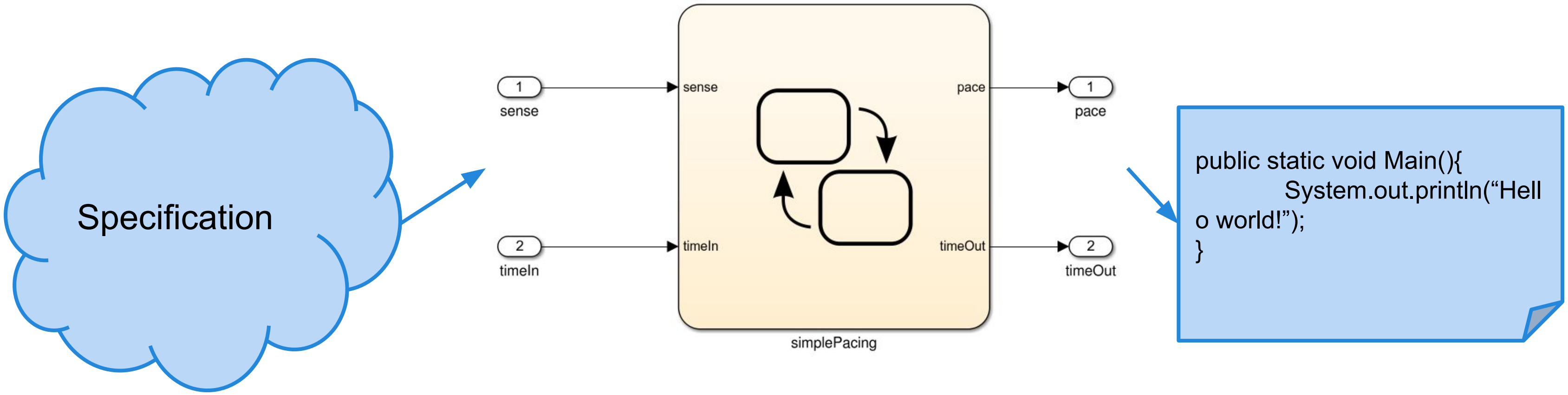
Model-Based Testing:
Decision Structures And Grammars

CSCE 747 - Lecture 12 - 02/16/2017

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?



If the model satisfies the specification...

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

And If the model accurately represents the program.

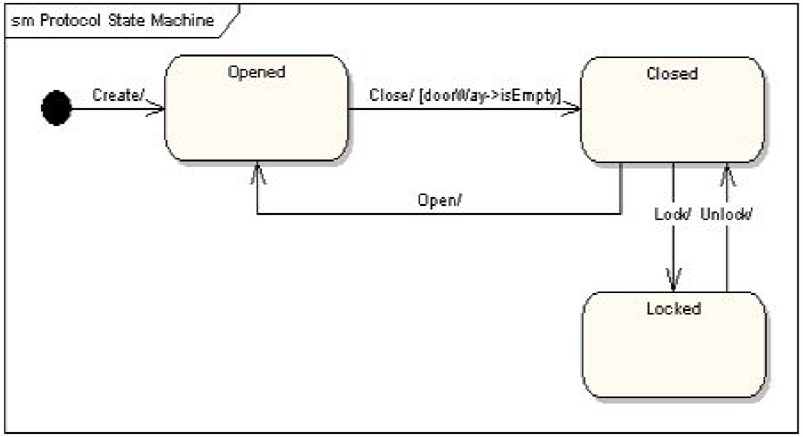
… Then we can derive test cases from the model that can be applied to the program. If the model and program do not agree, then there is a fault.

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

- 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.



State Machine Testing

- State machines have structure that can be *covered* by test cases.
  - State Coverage
  - Transition Coverage
  - Path-based Metrics (Single State/Transition Path Coverage, Boundary Interior Loop Coverage)

Other Forms of Model-Based Testing

- Decision Structures
  - Requirements often expressed as complex decision predicates.
  - Design test cases that *cover* these predicates.
- Grammars
  - Complex inputs are often described using grammars.
  - To help ensure that complex input structures are fully explored when designing test cases, we can measure *coverage* of the grammar.

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 *condition* is a predicate that cannot be decomposed further.
- A *decision*, is 2+ conditions, connected with operators (and, or, xor, implication).

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.

Example - Basic Condition Coverage

								?
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

Test 1: (T,-,-,-,-,-,F,-,-)

Test 2: (T,-,-,-,-,-,T,-,-)

Test 3: (F,F,F,-,F,-,F,-,-)

Example - Compound Condition Coverage

EduAc	T	T
BusAc	F	T
CP > CT1	F	F
YP > YT1	F	F
CP > Ct2	F	F
YP > YT2	F	F
SP > Sc	F	F
SP > T1	F	F
SP > T2	F	F

… etc
(2^9 combinations)

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

Removes 128 combinations
Removes 96 more combinations
Removes 64 more combinations

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

Infant	T	F	F	F	F	F	T	F	F	F	T	F	F	F	F
Child	F	F	T	T	F	F	F	T	F	F	F	T	F	F	F
Domestic	T	T	-	-	-	F	F	F	F	F	F	F	F	F	F
International	F	F	-	-	-	T	T	T	T	T	T	T	T	F	T
Early	-	-	T	F	T	-	-	-	-		-	-	-	-	-
Off-Season	-	-	-	-	-	T	T	T	F	T	T	T	F	T	F
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.

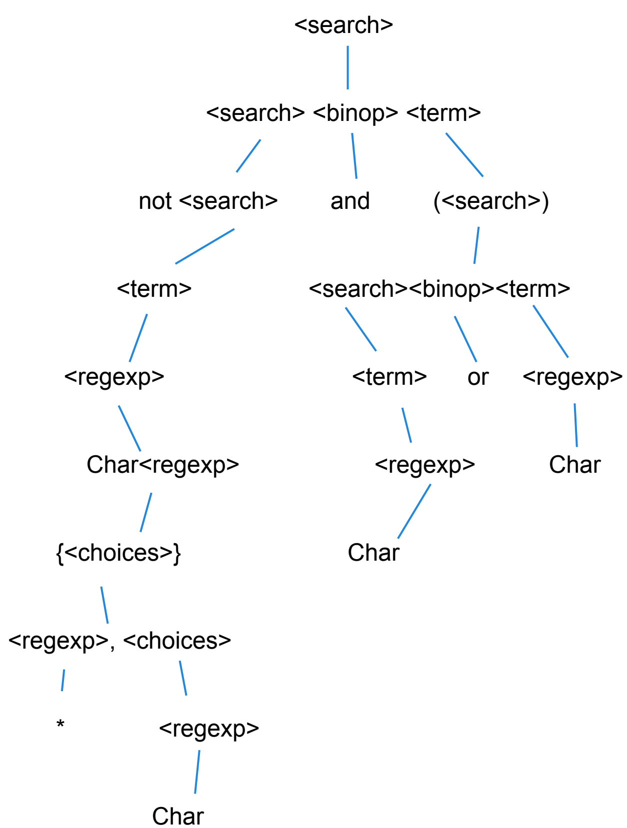
Selecting Productions

- 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>

Activity - Production Coverage

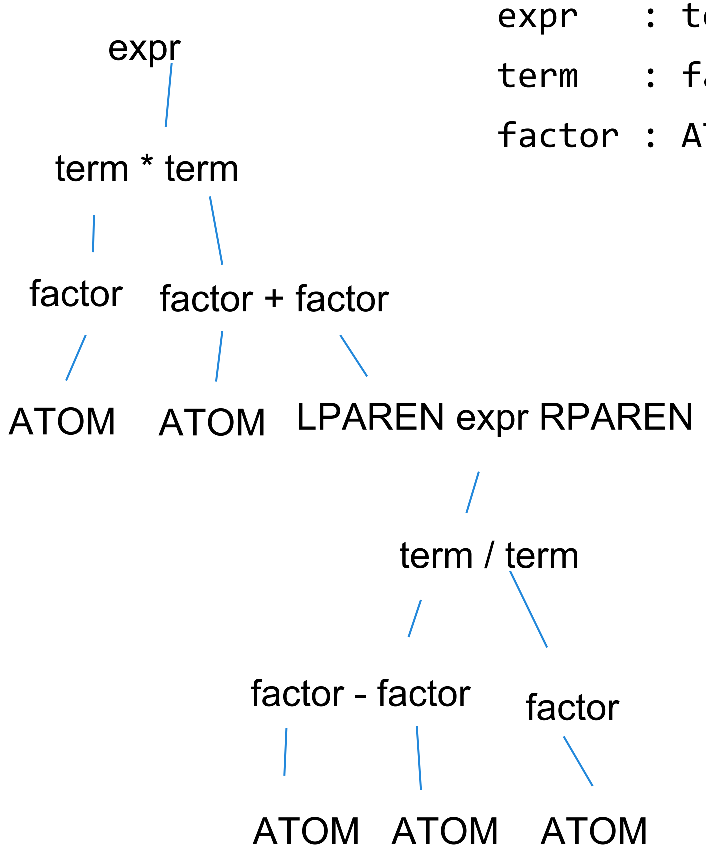
Derive a test suite that covers each production in this grammar.

```
expr   : term | term * term | term / term
term   : factor | factor + factor | factor - factor
factor : ATOM | LPAREN expr RPAREN

ATOM = 0..9
LPAREN = (
RPAREN = )
```

Activity Solution

```
expr   : term | term * term | term / term
term   : factor | factor + factor | factor - factor
factor : ATOM | LPAREN expr RPAREN
```



ATOM * ATOM + (ATOM - ATOM / ATOM)
ex: 9 * 8 + (7 - 6 / 5)

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.

Next Time

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

- Homework:
  - Homework 2 - questions?