CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 13: Model-Based Testing

Gregory Gay

DIT635 - March 4, 2020

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.

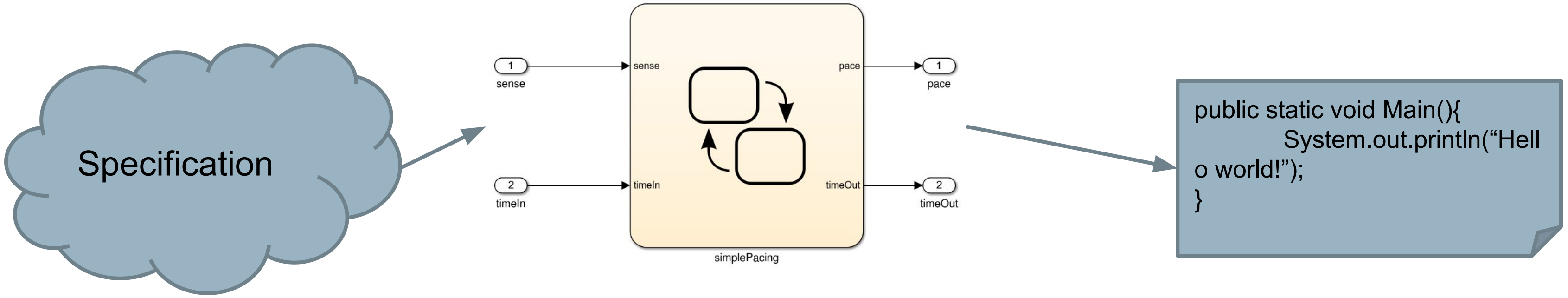
Behavior Modeling

- **Abstraction** - simplify a problem by identifying and focusing on important aspects while ignoring all other details.
- Key to solving *many* computing problems.
  - Solve a simpler version, then apply to the big problem.
- A **model** is a simplified representation of an artifact, focusing on one facet of that artifact.
  - The model ignores *all* other elements of that artifact.

Software Models

- A **model** is an abstraction of the system being developed.
  - By abstracting away unnecessary details, extremely powerful analyses can be performed.
- Can be extracted from specifications and design plans
  - Illustrate the *intended* behavior of the system.
  - Often take the form of state machines.
    - Events cause the system to react, changing its internal state.

What Can We Do With This Model?



```
public static void Main(){
        System.out.println("Hell
o world!");
}
```

If the model satisfies the specification...

And If the model is well-formed, consistent, and complete.

And If the model accurately represents the program.

… Then we can derive test cases from the model that can be applied to the program. If the model and program do not agree, then there is a fault.

Model-Based Testing

- Models describe the *structure* of the input space.
  - They identify what will happen when types of input are applied to the system.
- That structure can be exploited:
  - Identify input partitions.
  - Identify constraints on inputs.
  - Identify significant input combinations.
- Can derive and satisfy coverage metrics for certain types of models.

Model Properties

To be useful, a model must be:

- Compact
  - Models must be simplified enough to be analyzed.
  - “How simple” depends on how it will be used.
- Predictive
  - Represent the real system well enough to distinguish between good and bad outcomes of analyses.
  - No single model usually represents all characteristics of the system well enough for all types of analysis.

Model Properties

To be useful, a model must be:

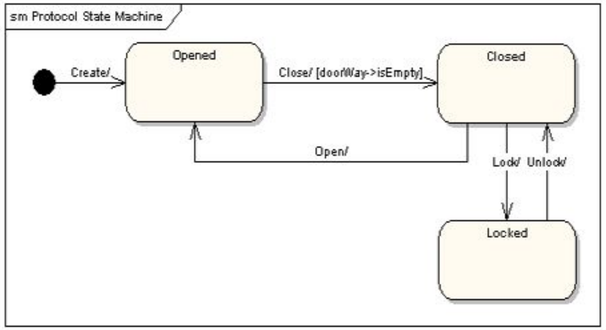
- Meaningful
  - Must provide more information than success and failure. Must allow diagnoses of the causes of failure.
- Sufficiently General
  - Models must be practical for use in the domain of interest.
  - An analysis of C programs is not useful if it only works for programs without pointers.

Finite State Machines

Finite Abstraction

- A program execution can be viewed as a sequence of states alternating with actions.
- Software “behavior” is a sequence of state-action-state transitions.
- The set of all possible behaviors is often infinite.
  - Called the “state space” of the program.
  - Models of execution are finite abstractions (simplifications) of the full program’s state space.

Finite State Machines



- A directed graph.
- Nodes represent states
  - An abstract description of the current value of an entity's attributes.
- Edges represent transitions between states.
  - Events cause the state to change.
  - Labeled `event [guard] / activity`
    - `event`: The event that triggered the transition.
    - `guard`: Conditions that must be true to choose a transition.
    - `activity`: Behavior exhibited by the object when this transition is taken.

Some Terminology

- **Event -** Something that happens at a point in time.
  - Operator presses a self-test button on the device.
  - The alarm goes off.
- **Condition** - Describes a property that can be true or false and has duration.
  - The fuel level is high.
  - The alarm is on.

Some Terminology

- **State** - An abstract description of the current value of an entity’s attributes.
  - The controller is in the “self-test” state after the self-test button has been pressed, and leaves it when the rest button has been pressed.
  - The tank is in the “too-low” state when the fuel level is below the set threshold for N seconds.

States, Transitions, and Guards

- States change in response to events.
  - A state change is called a **transition**.
- When multiple responses to an event (transitions triggered by that event) are possible, the choice is guided by the current conditions.
  - These conditions are also called the **guards** on a transition.

State Transitions

Transitions are labeled in the form:

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

- `event`: The event that triggered the transition.
- `guard`: Conditions that must be true to choose this transition.
- `activity`: Behavior exhibited by the object when this transition is taken.

State Transitions

Transitions are labeled in the form:

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

- All three are optional.
  - Missing Activity: No output from this transition.
  - Missing Guard: Always take this transition if the event occurs.
  - Missing Event: Take this transition immediately.

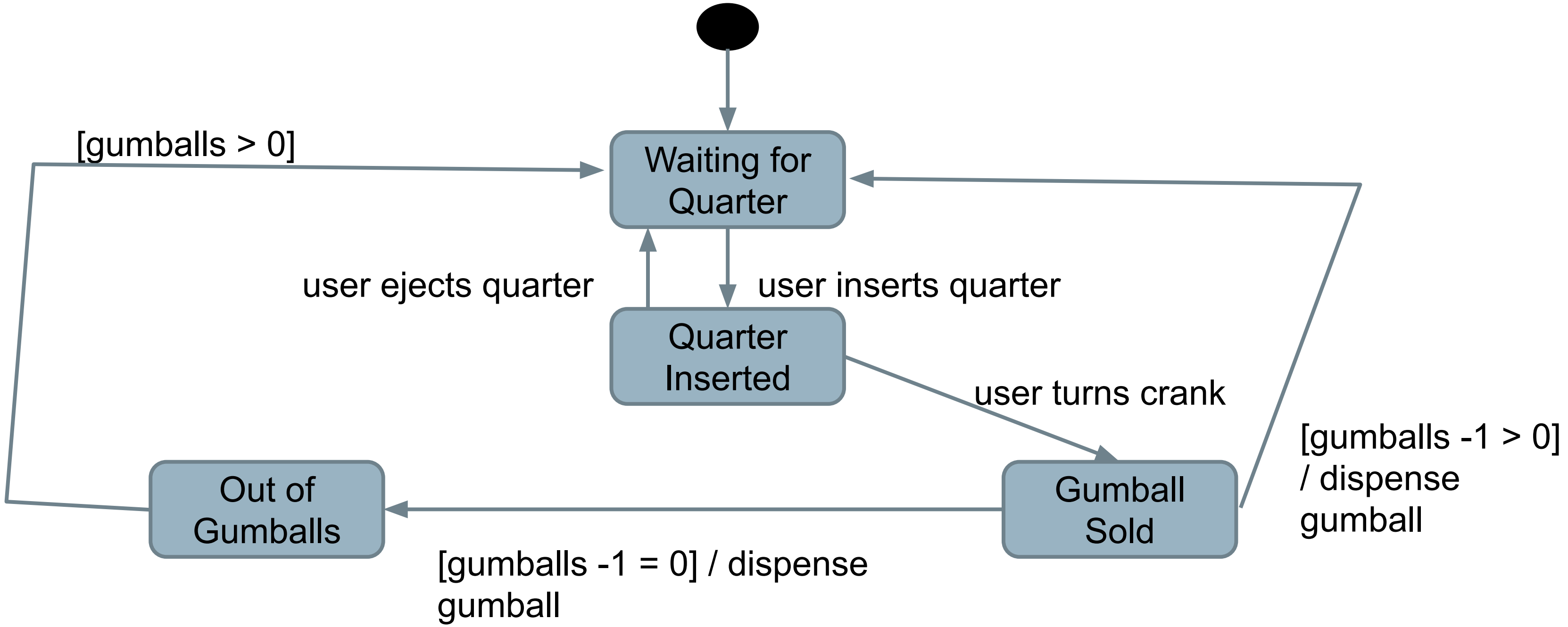
State Transition Examples

Transitions are labeled in the form:

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

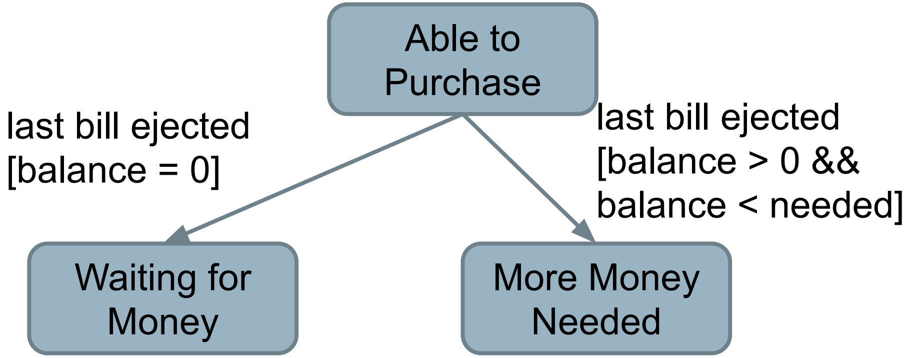
- The controller is in the "self-test" mode after the test button is pressed, and leaves it when the rest button is pressed.
  - Pressing self-test button is an **`event`**.
  - Pressing the rest button is an **`event`**.
- The tank is in the "too-low" state when fuel level is below the threshold for N seconds.
  - Fuel level below threshold for N seconds is a **`guard`**.

Example: Gumball Machine



More on Transitions

Guards must be mutually exclusive

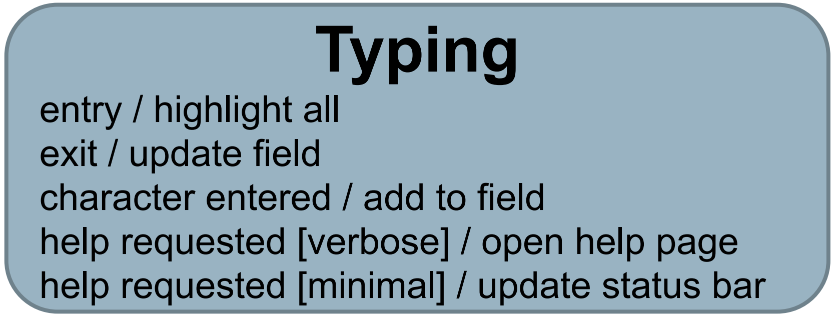

Able to Purchase
last bill ejected [balance = 0]
last bill ejected [balance > 0 && balance < needed]
Waiting for Money
More Money Needed

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

last bill ejected [balance > 0 && balance >= needed]

Internal Activities

States 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 until a transition occurs.
  - On each “time step”.
  - Entry and exit not re-triggered without a self-transition.

Example: Maintenance

If the product is covered by warranty or maintenance contract, maintenance can be requested through the web site or by bringing the item to a designated maintenance station.

No Maintenance

Waiting for Pick Up

If the maintenance is requested by web and the customer is a US resident, the item is picked up from the customer. Otherwise, the customer will ship the item.

Request - No Warranty

If the product is not covered by warranty or the warranty number is not valid, the item must be brought to a maintenance station. The station 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.

Wait for Acceptance

Wait for Returning

Example: Maintenance

Repair at Station

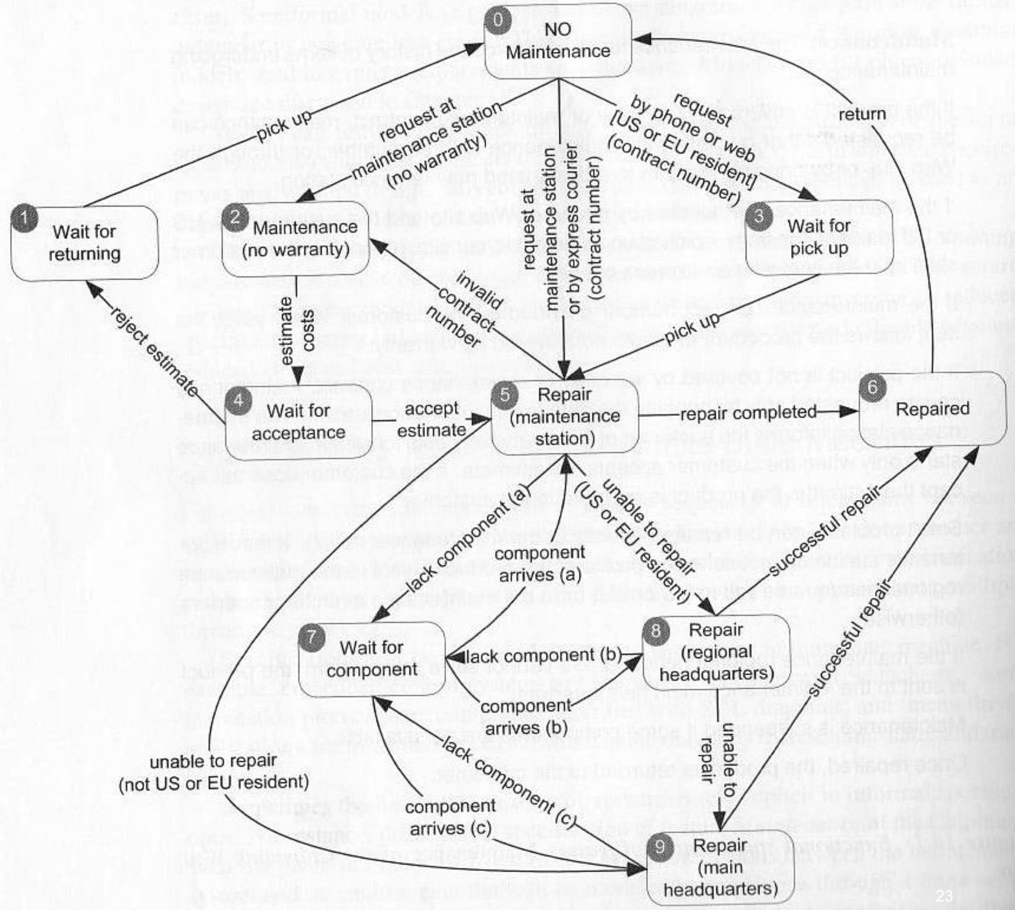
If the maintenance station cannot solve the problem, the product is sent to the regional headquarters (if in the US) or the main headquarters (otherwise). If the regional headquarters cannot solve the problem, the product is sent to main headquarters. Repair at Regional HQ Repair at Main HQ

Maintenance is suspended if some components are not available. Wait for Component

Once repaired, the product is returned to the customer.

Repaired

Example: Maintenance



Finite State Space

- Most systems have an *infinite* number of states.
  - For a communication protocol, there are an infinite number of possible messages that can be passed.
- Non-finite components must be ignored or abstracted until the model is finite.
  - For the communication protocol, the message text *doesn't matter*. How it is used does matter.
  - Requires an *abstraction function* to map back to the real system.

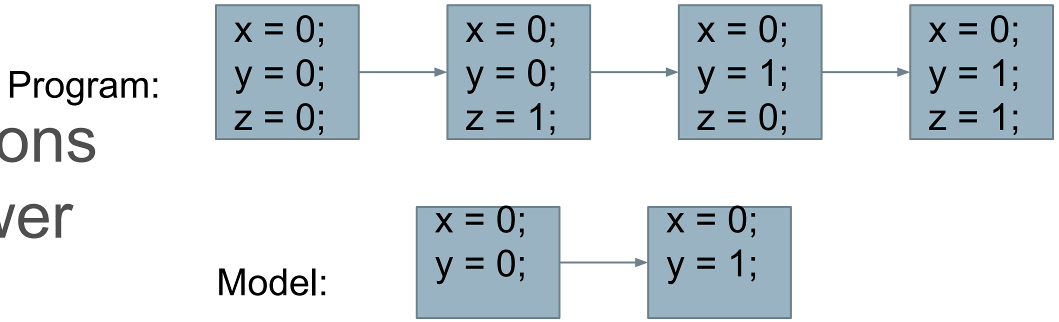
Abstraction Functions

- We can link a concrete state to a model state through an *abstraction function*.
  - Translates the real program to a model by stripping away details.
  - Groups states that only differ through details abstracted from the model.
  - This has two effects:
    - Sequences of transitions are collapsed into fewer execution steps.
    - Nondeterminism can be introduced.

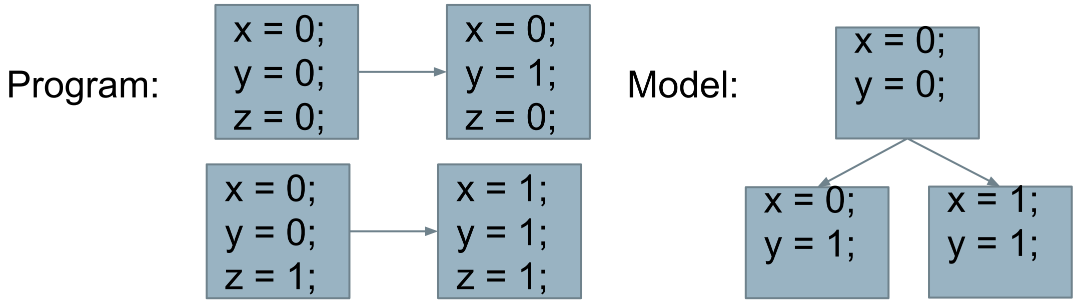
Abstraction Functions

This has two effects:

- Sequences of transitions are collapsed into fewer execution steps.



- Nondeterminism can be introduced.

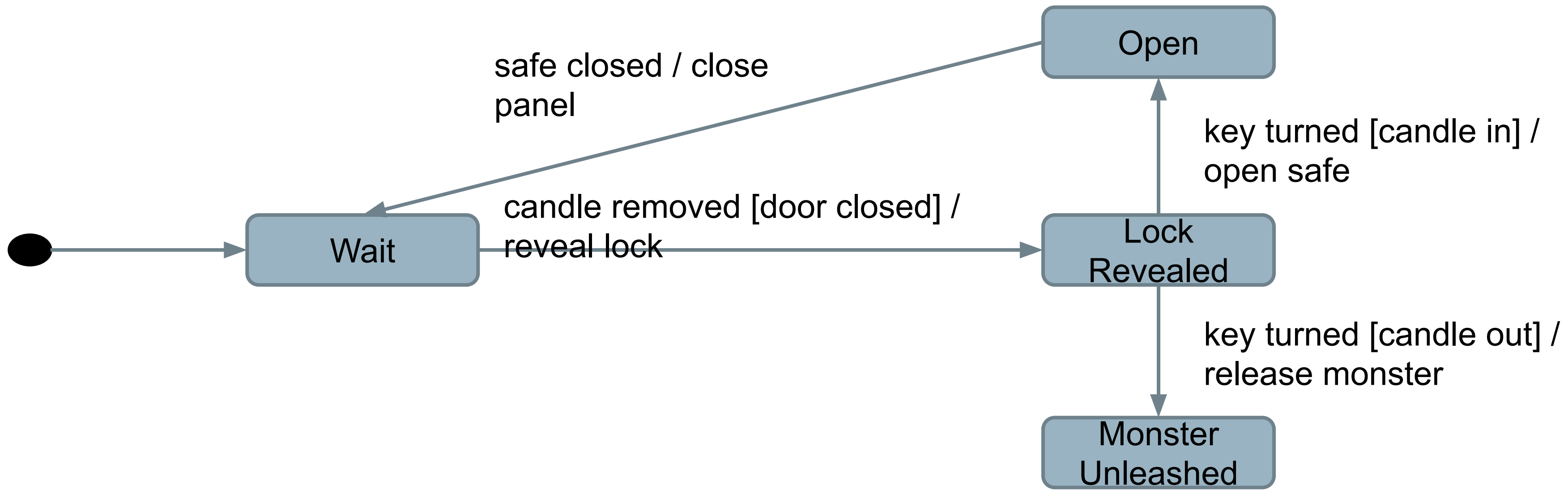


Activity - Secret Panel Controller

You must design a state machine for the controller of a secret panel in Dracula's castle.

Dracula wants to keep his valuables in a safe that's hard to find. So, to reveal the lock to the safe, Dracula must remove a strategic candle from its holder. This will reveal the lock only if the door is closed. Once Dracula can see the lock, he can insert his key to open the safe. For extra safety, the safe can only be opened if he replaces the candle first. If someone attempts to open the safe without replacing the candle, a monster is unleashed.

Activity Solution


Open
safe closed / close panel
key turned [candle in] / open safe
Wait
candle removed [door closed] / reveal lock
Lock Revealed
key turned [candle out] / release monster
Monster Unleashed

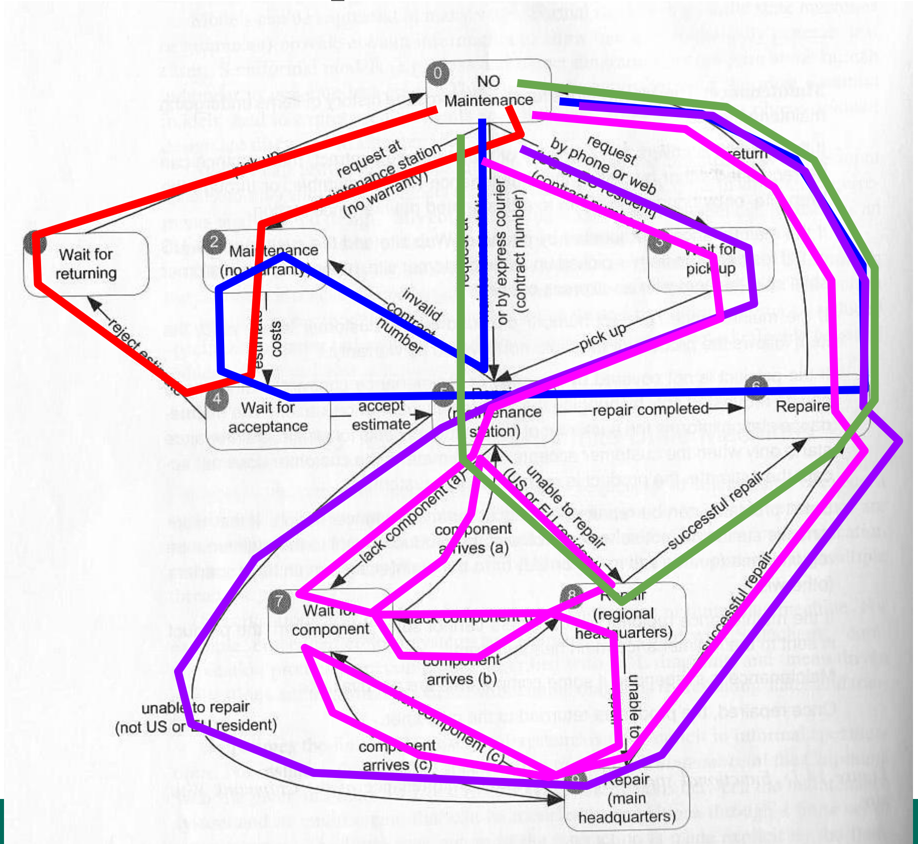
State Coverage

- Each state reached by one or more test cases.
- Like statement coverage - unless model has been placed in each state, faults cannot be revealed.
- Easy to understand and obtain, but low fault-revealing power.
    - The software takes action during the *transitions*, and most states can be reached through multiple transitions.

