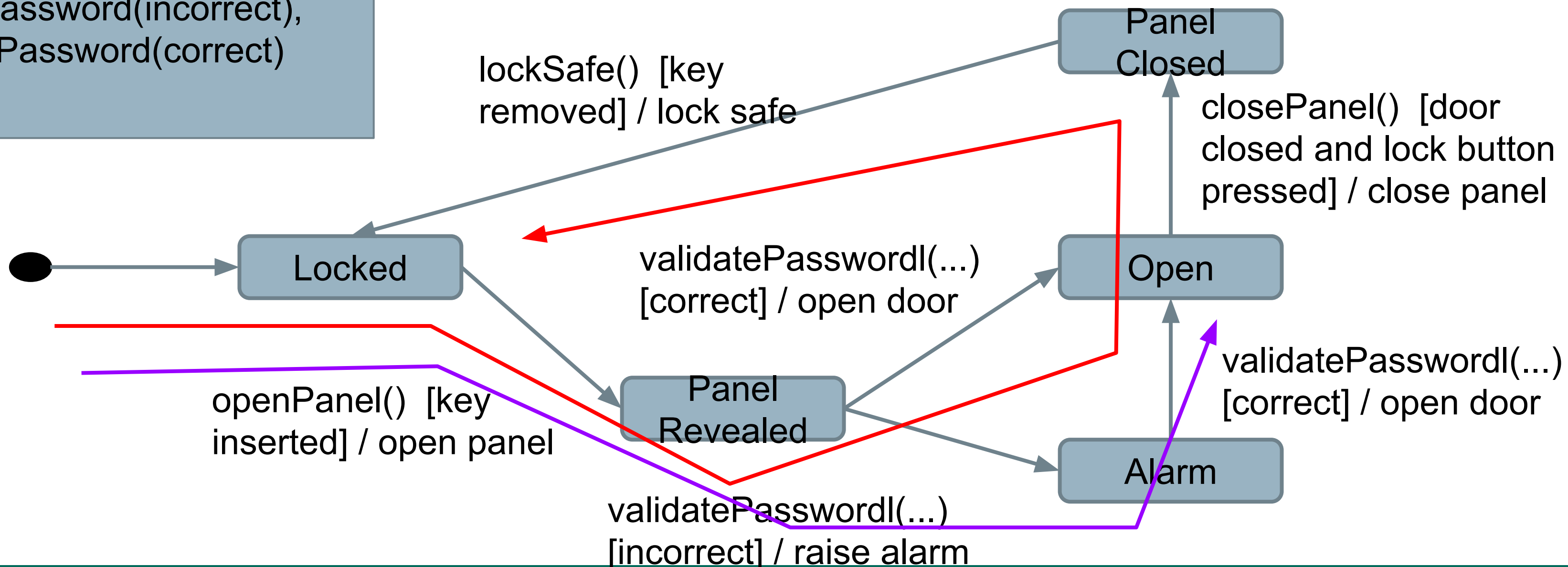
Activity Solution

Test 1: "Standard Path"
(insert key), openPanel(), validatePassword(correct), (close door, press lock button), closePanel(), (remove key), lockSafe()



Activity Solution

Test 2: "Trigger Alarm"
(insert key), openPanel(), validatePassword(incorrect), validatePassword(correct)



Activity Solution - Additional Tests

- These two tests achieve state and transition coverage, but do not verify all outcomes.
  - Also test alternate outcomes where guards are not met.
    - Key not inserted.
    - Door not closed.
    - Lock button not pressed.

We Have Learned

- Models can be used to systematically create tests.
  - Exercises stateful behavior of a class or functionality.
  - Maps well to requirements.
- State machines model expected behavior.
  - Cover states, transitions, non-looping paths, loops.
  - Can also verify properties over models as part of verification (next class).

Next Time

- Finite State Verification

- Assignment 4
  - Due Friday, March 14
  - Questions?

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY