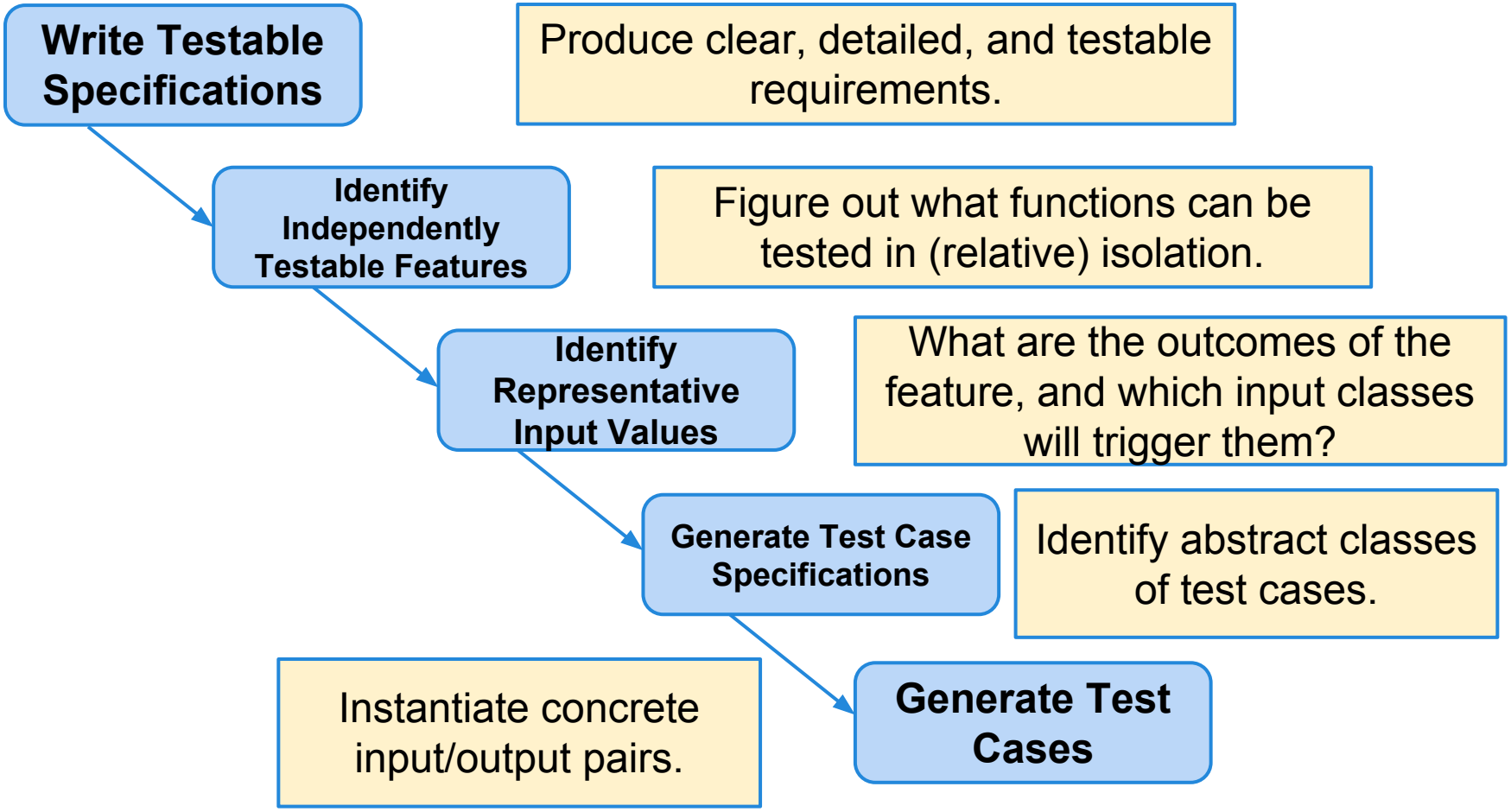
Model-Based Testing

CSCE 747 - Lecture 10 - 02/11/2016

Creating Requirements-Based Tests



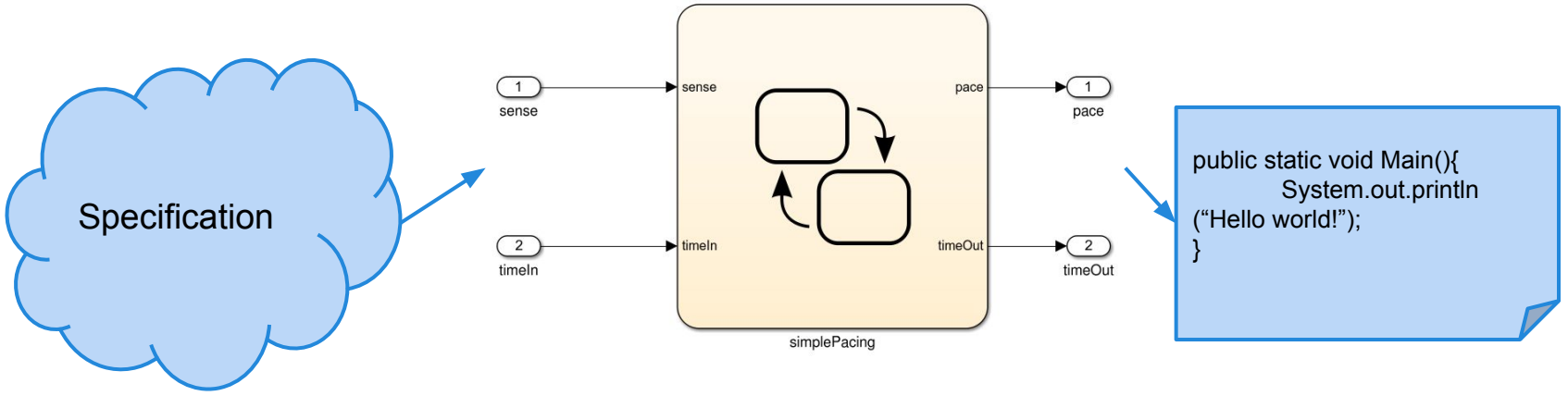
Creating Requirements-Based Tests

- This process is effective for identifying the independent partitions for each input.
  - Leaving us with a large number of test specifications
- Humans must still identify constraints on combinations of input choices and identify a subset of important test specifications.
- An alternative approach - build a model from the specification, and derive tests from the *structure* of the model.

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 the source code
  - Control-flow, data-flow diagrams
- Can be extracted from specifications and design plans
  - Illustrate the *intended* behavior of the system.

What Can We Do With This Model?



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.

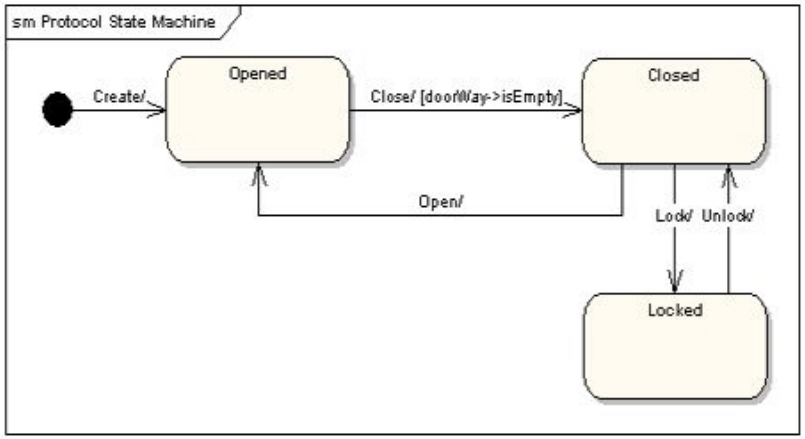
Model-Based Testing

- 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.

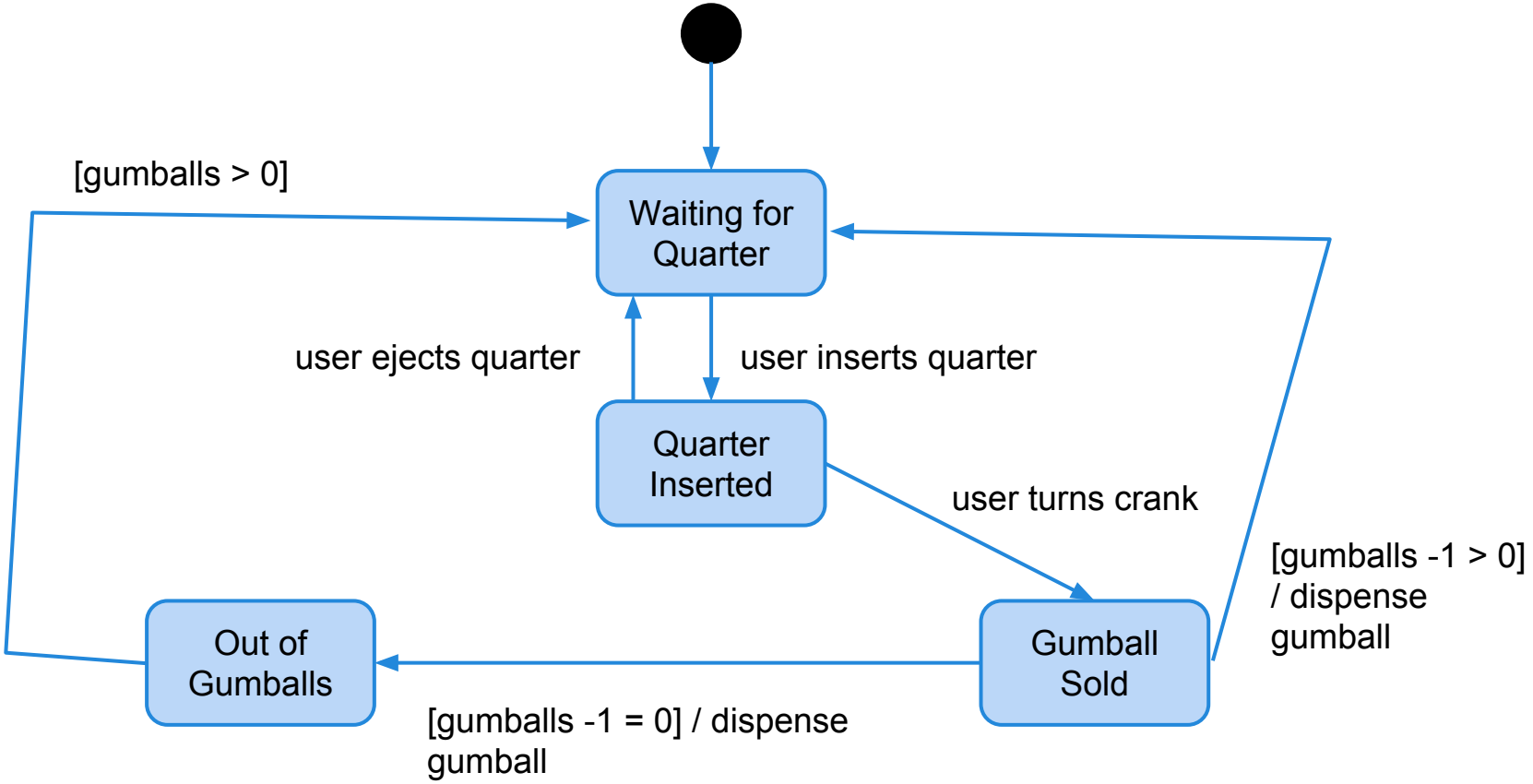
Finite State Machines

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.



Example: Gumball Machine



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

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

Waiting for Pick Up

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 c

Wait for Acceptance

Wait for Returning

Repair at Station

customer does not accept, the item is retur

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

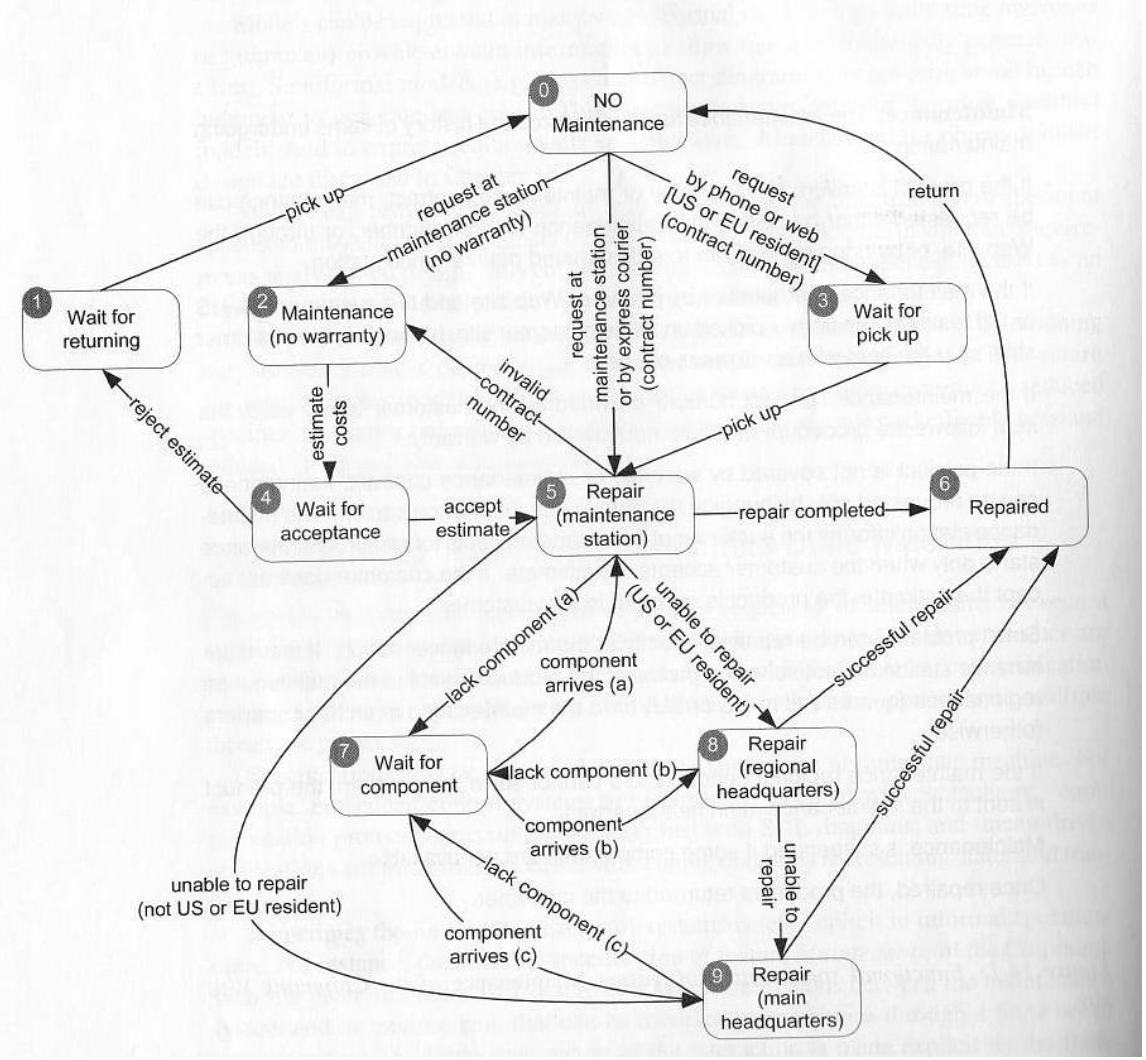
Maintenance is suspended if some components are not available.

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.
- To model such systems, 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.

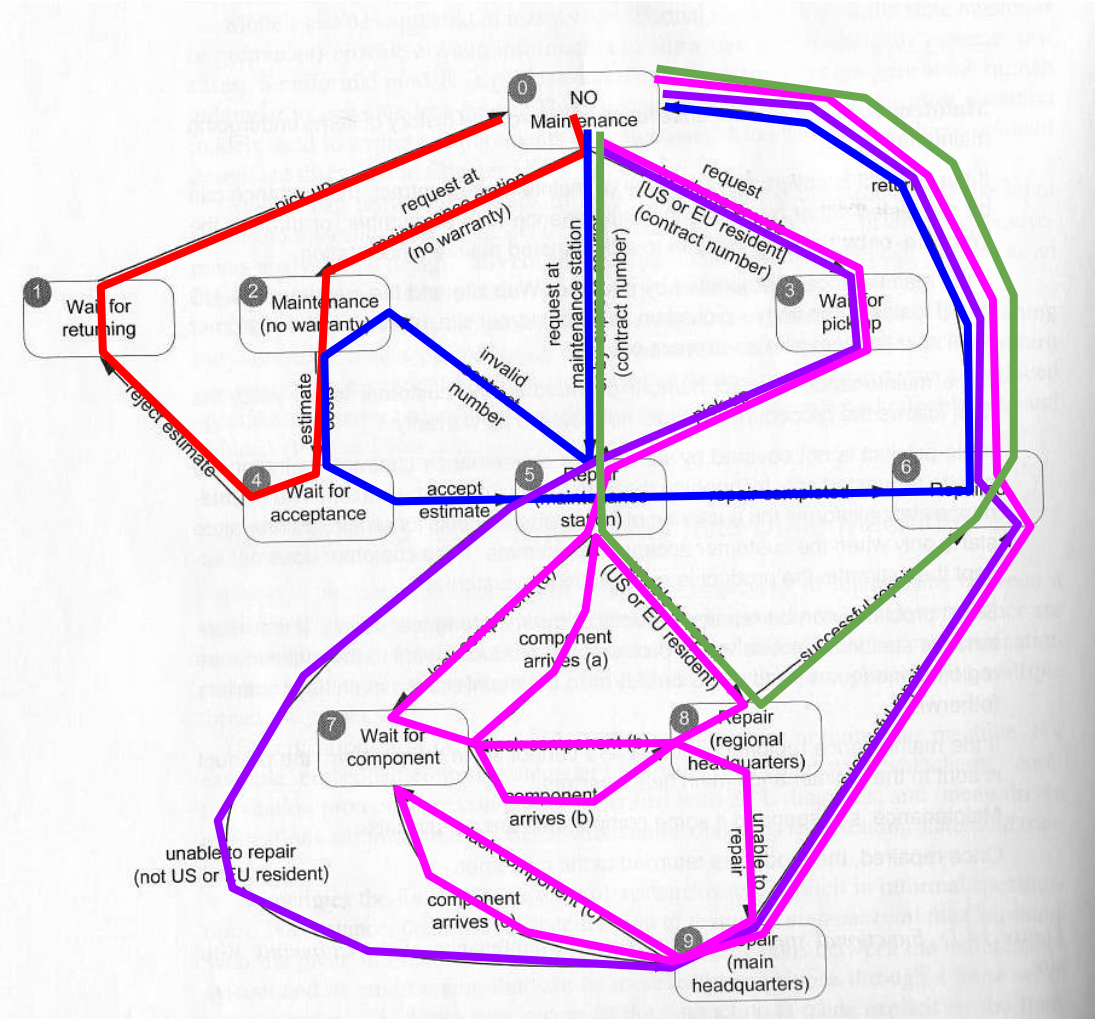
State Coverage

- Each state has been reached by one or more test cases.
- Analog to statement coverage - unless the model has been placed in each state, all 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



- Test cases often given as a list of states or transitions to be covered.
- No “final” states, could achieve transition coverage with one large test case.
  - Smarter to break down FSM and target sections in isolation.

Example Suite:
T1: 0-2-4-1-0
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.

Decision Structures

Logic Terminology

- A *predicate* is a function with a boolean outcome (true/false).
  - When the inputs of the function are clear, they are left implicit.
    - We don't care how accounts are represented. There is just a predicate "educational-customer".
- A *basic condition* is a predicate that cannot be decomposed further.
- A *complex condition*, or *decision*, is 2+ basic conditions, connected with operators.

Decision Structures

- Specifications are often expressed as *decision structures*.
  - Conditions on input values, and the corresponding actions or results.
  - Example:
    - NoDiscount = (indAcct ^ !(current > indThreshold) ^ !(offerPrice < indNormalPrice)) v (busAcct ^ !(current > busThreshold) ^ !(current > busYearlyThreshold) ^ !(offerPrice < busNormalPrice))
- Decision structures can be modeled as tables, relating predicate values to outputs.

Decision Tables

- Decision structures can be modeled as tables, relating predicate values to outputs.
- Rows represent basic conditions.
- Columns represent combinations of conditions, with the last row indicating the expected output for that combination.
- Cells are labeled T, F, or - (don’t care).
- Column is equivalent to a logical expression joining the required values.

Decision Tables

- Can be augmented with a set of constraints that limit combinations.
  - Formalize the relations among basic conditions
  - Expressions over predicates:
    - (Cond1 ^ !Cond2 => Cond3)
  - Short-hand for common combinations:
    - at-most-one(C1...Cn)
    - exactly-one(C1...Cn)

Cond1	T	F
Cond2	F	-
Cond3	T	T
Out	T	F

Example Decision Table

EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > Ct2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP > Sc	F	T	F	T	-	-	-	-
SP > T1	-	-	-	-	F	T	-	-
SP > T2	-	-	-	-	-	-	F	T
Out	Edu	SP	ND	SP	T1	SP	T2	SP

Constraints
at-most-one(EduAc,BusAc)
at-most-one(YP < YT1, YP > YT2)
at-most-one(CP < CT1, CP > CT2)
at-most-one(SP < T1, SP > T2)
YP > YT2 => YP > YT1
CP > CT2 => CP > CT1
SP > T2 => SP > T1

Abbreviations
CP = current purchase
YP = yearly purchase
C(Y)T = current/yearly threshold
SP = special price
Sc = scheduled price
T1 = tier 1
T2 = tier 2
Edu = educational discount
NP = no discount

Decision Table Coverage

- Basic Condition Coverage
  - Translate each column into a test case.
  - Don’t care entries can be filled out arbitrarily, as long as constraints are not violated.
- Compound Condition Coverage
  - All combinations of truth values for predicates must be covered by test cases.
  - Requires 2^n test cases for n predicates.
    - Can only be applied to small sets of predicates.

Decision Table Coverage

- Modified Decision/Condition Coverage (MC/DC)
  - Each column represents a test case.
  - In addition, new columns are generated by modifying the cells containing T and F.
  - If changing a value results in a test case consistent with an existing column, the two are merged back into one.
  - A test suite should not just test positive combinations of values, but also negative combinations.

Example Decision Table

EduAc	T	T	F	**T**	**F**	**F**	**F**	F	F	F	F	F
BusAc	-	-	F	**F**	**T**	**F**	**F**	F	F	F	F	F
CP > CT1	-	-	F	**F**	**F**	**T**	**F**	F	T	T	-	-
YP > YT1	-	-	-	**-**	**-**	**-**	**-**	-	-	-	-	-
CP > Ct2	-	-	-	**-**	**-**	**-**	**-**	-	F	F	T	T
YP > YT2	-	-	-	**-**	**-**	**-**	**-**	-	-	-	-	-
SP > Sc	F	T	F	**F**	**F**	**F**	**T**	T	-	-	-	-
SP > T1	-	-	-	**-**	**-**	**-**	**-**	-	F	T	-	-
SP > T2	-	-	-	**-**	**-**	**-**	**-**	-	-	-	F	T
Out	Edu	SP	ND	**Edu**	**ND**	**T2**	**SP**	SP	T1	SP	T2	SP

Activity

- Airline Ticket Discount Function
  - Read the specification and draw a decision table.
  - How many tests would be required for compound condition coverage?
  - Expand the table to form a MC/DC test suite. How many tests were added?

Activity - Decision Table

Infant	T	T	F	F	F	F
Child	F	F	T	T	F	F
Domestic	T	F	-	-	-	F
International	F	T	-	-	-	T
Early	-	-	T	-	T	-
Off-Season	-	-	-	-	-	T
Discount	80	70	20	10	10	15

Constraints:

- Infant => !Child
- Child => !Infant
- Domestic => !International
- International => !Domestic
- Domestic xor International

Activity - Decision Table

I	Infant	T	F	F	F	F	F	T	F	F		T	F	F	F	F
C	Child	F	F	T	T	F	F	F	T	F		F	T	F	F	F
D	Domestic	T	T	-	-	-	F	F	F	F		F	F	F	F	F
I	International	F	F	-	-	-	T	T	T	T		T	T	T	F	T
E	Early	-	-	T	F	T	-	-	-	-		-	-	-	-	-
C	Off-Season	-	-	-	-	-	T	T	T	F		T	T	F	T	F
D	**Discount**	80	0	20	10	10	15	70	15	0	5	70	15	0		

Constraints:

- Infant => !Child
- Child => !Infant
- Domestic => !International
- International => !Domestic
- (Domestic xor International)

Grammars

Grammars

- Specifications for complex documents or domain-specific languages are often structured as grammars.

<search> ::== <search> <binop> <term> | not <search> | <term>

<binop> ::== and | or

<term> ::== <regexp> | (<search>)

<regexp> :== Char<regexp> | Char | {<choices>} | *

<choices> ::== <regexp> | <regexp>,<choices>

- Tests can be derived from these structures.

Grammar-Based Input

- Grammars are useful for representing complex input of varying and unbounded size, with recursive structures and boundary conditions.
  - Example, XML files.
    - Document built from a set of standard tags.
    - There are rules on how those tags are formatted.
    - However, some tags may appear multiple times, are optional, or may appear in different orders.
  - Can use the grammar to derive input for a function.

Generating Input

- A test case is a string generated from that grammar, then fed to the function.
- A production is a grammar element:
  - <binop> ::== and | or
  - <binop> is a non-terminal symbol (it can be broken down further)
  - “and” is a terminal symbol (it can’t be broken down further)
- Start from a non-terminal symbol and apply productions to substitute substrings from non-terminals in the current string until we get a string entirely made of terminals.

Generating Input

- At each step, we must choose productions to apply to the string.
  - Generation is guided by coverage criteria, defined as coverage *over the grammar* rather than coverage over the program.
- Production Coverage - Each production must be exercised at least once by a test case.
  - Requires a strategy for how productions are selected.

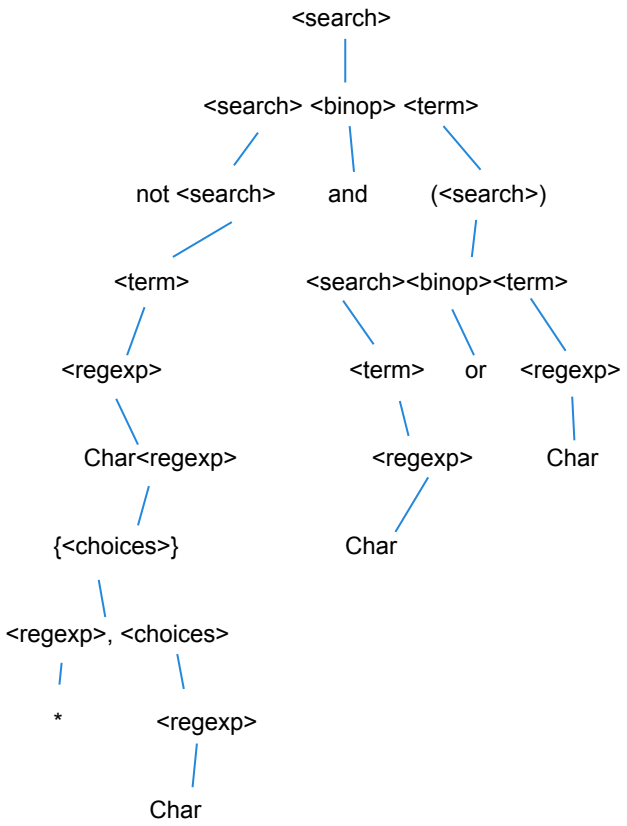
Selecting Productions

- Test and suite size can be tuned based on the strategy.
  - Favor productions with more terminals.
    - Large number of tests, each test will be small.
  - Favor productions with more non-terminals.
    - Small number of tests, where each test is larger.

<search> ::==	<search> <binop> <term> | not <search> | <term>
<binop> ::==	and | or
<term> ::==	<regexp> | (<search>)
<regexp> :==	Char<regexp> | Char | {<choices>} | *
<choices> ::==	<regexp> | <regexp>,<choices>

Production Coverage Example

"not Char {*,Char} and (Char or Char)"



<search> ::==	<search> <binop> <term> | not <search> | <term>
<binop> ::==	and | or
<term> ::==	<regexp> | (<search>)
<regexp> :==	Char<regexp> | Char | {<choices>} | *
<choices> ::==	<regexp> | <regexp>,<choices>

Boundary Condition Grammar-Based Coverage

- BCGBC applies boundary conditions on the number of times each recursive production is applied per test.
- Choose a minimum and maximum number of applications of a recursive production.
  - Generates tests that apply each the minimum, minimum + 1, maximum, maximum -1.
  - Similar to boundary interior coverage.

Boundary Condition Grammar-Based Coverage

- Results in production coverage, plus:
  - 0 required components (compSeq1 * min)
  - 1 required component (compSeq1 * min + 1)
  - 15 required components (compSeq1 * max -1)
  - 16 required components (compSeq1 * max)
  - 0 optional components (optSeq1 * min)
  - 1 optional component (optSeq1 * min + 1)
  - 15 optional components (optSeq1 * max -1)
  - 16 optional components (optSeq1 * max)

Probabilistic Grammar-Based Coverage

- Selection of productions can be biased by assigning weights to each production and factoring those into test generation.
  - For each production, assign a weight.
    - 10 = use 10x as often as those with weight 1
    - Equal weights indicate that those productions are used an equal number of times.
    - 0 = never use this production
- Multiple sets of weights can be kept to model different types of input.

We Have Learned

- If we build models from functional specifications, those models can be used to systematically generate test cases.
- Helps identify important combinations of input to the system.
- 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.
- Decision tables model complex combinations of conditions and their expected outcomes.
  - Cover basic conditions and their combinations.
- Grammars model complex functional input.
  - Cover different combinations of grammar productions.

Next Time

- Test Oracles
  - How do we judge the success of a test case?
  - Reading: Section 17.5-17.7

- Homework:
  - Reading assignment due tonight.
  - Homework 2 - questions?