Transition Coverage

- A transition specifies a pre/post-condition.
  - "If the system is in state S and sees event I, then after reacting to it, the system will be in state T."
  - A faulty system could violate any of these precondition, postcondition pairs.
- Coverage requires that every transition be covered by one or more test cases.
  - Subsumes state coverage.

Example: Maintenance



- If no "final" states, we could achieve transition coverage with one large test case.
  - Smarter to break down FSM and target sections in isolation.

Example Suite:

T1: request w/ no warranty (0->2) - estimate costs (2->4) - reject (4->1) - pick up (1->0)

T2: 0->5->2->4->5->6->0

T3: 0->3->5->9->6->0

T4: 0->3->5->7->5->8->7->8->9->7->9->6->0

T5: 0->5->8->6->0

History Sensitivity

- Transition coverage based on assumption that transitions out of a state are independent of transitions into a state.
- Many machines exhibit "history sensitivity".
    - Transitions available depend on the *history* of previous actions.
    - AKA - the path to the current state.
    - Can be a sign of a bad model design.
        - "wait for component" in example.
    - Path-based metrics can cope with sensitivity.

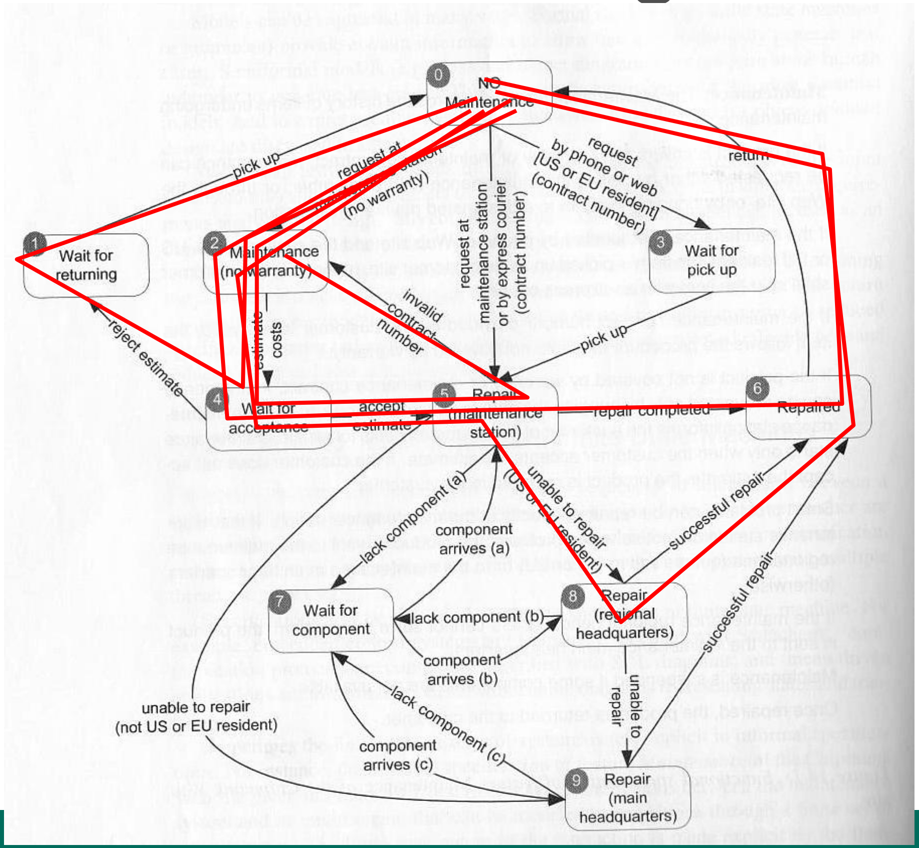
Path Coverage Metrics

- 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

Single State/Transition Path Coverage

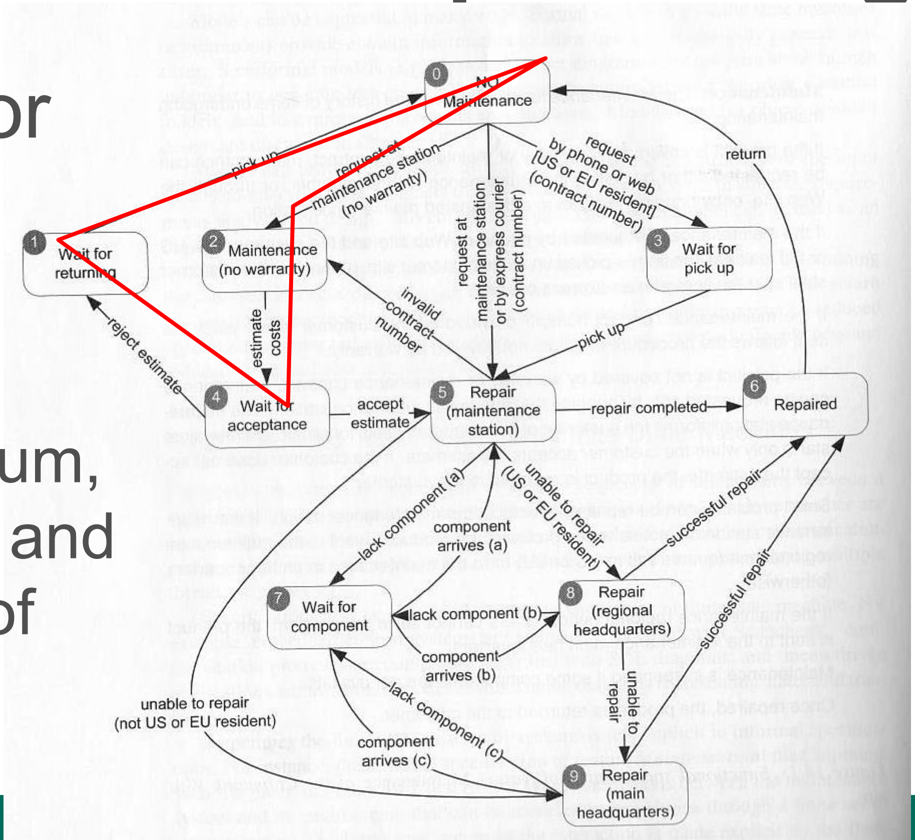
- Requires that each subpath that traverses states/transitions at most once to be included in a path that is exercised.



Boundary Interior Loop Coverage

Boundary Interior Loop Coverage

- Each distinct loop must be exercised minimum, an intermediate, and a large number of times.

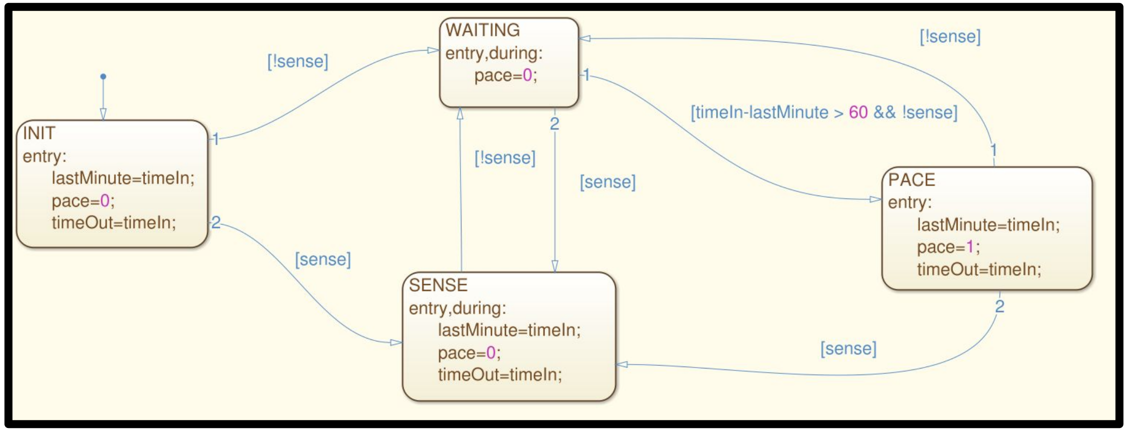


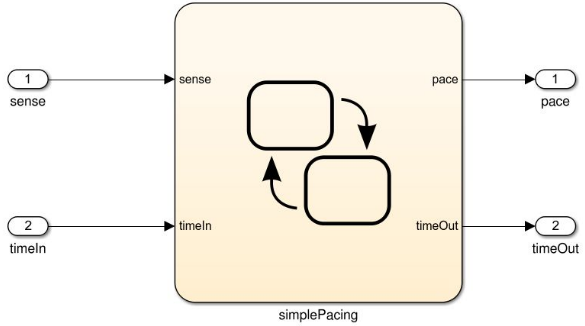
Test Generation

- Test cases created for models can be applied to programs.
  - Events can be translated into method input.
  - System output, when abstracted, should match model output.
- Model coverage is one form of requirements coverage. Tests should be effective for verification.

Activity

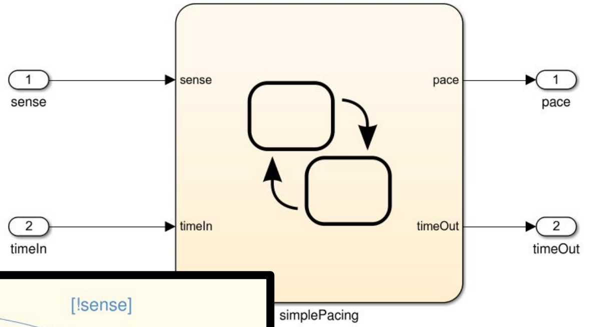
For this model, derive test suites that achieve state and transition coverage.

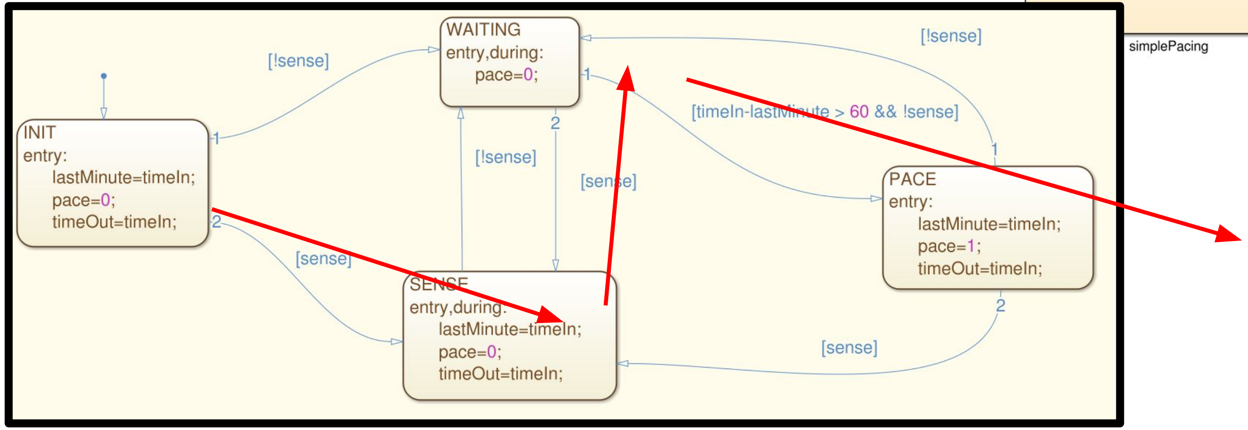




Activity - State Coverage

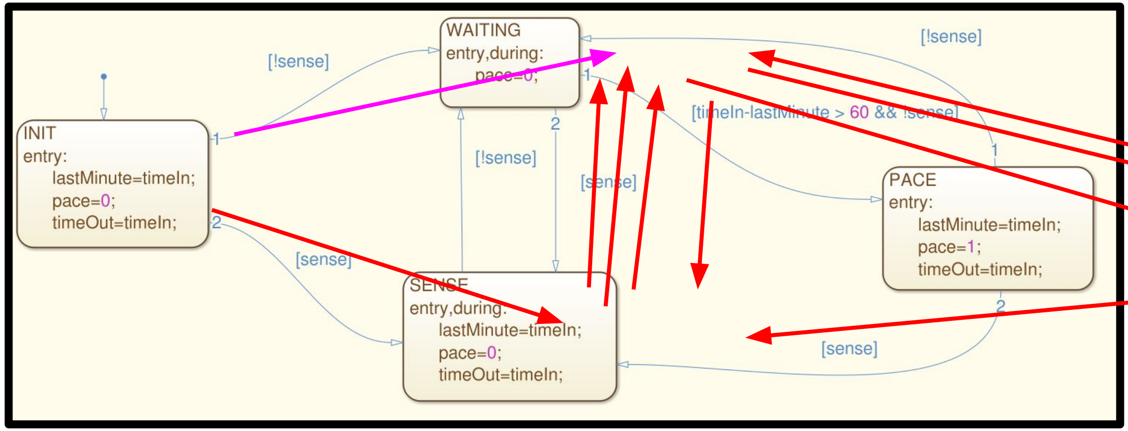
[true,1], [false,2], [false, 65]





Activity - Transition Coverage

1. [true,1], [false,2], [false, 65], [true, 66], [false, 77], [true, 78], [false, 79], [false, 140], [false, 141]
2. [false, 1]



We Have Learned

- If we build models from functional specifications, those models can be used to systematically generate test cases.
  - Models have structure. We can exploit that structure.
  - A form of functional testing.
- Helps identify important input.
- Coverage metrics based on the type of model guide test selection.

We Have Learned

- State machines model expected behavior.
  - Cover states, transitions, non-looping paths, loops.
  - Can also be used in finite state verification (next class)

Next Time

- Finite State Verification
  - Optional Reading - Pezze and Young, Chapter 8
- Homework 3
  - Due Friday, March 13
  - Questions?

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY