Software Testing and Validation

A.A. 2022/2023

Corso di Laurea in Informatica

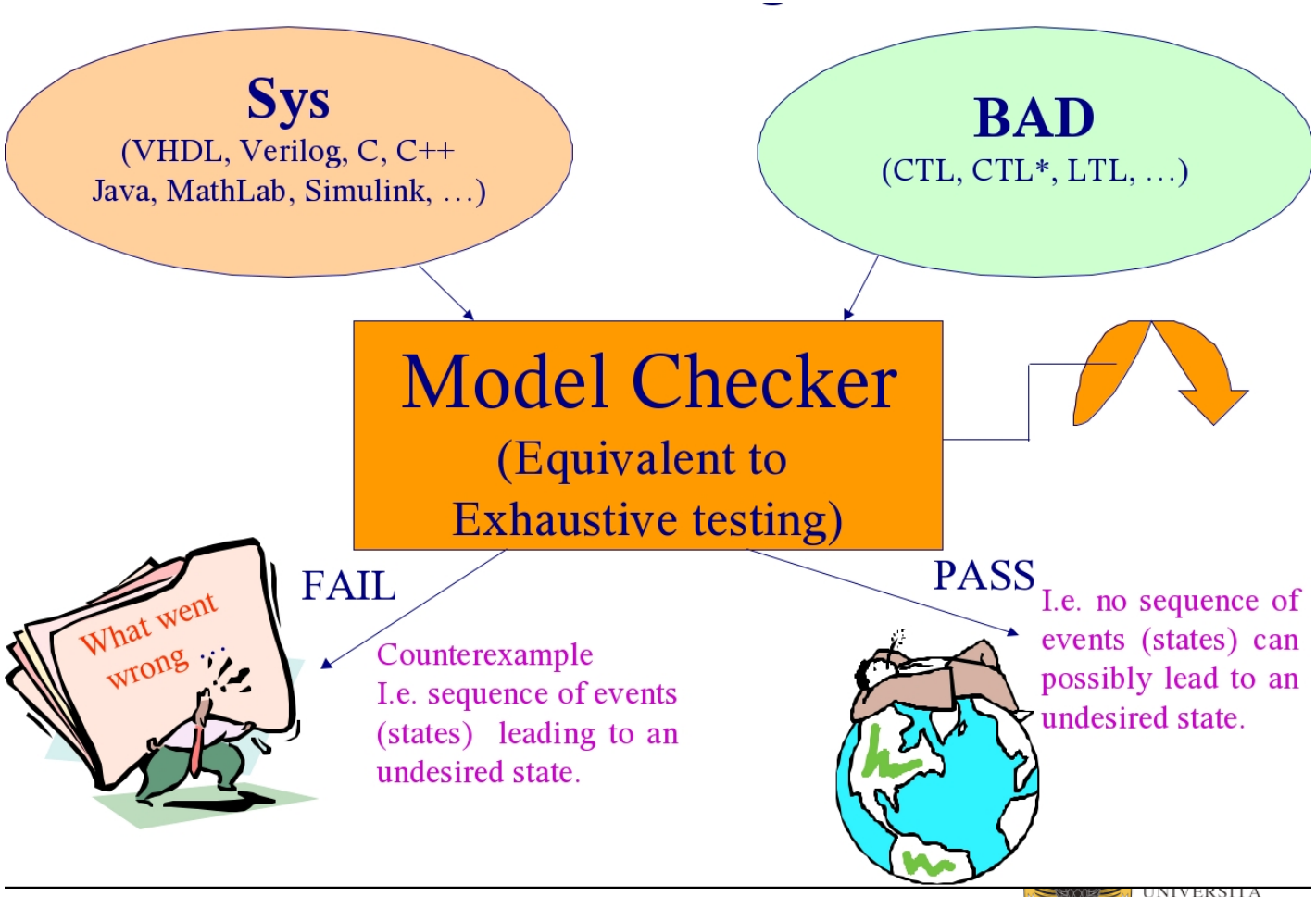
Testing Methodologies

Igor Melatti

Università degli Studi dell'Aquila

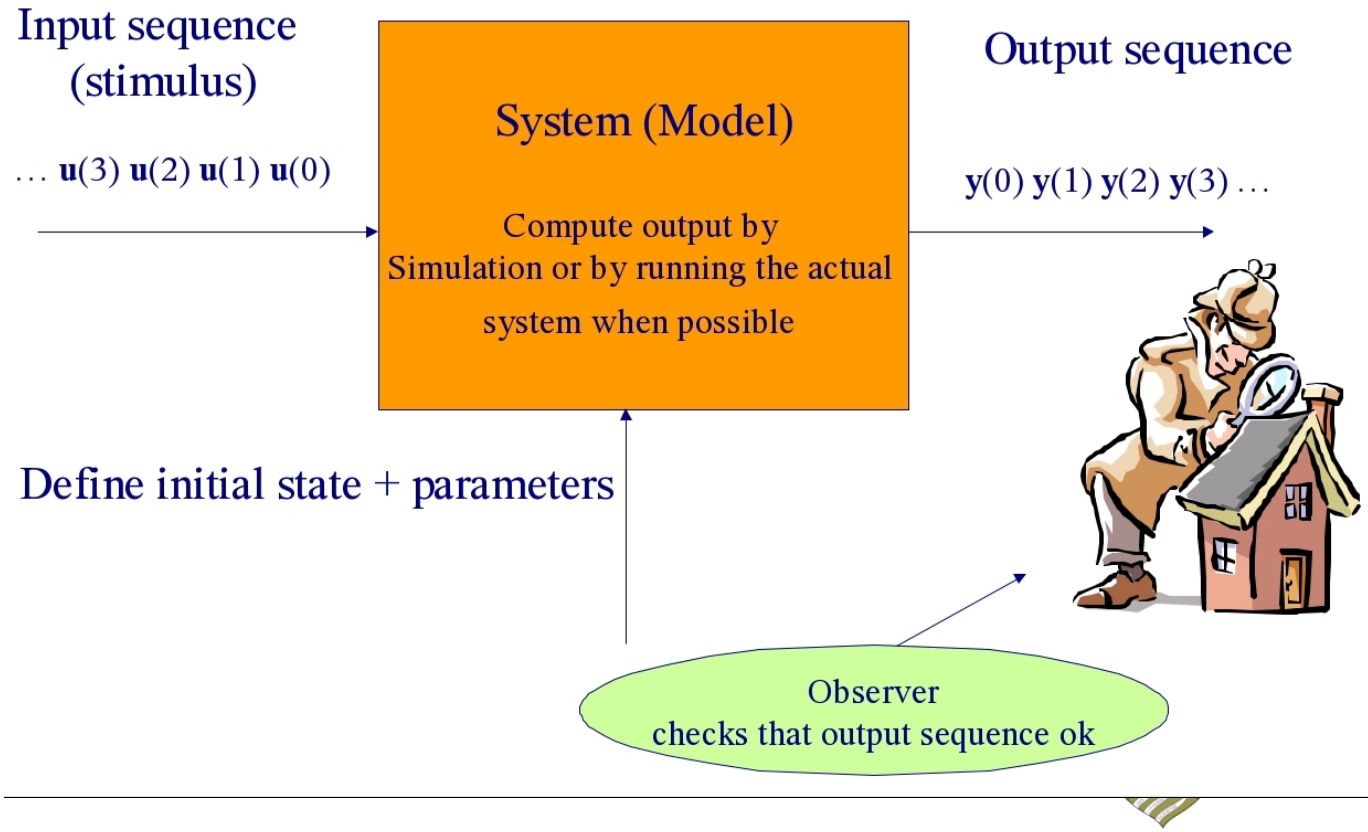
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Model Checking...



... to Testing

An approximate answer
BUG HUNTING: Testing + Simulation



Testing AKA Get the Inputs

- Model Checking main difficulty:
  - choose the corrent model checker
  - understand the system and model it within the model checker input language
  - understand the properties of the system and specify them within the model checker temporal logic
- Testing main difficulty: which inputs should I use?
  - also running tests and observing results are problems, but less difficult
- Recall that inputs are theoretically infinite and practically too many
  - a function taking an input integer...

Testing AKA Get the Inputs

- There are exeptions:
  - programs without inputs
    - e.g.: always returns the same constant, or a constant depending on previous executions
    - but one input is always present: launch the system...
  - programs taking enumerated inputs only
    - e.g.: a function taking two booleans
- In the vast majority of cases, too many inputs to consider them all, must somehow select a "meaningful" subset
  - i.e., so that errors, if present, are likely to be detected
- No general tools available, only some methodology as a good practice

Terminology

Program something to be tested

- could also not be a full program but a part of it

Test case A set of inputs, execution conditions, PASS/FAIL criterion

- input is anything the program to be tested can get
  - files, interrupts, mouse coordinates, ...
- execution condition: information on the test execution
  - typically, input timing: whether all input must be provided at the start or not
  - e.g., a sequence of interrupts if a given timing...
- PASS/FAIL: some way to check
  - e.g.: output must be equal to this expected result

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Terminology

Test case specification A formal or informal description of a test case

- "the input is two words" $\rightarrow$ a valid test case will be "goodbye all"

Test suite a set of test cases

Test execution running the test cases on (part of) the system

Test obligation some property a test case (specification) must fulfill

- e.g., "all input words must be at least 7 letters long"

Adequacy Criteria some property a test suite must fulfill

- e.g., "all test cases must contain at least 30 inputs"

Terminology

Unit Smallest unit of work in the program

- typically (but not always) close to single functions or single classes
- here, "unit of work" roughly refers to:
  - the smallest increment by which a software system grows or changes
  - the smallest unit that appears in a project schedule and budget
  - the smallest unit that may reasonably be associated with a suite of test cases (*unit testing*)

Function Mathematical concept (set of pairs)

Java Function Syntactical function in Java language

- works with all other languages, of course

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Terminology

Independently Testable Feature (ITF) Some functionality of the program which can be isolated from the other functionalities

- not necessary at code level: here, it is testing level
- e.g., a program or a function may be able to both sort and merge files
- however, sorting and merging may be ITF
- granularity depends on the program: from individual functions, to features of an integrated system composed of many programs
- going through individual classes and libraries
- when detected at unit testing, an ITF is usually a function/method or a class, but not only unit testing exists...

Typical Setting for Testing

1. The tester engineer (TE) must develop:
   - a set of test case specifications
   - test adequacy for the overall test suite
2. TE generates the test cases from the test case specifications
3. TE runs the test and collect the results
   - test are instrumented so as to also check adequacy criteria
4. If adequacy criteria are not met, revise test case specifications going back to step 1
5. Developers correct all discovered errors and TE starts again from 3
   - no need to wait for the step 4: as soon as an error is discovered, it can be corrected
   - it may happen that specifications should be updated too $\rightarrow$ back to step 1

Example (White-Box Testing)

```
public static String collapseSpaces(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != ' ' || last != ' ')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```

Example (Black-Box Testing)

We have a function taking one string s as an input and returning a string s' as an output.

Informally: s' is the same as s, but *consecutive* spaces are collapsed into one space.

Formally: let $S = \{(i,k) \mid s_i = \text{`` ''} \wedge (i > 1 \rightarrow s_{i-1} \neq \text{`` ''}) \wedge k > 1 \wedge \wedge_{n=1}^{k-1} s_{i+n} = \text{`` ''}\}$. Let
$S' = \{(j,k) \mid \exists (i,k) \in S \wedge j = i - \sum_{(i',k) \in S \mid i<i'} (k-1)\}$. Let
$\sigma(S,j) = \sum_{(i,k) \in S \mid i<j} (k-1)$.
Then, for all $j = 1, \ldots, |s| - \sum_{(i,k) \in S} (k-1)$, $s'_j = s_{j+\sigma(S',j)}$

Example

- Test case specification 1: all input should be at least 10 characters long
- Test case specification 2: all input should contain at least 3 spaces
- Test obligation 1: the test suite should contain an empty string
- Test obligation 2: in the `if`, the first clause should always evaluate to true and the second to false at least once (and/or viceversa)
- We generate the test suite following specifications
- We check if obligations are ok

Example

- Test obligation (*coverage*): all statements should be executed at least once
- What happens if the code has some unreachable code? No test suite is adequate!
- Quantitative measures could be used
- Suppose an adequacy criterion generates n obligations...
- ... if m of such obligations are met by the test suite, then it is $100\frac{m}{n}\%$ adequate

Functional Testing

- Typically, we do not only have a program: we also have a functional specification of its behaviour
  - in some logic, or even in natural language (starting comments of a function...)
  - requirements are expressed by users and specified by software engineers
- Functional specifications are the base for functional testing
- More precisely, *functional test case design* is about deriving test cases from functional specifications
- The structure of the program is completely ignored
  - e.g., "all `ifs` must be evaluated at least once" is not functional testing
- Also called *black-box* testing
- Cheaper than white-box or glass-box testing

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Functional Testing

- Functional testing techniques in brief:
  - input: program specification
  - output: test cases specification
- Core of the methodology: partitioning the possible behaviors of the program into a finite number of homogeneous classes
  - not an actual partition, as they may overlap
  - "homogeneous" in a broad sense, depends on the program
  - often requires to integrate program specifications, good for project documentation!
- Human effort required, similar to modeling in model checking
  - in few cases, if program specifications are already formal, the work is easier
  - e.g., a model checking specification may be directly translated in test cases

Functional Testing

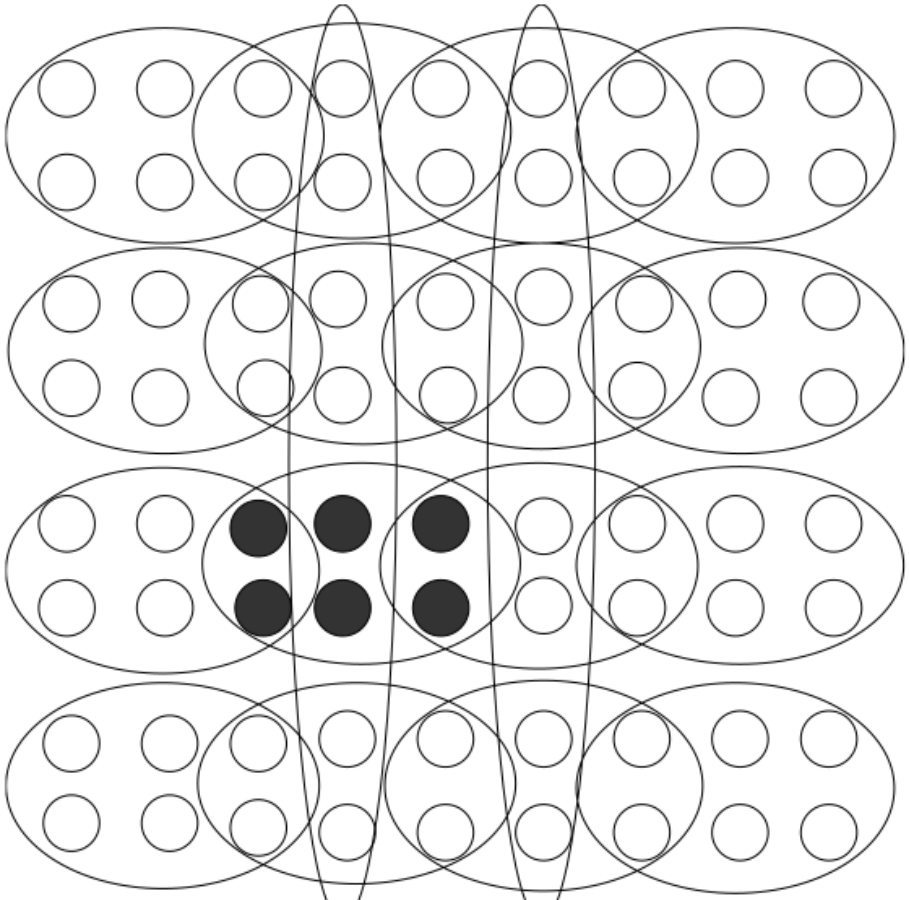
- A function with 2 32-bit integers has $2^{64} \approx 10^{21}$ possible different inputs
- Given budget limitations, only an extremely tiny fraction of inputs may be tested
  - limitations are both in money and time
- Random sampling: choose test cases from a random distribution
  - of course, depending on the program specifications
  - e.g., for a function taking 3 floating points and a string, we sample from $\mathbb{R}^3 \times A^*$, if A is the alphabet
- To do: build a program that generate test cases by sampling the given input space
  - by definition of test case, this must also include a function to check the result

From Pure Random Sampling to Partitioning

- Simply random is not a good choice, better a kind of "guided" random
- Best way to guide is perform *partitioning* of input space
  - not an actual partition: the union is the whole, but partitions may have some non-null intersection
  - however, "partitioning" is standard terminology for testing
- Typical desired property: there does not exist a partition containing both failure and non-failure inputs
- In fact, by random sampling from each partition, we will for sure consider all failure inputs
  - partition of failure inputs only $\rightarrow$ some failure will be detected
- Of course, the property is desirable but impossible to obtain in the general case

Partition Example

All inputs that lead to a failure belong to at least one class that contains only inputs that lead to failures
Suppose that one more input failure is added: is the desired property ok? if not, how to modify the partition to ensure it again?



Partition Testing

- *Partition testing*: any method that partition the input spaces in a finite number of partitions as seen above
- *Functional testing*: partition testing where the partition algorithm is based on the program specification
  - also called *specification-based partition testing*
- Generating test cases is more expensive: we also have to guarantee they belong to partitions
  - pure random does not check this, thus it is simpler
- Fewer test cases generated in the same amount of money and time
- However, it is typically more effective in finding failures

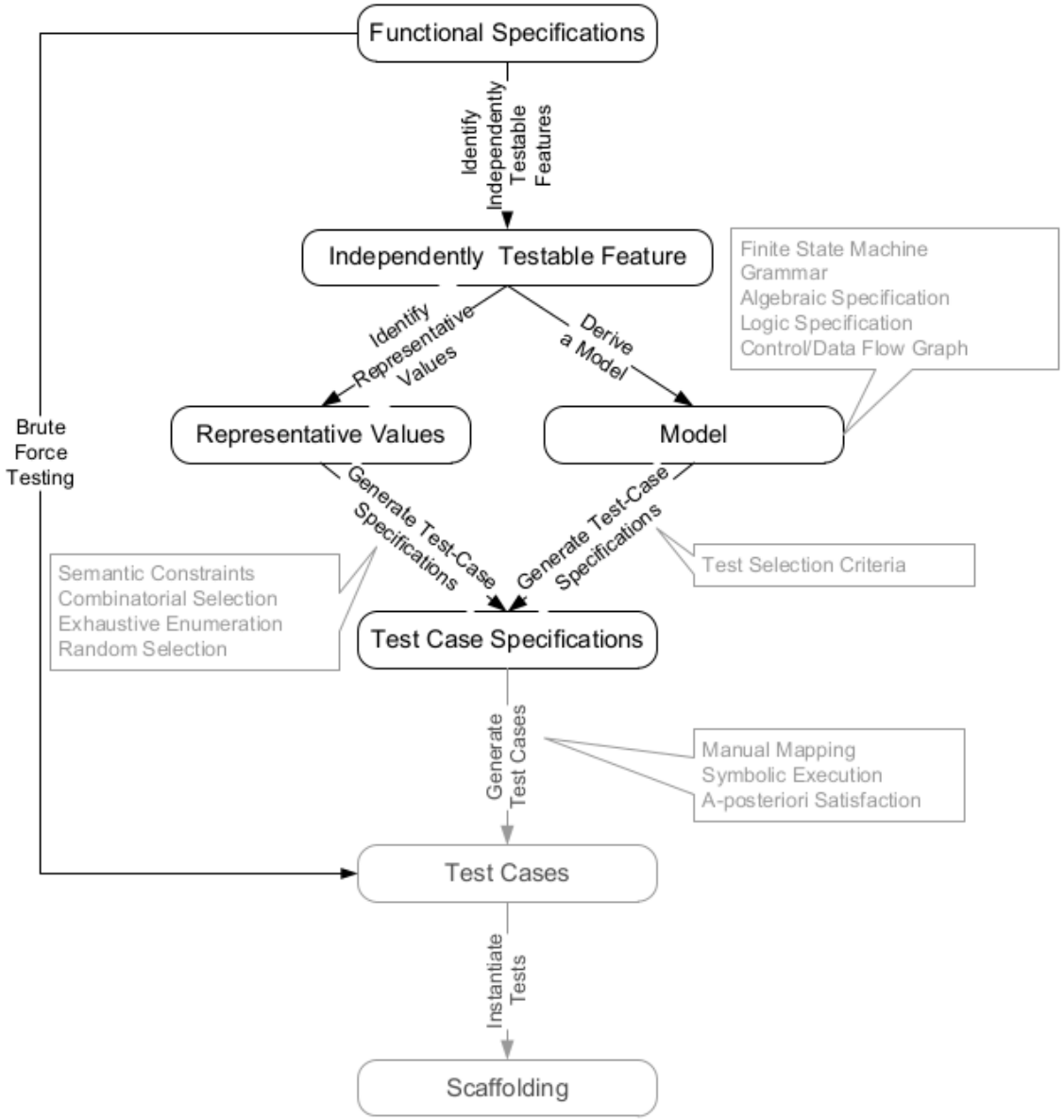
Partition Testing

- Ideally: all partitions are all-failure or all-non-failure
- Impossible to fully obtain such property, so how to at least come close?
  - could be ok if all failing inputs belong to at least one class that contains only failing inputs
- Experience is needed
  - learning to detect which classes of test case are "more alike" than others
  - in the sense that failure-prone test cases are likely to be concentrated in some classes
- The less the partitions, the closer to pure random testing
  - having few partitions may be a good trade-off given testing budget

A Systematic Approach for Functional Testing

- Idea: divide brain-intensive from automatable steps
- Many problems to overcome
  - a particular functional testing technique may be effective only for some kinds of software
  - or may require a given specification style
- The following is a general pattern of activities that captures the essential steps in different functional test design techniques
- In this way, relations among the techniques may become clearer
- Furthermore, the test designer may gain insights into adapting and extending these techniques

A Systematic Approach for Functional Testing



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

A Systematic Approach for Functional Testing

- Identify Independently Testable Features
  - web page specification: search the DB, update the DB, provide info from the DB
  - sub-functionalities: edit a pattern to search the DB, provide a form for registering, ...
  - rather than having a test case for multiple functionalities, it is better to devise separate test cases for each functionality of the system
  - different from module decomposition: program users perspective vs. developers
    - recall that a program user may also be another program
    - requires detailed specification
- Thus step 1 is to identify "separated" features
  - lack of documentation may make this difficult
  - thus, we are actually partitioning the input space, as said before

A Systematic Approach for Functional Testing

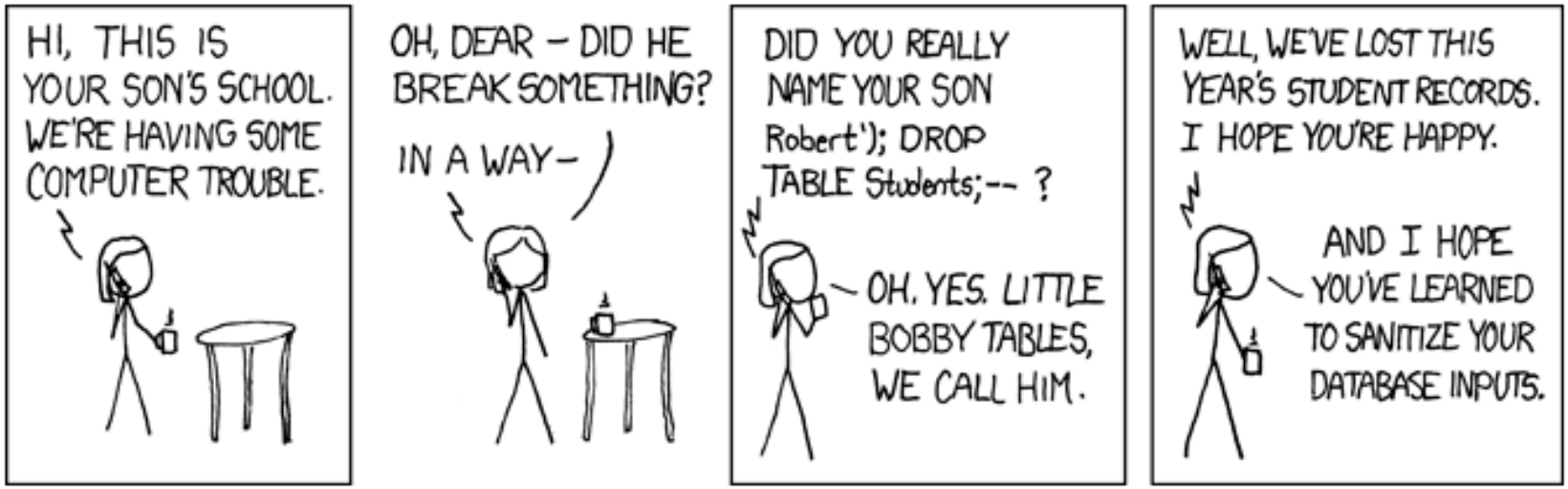
- For each ITF identified, a number of inputs are needed
  - e.g., a registration on a Web page (single ITF) needs name, surname, age, ...
  - in some cases, some input may be hidden and must be explicited
    - e.g., when checking if some name is in a database, the input is not only the name, but also the database!
- For each of such inputs, a choice is needed between:
  1. identify representative values
     - meaningful fixed values for inputs are directly enumerated
  2. build a model
     - some model is built which generates values for input
     - e.g., a grammar may be built, defining the valid values
     - also an algorithm could be written

A Systematic Approach for Functional Testing

- Combine the values of the different inputs involved
  - if all of them have been enumerated in the previous step, the Cartesian product may be used
  - however, this works for few inputs with few values, as the size explodes
  - e.g., 6 inputs with 6 values each results in about 50k tests
- Thus, we need something more clever:
  - detect illegal combinations
  - select a practical and meaningful subset of legal combinations
- Example: 2 input numbers representing length of a string and number of special characters
  - the 0 length + more than 1 special character is illegal

Illegal Combinations: What To Do?

- It could be straightforward to think that illegal combinations of inputs must be always ruled out
- However, illegal combinations often have to be tested as well
- We may consider two possible cases:
  1. the ITF is for the "general public"
  2. the ITF is an API, to be invoked by programs
- As for case 1, illegal combinations should always be tested



Illegal Combinations: What To Do?

- Must illegal combinations of input be always ruled out? NO!
- We may consider two possible cases:
  1. the ITF is for the "general public"
  2. the ITF is an API, to be invoked by programs
- As for case 1, illegal combinations should always be tested
  - generic users may easily feed "wrong" inputs
  - an error must be returned, not a failure!
- As for case 2, it depends
  - if the ITF is for internal use only, and some assurance of compliance is present from the specifications, we may rule out the illegal combination
  - otherwise, generic programs may be as generic users...

A Systematic Approach for Functional Testing

- General techniques reducing Cartesian products do not exist
- Insights may be present in the documentation/specification
- Typical strategies include:
  - considering a subset of each ITF
  - considering exhaustive combinations only for selected pairs
- The output is a test case specification, but may also be directly a test case
  - also an algorithm could be viewed as a test case specification
- Finally, generate the test case and instantiate (i.e., run) it
  - select one or more test cases for each test case specification
  - scaffolding for the actual run, we will be back on this

Combinatorial Testing

- Describes the methodologies to obtain the general approach for functional testing described above
- Two main techniques may be employed
  - Category-Partition Testing
  - Catalog-Based Testing
- A third technique only deals with the combinatorial part, thus may be applied to both: Pairwise Combination Testing

Category-Partition Testing

- Suppose we have selected an ITF and one parameter of such ITF
- We have to list all of its *categories*
  - some characteristic which may differentiate among possible inputs for that parameter
  - e.g.: for a string, its length, or the number of special characters
  - e.g., for an integer, being positive or negative
  - categories may be defined also for combinations of parameters (*environmental conditions*)
  - e.g.: for a parameter string (pattern) to be found in a text: number of occurrences
  - in some cases, the "expected result" category could be added

Category-Partition Testing

- Then, we *partition* each category into *choices*
  - in the testing sense: different partitions may have non-empty intersections
- This is done by providing general and coincise rules to each category
  - e.g., the length of a string may be 0, 1, between 5 and 20, greater than 50
  - e.g., an integer may be negative, 0 or strictly positive
- Defining and using *properties* might help for impossible combinations
  - e.g., if the length of a string is 0 the property may be "property:Empty"
  - e.g., if the number of special characters is 1, it may be applied only "if:NonEmpty"

Catalog-Based Testing

- Suppose we have selected an ITF and its parameters
- Three steps:
  1. Identify variables, definitions, preconditions, postconditions and operations on ITF parameters from the specification
  2. Derive a first set of test case specifications from the items identified above
  3. Complete the test case specifications using catalogs
- Catalogs are built over time and experience, help in identify values for a specific class
  - each software house (and developer/test engineer) has its own
  - e.g., when an integer is involved, always include a test with that integer equal to 0
- Catalog-Based Testing is a good technique also without catalogs
  - on the contrary, using catalog only may not be a good idea
  - catalogs may also be used in Category-Partition Testing

Catalog Example

Boolean
- [in/out] True
- [in/out] False

Enumeration
- [in/out] Each enumerated value
- [in] Some value outside the enumerated set

Range $L \ldots U$
- [in] $L-1$ (the element immediately preceding the lower bound)
- [in/out] L (the lower bound)
- [in/out] A value between L and U
- [in/out] U (the upper bound)
- [in] $U+1$ (the element immediately following the upper bound)

Numeric Constant C
- [in/out] C (the constant value)
- [in] $C-1$ (the element immediately preceding the constant value)
- [in] $C+1$ (the element immediately following the constant value)
- [in] Any other constant compatible with C

Non-Numeric Constant C
- [in/out] C (the constant value)
- [in] Any other constant compatible with C
- [in] Some other compatible value

Sequence
- [in/out] Empty
- [in/out] A single element
- [in/out] More than one element
- [in/out] Maximum length (if bounded) or very long
- [in] Longer than maximum length (if bounded)
- [in] Incorrectly terminated

Scan with action on elements P
- [in] P occurs at beginning of sequence
- [in] P occurs in interior of sequence
- [in] P occurs at end of sequence
- [in] PP occurs contiguously
- [in] P does not occur in sequence
- [in] pP where p is a proper prefix of P
- [in] Proper prefix p occurs at end of sequence

Identify Elementary Items of the Specification

- From initial specification of a unit to elementary items of basic types:

  Preconditions conditions on input which must be true before invoking the unit test
  - may be checked by the unit itself (*validated preconditions*)...
  - or by the outside caller (*assumed preconditions*)

  Postconditions result of execution

  Variables input, output or intermediate

  Operations performed on input and/or intermediate values

  Definitions shorthands in the specification

Derive a First Test Case Specifications

- We want to partition the input domain, and we use the previously collected information for this purpose
  - for each validated precondition P, we have two classes of inputs: inputs in which P holds, and inputs in which P does not hold
    - a single validated precondition may be split in two or more parts, if it involves ANDs or ORs
  - for each assumed precondition P, we only consider input satisfying P
    - otherwise, unit behaviour is typically undefined
  - if a postcondition has a guard, consider the guard as a validated precondition
  - if a definition involving variables has a guard, consider the guard as a validated precondition

Complete Test Case Specifications Using Catalogs

- Generate additional test case specifications from variables and operations
- This is done using a pre-existing catalog, to be sequentially scanned:
  - for each catalog entry, analyze all elementary items and possibly add cases
- A catalog is typically organized with an entry for each type of variable or operation
- Inside each entry, a distinction is made between input, output or input/output variables
- Then, a suggestion on values to be considered is provided
- Different catalogs may be used by different companies
  - also, in the same company for different application domains

Collapsing Spaces Example

```
public static String collapseSpaces(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != ' ' || last != ' ')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```

Collapsing Spaces Example

We have a function taking one string s as an input and returning a string s' as an output.

Informally: s' is the same as s, but *consecutive* spaces are collapsed into one space.

Formally: let $S = \{(i,k) \mid s_i = \text{' '} \wedge (i > 1 \rightarrow s_{i-1} \neq \text{' '}) \wedge k > 1 \wedge \wedge_{n=1}^{k-1} s_{i+n} = \text{' '}\}$. Let
$S' = \{(j,k) \mid \exists (i,k) \in S \wedge j = i - \sum_{(i',k) \in S \mid i<i'} (k-1)\}$. Let
$\sigma(S,j) = \sum_{(i,k) \in S \mid i<j} (k-1)$.
Then, for all $j = 1, \ldots, |s| - \sum_{(i,k) \in S} (k-1)$, $s'_j = s_{j+\sigma(S',j)}$

Collapsing Spaces Example: Category Partition Testing

- Suppose we are performing black-box unit testing; in such a case, we are forced to consider this function as an ITF
  - inputs are already identified: exactly one string
  - thus, no problems for combinations...
- Now, let us use both Category-Partition and Catalog-based Testing
- For the Category-Partition, let us begin with the characteristics of our lone input (*categorization phase*)
  - length
  - number of spaces
  - number of occurrences of consecutive spaces
  - max number of consecutive spaces
  - number of consecutive starting/trailing spaces
  - number of special characters
  - spaces only
- For the *partitioning phase*, see `cases_collapse.xls`

Another Collapsing Spaces Example

We have a function taking one string s and an integer $n \geq 2$ as an input and returning a string s' as an output.

Informally: s' is the same as s, but k consecutive spaces, s.t. $k \geq n$, are collapsed into one space.

Formally: let $S = \{(i,k) \mid s_i = \text{' '} \wedge (i > 1 \rightarrow s_{i-1} \neq \text{' '}) \wedge k \geq n \wedge \wedge_{n=1}^{k-1} s_{i+n} = \text{' '}\}$. Let
$S' = \{(j,k) \mid \exists (i,k) \in S \wedge j = i - \sum_{(i',k) \in S \mid i<i'} (k-1)\}$. Let
$\sigma(S,j) = \sum_{(i,k) \in S \mid i<j} (k-1)$.
Then, for all $j = 1, \ldots, |s| - \sum_{(i,k) \in S} (k-1)$, $s'_j = s_{j+\sigma(S',j)}$

Another Collapsing Spaces Example: Category Partition Testing

- For the Category-Partition, let us begin with the characteristics of the input k (s is as in the previous example)
  - interval
  - domain
    - if the input is provided via a GUI, it could be not an integer...
- We also have *environmental* characteristics, i.e., which look at both inputs
  - number of k consecutive spaces occurring in s
  - expected result
- For the *partitioning phase*, see `cases_collapse_k.xls`

Roots of a 2nd Degree Equation

```
class  Roots {
    double root_one, root_two;
    int num_roots;
    public roots(double a, double b, double c) {
        double q;
        double r;
        // Apply the textbook quadratic formula:
        // Roots = -b +- sqrt(b^2 - 4ac) / 2a
        q = b*b - 4*a*c;
        if (q > 0 && a != 0) {
            // If b^2 > 4ac, there are two distinct roots
            num_roots = 2;
            r = (double) Math.sqrt(q) ;
            root_one =   ((0-b) + r)/(2*a);
            root_two =   ((0-b) - r)/(2*a);
        } else if (q==0) { // (BUG HERE)
            // The equation has exactly one root
            num_roots = 1;
            root_one = (0-b)/(2*a);
            root_two = root_one;
        } else {
            // The equation has no roots if b^2 < 4ac
            num_roots = 0;
            root_one = -1;
            root_two = -1;
        }
    }
    public int num_roots() { return num_roots; }
    public double first_root()   { return root_one; }
    public double second_root() { return root_two; }
}
```

Roots of a 2nd Degree Equation

We have a function taking three floating point numbers a, b, c. It returns three floating point numbers n, r_1, r_2.

Formally: n is the number of roots of the equation $ax^2 + bx + c = 0$. If $n = 1$, then $r_1 = r_2$ is the root, if $n = 2$ then $r_1 > r_2$ are the two roots, if $n = 0$ then $r_1 = r_2 = -1$.

With details: let $R = \{x \in \mathbb{R} \mid ax^2 + bx + c = 0\}$ and $\Delta = b^2 - 4ac$. Then, $n = |R|$. Furthermore, for $\Delta = 0$, $n = 1$ and $R = \{\xi\}$, $r_1 = r_2 = \xi$. Furthermore, for $\Delta > 0$, $n = 2$ and $R = \{\xi_1, \xi_2\}$, $r_1 = \xi, r_2 = \xi_2$ with $r_1 > r_2$. Finally, for $\Delta < 0$, $n = 0, R = \varnothing$ and $r_1 = r_2 = -1$.

Roots of a 2nd Degree Equation: Category Partition Testing

- For the Category-Partition, let us begin with the characteristics of our three inputs separately (*categorization phase*)
  - interval
  - domain
  - validity
- As for the environment:
  - interval for Δ
- For the *partitioning phase*, see `cases_roots.xls`

Collapsing Spaces Example

```
public static String collapseSpaces(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != ' ' || last != ' ')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```

Collapsing Spaces Example

We have a function taking one string s as an input and returning a string s' as an output.

Informally: s' is the same as s, but *consecutive* spaces are collapsed into one space.

Formally: let $S = \{(i,k) \mid s_i = \text{' '} \wedge (i > 1 \rightarrow s_{i-1} \neq \text{' '}) \wedge k > 1 \wedge \wedge_{n=1}^{k-1} s_{i+n} = \text{' '}\}$. Let
$S' = \{(j,k) \mid \exists (i,k) \in S \wedge j = i - \sum_{(i',k) \in S \mid i<i'} (k-1)\}$. Let
$\sigma(S,j) = \sum_{(i,k) \in S \mid i<j} (k-1)$.
Then, for all $j = 1, \ldots, |s| - \sum_{(i,k) \in S} (k-1)$, $s'_j = s_{j+\sigma(S',j)}$

Collapsing Spaces Example: Catalog-based Testing

- For the Catalog-based Testing, let us begin with the elementary items:
  - Variables
    - s: input string
    - s': output string
  - Definitions
    - a "space" corresponds to ASCII code 0x20
  - Assumed Preconditions:
    - s is a NULL-terminated string of characters
  - Validated Preconditions: NONE
  - Postconditions:
    - if s does not contain occurrences of $n \geq 2$ consecutive spaces, $s' = s$
    - if s contains occurrences of $n \geq 2$ consecutive spaces, s' is initially equal to s, then all consecutive spaces are replaced with one space only
  - Operations
    - scan s, searching for consecutive spaces

Collapsing Spaces Example: Catalog-based Testing

- We now generate test cases, by also specifying from where they come
  - `POST1`: without consecutive spaces, $s' = s$
    - `TC-POST1-1`: s does not contain consecutive spaces
    - `TC-POST1-2`: s contains $n \geq 2$ consecutive spaces
  - `POST2`: with consecutive spaces, in s' they are replaced by single spaces
    - we will obtain the same as before, thus we can skip
  - We are now ready to apply the catalog

Catalog Example

Boolean
- [in/out] True
- [in/out] False

Enumeration
- [in/out] Each enumerated value
- [in] Some value outside the enumerated set

Range $L \ldots U$
- [in] $L-1$ (the element immediately preceding the lower bound)
- [in/out] L (the lower bound)
- [in/out] A value between L and U
- [in/out] U (the upper bound)
- [in] $U+1$ (the element immediately following the upper bound)

Numeric Constant C
- [in/out] C (the constant value)
- [in] $C-1$ (the element immediately preceding the constant value)
- [in] $C+1$ (the element immediately following the constant value)
- [in] Any other constant compatible with C

Non-Numeric Constant C
- [in/out] C (the constant value)
- [in] Any other constant compatible with C
- [in] Some other compatible value

Sequence
- [in/out] Empty
- [in/out] A single element
- [in/out] More than one element
- [in/out] Maximum length (if bounded) or very long
- [in] Longer than maximum length (if bounded)
- [in] Incorrectly terminated

Scan with action on elements P
- [in] P occurs at beginning of sequence
- [in] P occurs in interior of sequence
- [in] P occurs at end of sequence
- [in] PP occurs contiguously
- [in] P does not occur in sequence
- [in] pP where p is a proper prefix of P
- [in] Proper prefix p occurs at end of sequence

Collapsing Spaces Example: Catalog-based Testing

- "Enumeration" may be applied to the definition, thus
  - `TC-DEF-1`: *s* contains a space
    - already inside `TC-POST1-2`, we may skip this
  - `TC-DEF-1`: *s* does not contain a space
- Also "Non-numeric Constant" may be applied to the definition
  - `TC-DEF-2`: *s* contains a TAB
  - `TC-DEF-3`: *s* contains a CR
  - `TC-DEF-4`: *s* contains a LF
  - `TC-DEF-5`: *s* contains a CR+LF

Collapsing Spaces Example: Catalog-based Testing

- "Sequence" may be applied to our string s, thus:
  - `TC-VAR1-1`: s is the empty string
  - s is a string with only one character, i.e.:
    - `TC-VAR1-2-1`: s is a TAB
    - `TC-VAR1-2-2`: s is a CR
    - `TC-VAR1-2-3`: s is a LF
    - `TC-VAR1-2-4`: s is a CR+LF (ok, two characters)
    - `TC-VAR1-2-5`: s is a special character different from above
    - `TC-VAR1-2-6`: s is a non-special character
  - `TC-VAR1-3`: s has length > 1
  - `TC-VAR1-4`: s has a very high length, say > 1000
  - if this is an HTML page, with a limit on s, try a length greater than that limit

Collapsing Spaces Example: Catalog-based Testing

- "Operation" may be applied to our operation, thus:
  - `TC-OP-1`: s begins with two spaces
  - `TC-OP-2`: s contains two spaces
    - might correct `TC-POST1-2`, so as $n > 2$
  - `TC-OP-3`: s ends with two spaces
  - `TC-OP-4`: s contains 4 spaces
    - might correct `TC-POST1-2`, so as $n > 2 \wedge n \neq 4$
  - `TC-OP-5`: s contains 3 spaces
    - might correct `TC-POST1-2`, so as $n \geq 5$
  - `TC-OP-6`: s begins with one space

Another Collapsing Spaces Example

We have a function taking one string s and an integer $n \geq 2$ as an input and returning a string s' as an output.

Informally: s' is the same as s, but k consecutive spaces, s.t. $k \geq n$, are collapsed into one space.

Formally: let $S = \{(i,k) \mid s_i = \text{' '} \wedge (i > 1 \rightarrow s_{i-1} \neq \text{' '}) \wedge k \geq n \wedge \wedge_{n=1}^{k-1} s_{i+n} = \text{' '}\}$. Let
$S' = \{(j,k) \mid \exists (i,k) \in S \wedge j = i - \sum_{(i',k) \in S \mid i<i'} (k-1)\}$. Let
$\sigma(S,j) = \sum_{(i,k) \in S \mid i<j} (k-1)$.
Then, for all $j = 1, \ldots, |s| - \sum_{(i,k) \in S} (k-1)$, $s'_j = s_{j+\sigma(S',j)}$

Another Collapsing Spaces Example: Catalog-based Testing

- Let us add elementary items for input k:
  - Add variable: k, as the input number of consecutive spaces
  - Add assumed precondition: $k \in \mathbb{Z}$
    - might not be true if this is an HTML form...
  - Validated Preconditions: $k \geq 2$
  - Postconditions (updated):
    - if s does not contain occurrences of $n \geq k$ consecutive spaces, $s' = s$
    - if s contains occurrences of $n \geq k$ consecutive spaces, s' is initially equal to s, then all $n \geq k$ consecutive spaces are replaced with one space only
  - Operations (updated):
    - scan s, searching for at least k consecutive spaces

Another Collapsing Spaces Example: Catalog-based Testing

- `TC-POST1-1`: s does not contain $n \geq k$ consecutive spaces
- `TC-POST1-2`: s contains $n \geq k$ consecutive spaces
- `TC-POST3-1`: $k < 2$
- `TC-POST3-2`: $k \geq 2$
- `TC-VAR2-1`: $k = 1$
- `TC-VAR2-2`: $k = 2$
- `TC-VAR2-3`: $k > 2$

Roots of a 2nd Degree Equation

```
class  Roots {
    double root_one, root_two;
    int num_roots;
    public roots(double a, double b, double c) {
        double q;
        double r;
        // Apply the textbook quadratic formula:
        // Roots = -b +- sqrt(b^2 - 4ac) / 2a
        q = b*b - 4*a*c;
        if (q > 0 && a != 0) {
            // If b^2 > 4ac, there are two distinct roots
            num_roots = 2;
            r = (double) Math.sqrt(q) ;
            root_one =   ((0-b) + r)/(2*a);
            root_two =   ((0-b) - r)/(2*a);
        } else if (q==0) { // (BUG HERE)
            // The equation has exactly one root
            num_roots = 1;
            root_one = (0-b)/(2*a);
            root_two = root_one;
        } else {
            // The equation has no roots if b^2 < 4ac
            num_roots = 0;
            root_one = -1;
            root_two = -1;
        }
    }
    public int num_roots() { return num_roots; }
    public double first_root()   { return root_one; }
    public double second_root() { return root_two; }
}
```

Roots of a 2nd Degree Equation

We have a function taking three floating point numbers a, b, c. It returns three floating point numbers n, r_1, r_2.

Formally: n is the number of roots of the equation $ax^2 + bx + c = 0$. If $n = 1$, then $r_1 = r_2$ is the root, if $n = 2$ then $r_1 > r_2$ are the two roots, if $n = 0$ then $r_1 = r_2 = -1$.

With details: let $R = \{x \in \mathbb{R} \mid ax^2 + bx + c = 0\}$ and $\Delta = b^2 - 4ac$. Then, $n = |R|$. Furthermore, for $\Delta = 0$, $n = 1$ and $R = \{\xi\}$, $r_1 = r_2 = \xi$. Furthermore, for $\Delta > 0$, $n = 2$ and $R = \{\xi_1, \xi_2\}$, $r_1 = \xi, r_2 = \xi_2$ with $r_1 > r_2$. Finally, for $\Delta < 0$, $n = 0, R = \varnothing$ and $r_1 = r_2 = -1$.

Roots of a 2nd Degree Equation: Catalog-based Testing

- `VAR1`, `VAR2`, `VAR3`: a, b, c
- `VAR4`, `VAR5`, `VAR6`: n, r_1, r_2
- `DEF1`: $\Delta = b^2 - 4ac$
- `PRE1`, `PRE2`, `PRE3` (assumed or validated): $a, b, c \in \mathbb{R}$
- `POST1`: if $\Delta < 0$, then $n = 0, r_1 = r_2 = -1$
- `POST2`: if $\Delta = 0$, then $n = 1$ and $r_1 = r_2$ are s.t. $ar_1^2 + br_1 + c = 0$
- `POST3`: if $\Delta > 0$, then $n = 2$ and, $\forall r \in \{r_1, r_2\}$, $ar^2 + br + c = 0$
- `OP1`: compute Δ
- `OP2`, `OP3`: compute $\frac{-b \pm \sqrt{\Delta}}{2a}$

Roots of a 2nd Degree Equation: Catalog-based Testing

- From `POST1`: $\Delta < 0$, $\Delta \geq 0$
- From `POST2`: $\Delta = 0$, $\Delta \neq 0$
- From `POST3`: $\Delta > 0$, $\Delta \leq 0$
- Thus, `TC-POST-1`, `TC-POST-2`, `TC-POST-3`: $\Delta < 0, \Delta = 0, \Delta > 0$
- If `PRE1` is validated (same for `PRE2`, `PRE3`):
  - `TC-PRE1-1`: *a* contains multiple dots
  - `TC-PRE1-2`: *a* contains multiple E
  - `TC-PRE1-3`: *a* is bigger than maximum `long double` (if applicable)
  - `TC-PRE1-4`: $a > 0$ is lower than `long double` epsilon (if applicable)
  - `TC-PRE1-5`: *a* contains alphabetic characters (different from E/e)

Pairwise Combination Testing

- Suppose that we have a set of test cases specifications generated by Category-Partition or Catalog-based Testing
- Pure Category-Partition or Catalog-based Testing considers all possible combinations of test case specifications
- Easily, the number of resulting combinations may be intractably high
  - similar to the state space explosion problem in model checking
- Some adjustment may be performed by applying some "reasonable" constraint
- Pairwise Combination Testing offers a systematic way to cut the number of resulting combinations

Pairwise Combination Testing

- Suppose we have $T_1, \ldots, T_n$ "single" test cases specifications (*test factors*)
  - each test factor T_i has $|T_i|$ different choices
  - e.g., in the collapsing spaces example, we have $|T_{\text{length}}| = 4, |T_{\text{spaces}}| = 5, |T_{\text{occConsSp}}| = 3$
  - may also work with "raw" test cases
- Then, the number of combined test case specifications is $\prod_{i=1}^{n} |T_i|$
- With pairwise combination testing, they are usually equal to $|T_M| \cdot |T_m|$, being T_m, T_M the two bigger sets
  - a generic formula for the size is not available
  - however, effective tools are available to generate test cases
  - see, e.g., `https://github.com/microsoft/pict`
- It has been shown that this is a *practical* good solution for testing

Pairwise Combination Testing

- The following properties must be true for the Pairwise Testing result $R \subseteq \prod_{i=1}^{n} T_i$
  - $\forall 1 \leq i < j \leq n$, $\forall (v_1, v_2) \in T_i \times T_j$, $\exists (w_1, \ldots, w_n) \in R$ s.t. $w_i = v_1 \wedge w_j = v_2$
  - that is: for any possible choice of two test factors, all possible pairs of values are present in the result
  - *orthogonal arrays*: all pairs are covered the same number of times
  - for $(v_1, v_2) \in T_i \times T_j$, let $C(v_1, v_2) = \{(w_1, \ldots, w_n) \in R$ s.t. $w_i = v_1 \wedge w_j = v_2\}$
  - then, $\exists p$ s.t., for all $(v_1, v_2) \in T_i \times T_j$, $|C(v_1, v_2)| = p$
- May be generalized to $k \leq n$
  - consider k-tuples instead of pairs
  - $R = \prod_{i=1}^{n} T_i$ if only if $n = k$
- Some Category-Partition-like constraint may still be applied on the result
  - this is what also `pict` allows to do

Structural Testing

- Functional testing is mainly black-box: no need of seeing the actual software
- If the source code is available, something may be done to add some more test cases
  - not necessarily the full "program", also some model may be sufficient
  - e.g., a control flow graph
- Typical question: did we cover all relevant parts of the software?

Structural Testing

- If some statement has never been executed during all tests, that is typically not good
- This happens if the statement is executed only if C holds, and for all test cases C does not hold
  - inside an `if`, but it is not the only case
  - may be a `while`, or after a `return`...
- Of course, it may happen that a statement will never be executed at all (*unreachable code*)
- Of course, having all statements executed does not guarantee errors are cought
  - it may be the case that only given inputs trigger a failure
  - or the implementation may be faulty w.r.t. the specification, not considering some cases
  - so, structural testing typically *complements* functional testing

Structural Testing and Adequacy Criteria

- One major usage of structural testing is to define *adequacy criteria*
- That is: suppose we have already generated a test suite for our ITF
- Is this test suite adequate w.r.t. some measurable criteria?
- In the following, we will mainly provide definitions for meaningful criteria which exploits program source code (mainly its CFG)
- Is it possible to reverse this reasoning? That is:
  1. we select an adequacy criterion
  2. we generate test case specifications which pass the criterion
  3. we generate test cases fulfilling the test case specifications

Structural Testing and Adequacy Criteria

- The step 3 above is all but simple
- As an example, some adequacy criteria require to make a given condition true or false
  - a condition may be used by an `if` or a `while`
- Of course, in the general case this is an undecidable problem
  - that is: given a program and a condition inside it, make the condition be true/false
  - some of the difficulties: it may be the case that the variables are changed before arriving to the condition, thus a reverse engineering is required
  - the condition may involve a function call
  - the condition may be unreachable for the selected values
  - ...
- Human effort is required, or sub-optimal solutions must be accepted

Statement Testing

- Faults are in statements
  - also including expression evaluations
- A fault in a statement cannot be revealed without executing the faulty statement
- A test suite T for a program P satisfies the *statement adequacy criterion* for P, iff, for each statement S of P, there exists at least one test case in T that causes the execution of S
  - i.e., every node in the control flow graph of P is visited by some execution path exercised by a test case in T
- If not all statements, let us measure how many of them: *statement coverage* $C_S = \frac{\#\text{execd statm}}{\#\text{all statm}}$

Statement and Block Testing

- If the CFG is provided, then "blocks" are considered instead of "statements"
  - depending on CFG granularity
- If $C_{Ss}(T), C_{Sb}(T)$ are statement and block coverage for a test suite T, and $C_{Ss}(T_1) > C_{Ss}(T_2)$, then $C_{Sb}(T_1) > C_{Sb}(T_2)$
  - there is a kind of monotonicity
- On the other hand, test suites with fewer test cases may achieve a higher coverage than test suites with more
  - e.g., because the input is a string, so having just few long strings may be enough, whilst having many short strings may miss some statement

How to Measure Adequacy?

- Suppose we have a code and a test suite, how can we actually measure statement coverage?
- We have to instrument the code: see `collapse_spaces.java`
- It may also be done by existing tools: e.g., CTC++
  - works for C, C++ and Java
  - commercial product, works on all platforms
  - enhanced compiler: you invoke CTC++ compiler and run the executable as many times you want
  - CTC++ additional tools are available to analyze coverages
  - not only statement coverage, also the following ones
- As far as the lecturer knows, no free-to-use tools exists for code coverage instrumentation

Branch Testing

- If statement coverage is full, this means that all conditions are evaluated at least once
- However, this does not imply that all *branches* are considered
  - a branch is an edge between two blocks, traversed iff some condition holds
- That is, that all conditions are evaluated at least once true and at least once false
- In fact, an *else* may be missing, thus a perfect statement coverage test suite may not consider the condition being false
- This may be a problem in many cases

Branch Testing

- T satisfies the *branch adequacy criterion* for P iff, for each branch B of P, there exists at least one test case in T that causes execution of B
- In the CFG, all edges must be exercised by some test case in T
- Branch coverage ratio: $C_B = \frac{\#\text{execd branches}}{\#\text{all branches}}$

Condition Testing

- If conditions are composed by multiple atomic propositions, branch coverage could be not enough
  - e.g., a condition $A \wedge B$, with A always true, may pass the branch coverage criterion...
- Thus, we want all atomic propositions in all conditions to be evaluated at least once true and at least once false
- T satisfies the *condition adequacy criterion* for P iff, for each basic condition C in P, C has a true outcome in at least one test case and a false outcome in at least one test case in T
- Cannot be directly observed in CFGs
- Basic condition coverage ratio: $C_{BC} = \frac{\#\text{resulting truth values}}{2\#\text{all basic conditions}}$

Branch and Condition Testing

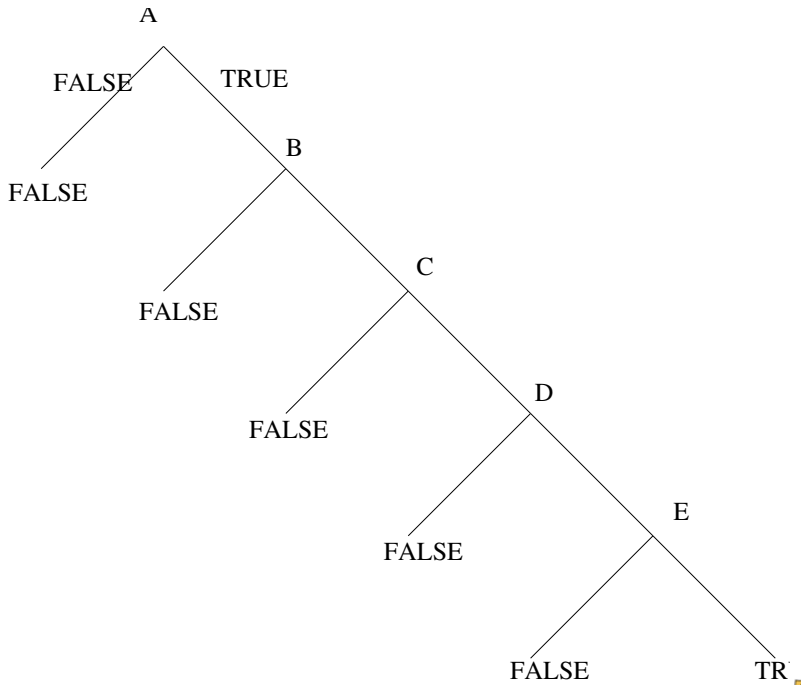
- We already saw that branch coverage does not imply condition coverage
- Neither the viceversa holds: for example, $A \wedge B$ may be exercised by $A = 1, B = 0$ and $A = 0, B = 1$, which is ok for condition but not for branch
- *Branch and condition adequacy criterion*: satisfied only if both condition and branch coverage are satisfied
- Both coverage ratios can be considered

Compound Condition Testing

- A more "natural" way to deal with both branch and condition coverage
- *Compound condition adequacy criterion*: obtain the abstract evaluation tree of each expression, then each branch of such tree must be covered
  - in an abstract evaluation tree (AET), internal nodes are labeled with conditions, while edges and leaves are labeled with true or false
  - it is the same as OBDDs, with conditions instead of variables
- Thus, there is a test case specification for each path from the root to a leaf
- In the worst case, it is exponential

Compound Condition Testing: Examples

$$A \wedge B \wedge C \wedge D \wedge E$$



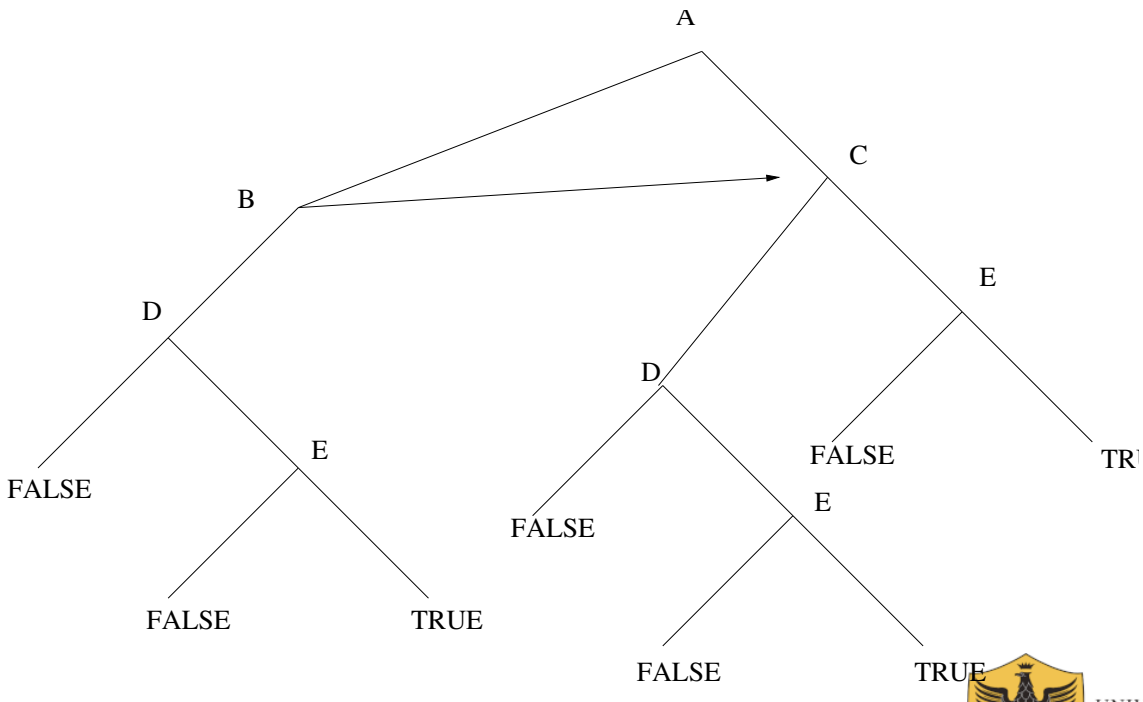
Compound Condition Testing: Examples

Compound condition for $A \wedge B \wedge C \wedge D \wedge E$

Test Case	a	b	c	d	e
(1)	*True*	*True*	*True*	*True*	*True*
(2)	*True*	*True*	*True*	*True*	*False*
(3)	*True*	*True*	*True*	*False*	–
(4)	*True*	*True*	*False*	–	–
(5)	*True*	*False*	–	–	–
(6)	*False*	–	–	–	–

Compound Condition Testing: Examples

$$(((A \vee B) \wedge C) \vee D) \wedge E$$



Compound Condition Testing: Examples

Compound condition for $(((A \vee B) \wedge C) \vee D) \wedge E$

Test Case	a	b	c	d	e
(1)	*True*	–	*True*	–	*True*
(2)	*False*	*True*	*True*	–	*True*
(3)	*True*	–	*False*	*True*	*True*
(4)	*False*	*True*	*False*	*True*	*True*
(5)	*False*	*False*	–	*True*	*True*
(6)	*True*	–	*True*	–	*False*
(7)	*False*	*True*	*True*	–	*False*
(8)	*True*	–	*False*	*True*	*False*
(9)	*False*	*True*	*False*	*True*	*False*
(10)	*False*	*False*	–	*True*	*False*
(11)	*True*	–	*False*	*False*	–
(12)	*False*	*True*	*False*	*False*	–
(13)	*False*	*False*	–	*False*	–

MC/DC Testing

- Compound condition may require 2^N test cases, if there are N conditions
- Compound condition heavily depends on the structure itself of the condition
  - $A \wedge B \wedge C \wedge D \wedge E$ requires 6 test cases
  - $(((A \vee B) \wedge C) \vee D) \wedge E$ requires 13 test cases
- Modified Condition/Decision Coverage (MC/DC) overcomes this problem
- It may be proved that, if N is the number of basic condition, then at most $N + 1$ test cases are needed for MC/DC testing
- Thus, $(((A \vee B) \wedge C) \vee D) \wedge E$ requires 6 test cases
- Can be computed automatically, given the abstract evaluation tree
- Required by many testing standards, e.g., in Aviation

MC/DC Testing: Definition and Properties

- *Condition*: atomic proposition
  - no boolean operators occur
  - e.g.: `a <= b`
- *Decision*: a whole boolean expression
  - a boolean combination of conditions
- Each decision in the program under test has taken all possible outcomes at least once
- Each condition in a decision has taken all possible outcomes at least once
- Each condition in a decision affects independently and correctly the outcome of this decision

MC/DC Testing

Compound condition for $A \wedge B \wedge C \wedge D \wedge E$

Test Case	a	b	c	d	e
(1)	*True*	*True*	*True*	*True*	*True*
(2)	*True*	*True*	*True*	*True*	*False*
(3)	*True*	*True*	*True*	*False*	–
(4)	*True*	*True*	*False*	–	–
(5)	*True*	*False*	–	–	–
(6)	*False*	–	–	–	–

MC/DC Testing

Compound condition for $(((A \vee B) \wedge C) \vee D) \wedge E$

Test Case	a	b	c	d	e
(1)	*True*	–	*True*	–	*True*
(2)	*False*	*True*	*True*	–	*True*
(3)	*True*	–	*False*	*True*	*True*
(4)	*False*	*True*	*False*	*True*	*True*
(5)	*False*	*False*	–	*True*	*True*
(6)	*True*	–	*True*	–	*False*
(7)	*False*	*True*	*True*	–	*False*
(8)	*True*	–	*False*	*True*	*False*
(9)	*False*	*True*	*False*	*True*	*False*
(10)	*False*	*False*	–	*True*	*False*
(11)	*True*	–	*False*	*False*	–
(12)	*False*	*True*	*False*	*False*	–
(13)	*False*	*False*	–	*False*	–

MC/DC Testing

MC/DC condition for $(((A \vee B) \wedge C) \vee D) \wedge E$

	a	b	c	d	e	Decision
(1)	*True*	–	*True*	–	*True*	*True*
(2)	*False*	*True*	*True*	–	*True*	*True*
(3)	*True*	–	*False*	*True*	*True*	*True*
(6)	*True*	–	*True*	–	*False*	*False*
(11)	*True*	–	*False*	*False*	–	*False*
(13)	*False*	*False*	–	*False*	–	*False*

Each condition in a decision has taken all possible outcomes at least once $\rightarrow$ simply look at columns

Each condition in a decision affects independently and correctly the outcome of this decision $\rightarrow$ row pairs corresponding to underlined items only differs for exactly one condition (the underlined one)

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Algorithm for MC/DC Testing

```
GenMCDCCases(D) {
  γ, C ← compoundCondition(D);
  // C: set of conditions on which D depends
  ok ← ∅;
  for each condition C ∈ C {
    for each pair (i, j) ∈ |γ|², i < j {
      let r1, r2 be the i and j-th row of γ;
      // recall that don't cares match all
      if (γ(r1, C) ≠ γ(r2, C) ∧ D(r1) ≠ D(r2) ∧
          ∀C' ∈ C. C' ≠ C → γ(r1, C') = γ(r2, C')) {
        ok ← ok ∪ {r1, r2};
        break;
      }
    }
  }
  return γ \ rows not in ok;
}
```

Path Testing

- Suppose we have the CFG of a program: we may consider *paths* in it
  - starting from the root and having some finite length
  - finite length is required as testing must provide an answer at some time...
- Path adequacy criterion: for each path p of P, there exists at least one test case in T that causes the execution of p
- That is, every path p is exercised by a test case in T
- Path coverage is defined as $C_P = \frac{\#\text{execd paths}}{\#\text{all paths}}$
- If the CFG has loops, the denominator is infinite, thus $C_P = 0$
  - all non-trivial CFGs have loops...
- A form of path coverage may be achieved with model checking

Practical Path Coverage

- Loop boundary adequacy criterion: for each loop p in P containing a loop l, the following holds
  - in at least one execution, control reaches the loop, and then the loop control condition evaluates to False the first time it is evaluated
  - in at least one execution, control reaches the loop, and then the body of the loop is executed exactly once before control leaves the loop
  - in at least one execution, the body of the loop is repeated more than once
- Thus, we execute 0, 1 or many times the loop
- The intuition is that the loop boundary coverage criteria reflect a deeper structure in the design of a program

Practical Path Coverage

- Linear Code Sequence and Jump (LCSAJ): a body of code through which the flow of control may proceed sequentially, terminated by a jump in the control flow
- We may define sequences of LCSAJs
  - sequences of length 1 are almost equivalent to branch coverage (excluding some ill-based code)
- TER_N, for $N \geq 1$, is the Test Effectiveness Ratio
  - $\text{TER}_1 = T_S$ (statements)
  - $\text{TER}_2 = T_B$ (branches)
  - $\text{TER}_{i+2} = \frac{\#\text{execd } i \text{ consecutive LCSAJs}}{\#\text{all } i \text{ consecutive LCSAJs}}$
- $\text{TER}_N = 1$ implies $TER_i = 1$ for all $i < N$
  - TER_i with $i > 3$ only required by very critical systems

Practical Path Coverage

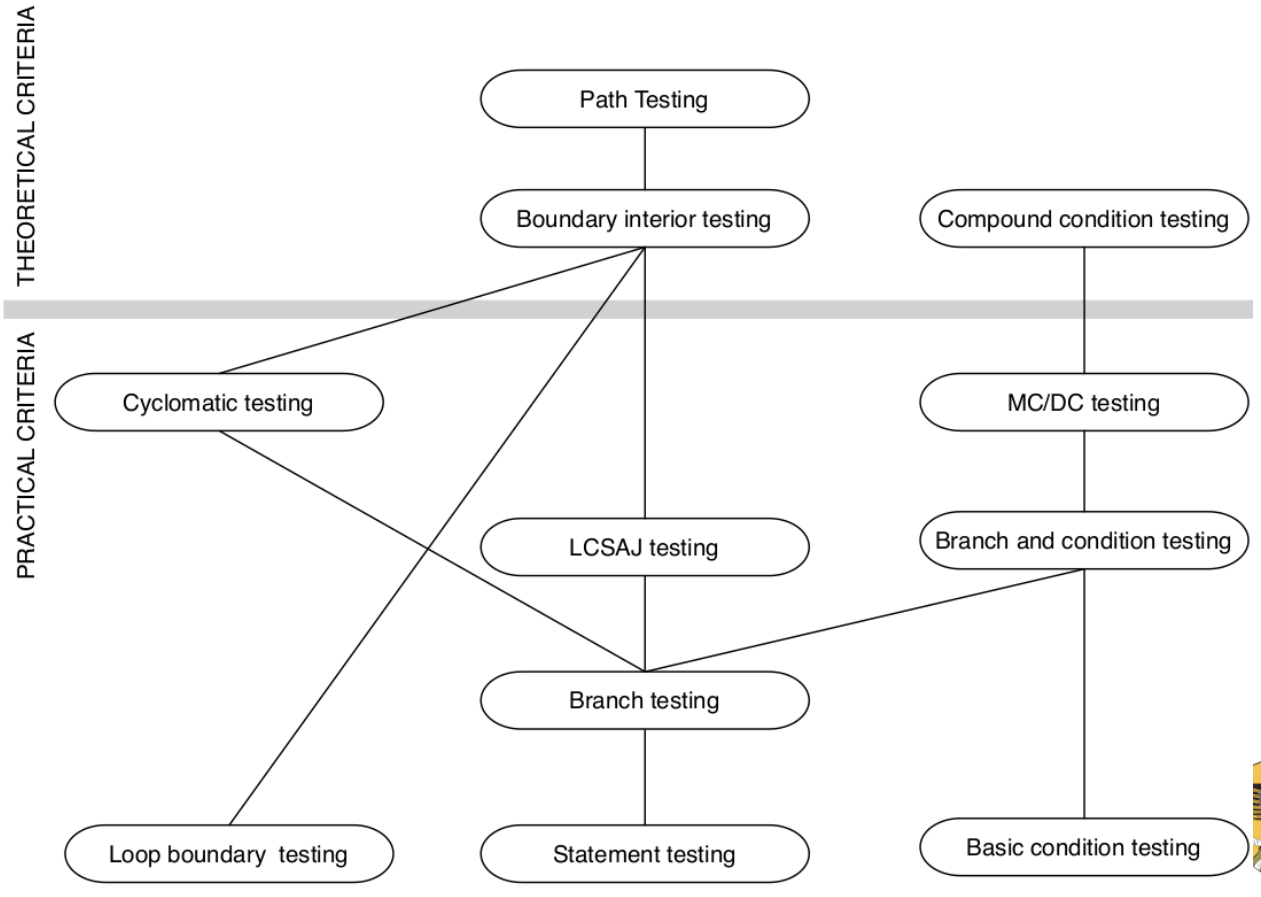
- Cyclomatic testing, based on the definition of *basis set*
  - let $\mathcal{P}$ be the set of all paths
  - $B \subseteq \mathcal{P}$ is a basis set iff, for all $p \in \mathcal{P}$, p is the concatenation of some $q \in B$
  - it can be proved that $|B| = e - n + 2c$
    - e, n are number of edges and nodes
    - c is the number of strongly connected components
  - for a procedure, just connect the exit back to the entry to obtain $c = 1$ so $|B| = e - n + 2$
  - this is the *cyclomatic complexity* of the program
- Cyclomatic testing: exercise every path in the basis set at least once

Procedure Call Testing

- The previous techniques are ok for single procedures/functions
- When it comes to integration or system testing, it is needed to put the single pieces together
- Call coverage: exercise all different calls to C
  - calling the same procedure twice in different points counts as two
- Good news: if C is called by A and B only, and statement coverage of A and B has already been completed, then we are done!
- Bad news: for procedures with side effects, *call sequences* are important
  - especially true for object oriented programming, we will be back on this

Subsumptions

Test coverage criterion A subsumes test coverage criterion B iff, for every program P, every test set satisfying A with respect to P also satisfies B with respect to P.



Data Flow Testing

- Again, white-box: it aids path coverage
  - paths are selected basing on how one syntactic element can affect the computation of another
  - criteria based on control structure alone fail on considering data interactions
- Computing the wrong value leads to a failure only when that value is subsequently used
- Data flow testing ensures that each computed value is actually used
- Thus, paths more likely to lead to failures are considered

Definition-Use Associations

- Data flow testing is based on definition-use pair
  - recall: a definition writes a value in a variable, a use reads a value from a variable
  - a definition-use pair consider a definition and a following use which is not killed by another definition
- Each definition-use pair defines a definition-clear path
  - there may be several uses after the definition
- A static data flow analyzer is needed
  - for not-too-big procedures, manual instrumentation is also possible

Data Flow Testing Criteria

- *All DU pairs adequacy criterion*: each DU pair must be exercised in at least one program execution
  - an erroneous value in a definition might be revealed only by its use
- A test suite T for a program P satisfies the all DU pairs adequacy criterion iff, for each DU pair (d, u) of P, at least one test case in T exercises (d, u)
- Unsurprisingly, $C_{DU} = \frac{\#\text{execd DU pairs}}{\#\text{all DU pairs}}$
- Finer than statement and branch coverage

Data Flow Testing Criteria

```
int x = 1, y = 2;
if (y <= 2) // or while
  y++;
y = x;
```

- *All DU paths adequacy criterion*: for each DU pair, each of the corresponding DU paths must be exercised in at least one program execution
  - if the path contains a loop, discard the loop (simple path)
- A test suite T for a program P satisfies the all DU paths adequacy criterion iff, for each DU pair (d, u) of P and simple path p from d to u, at least one test case in T exercises p
- Unsurprisingly, $C_{DUP} = \frac{\#\text{execdDU simple paths}}{\#\text{all DU simple paths}}$
- Of course, subsumes the all DU pairs coverage criterion

Data Flow Testing Criteria

- *All definitions adequacy criterion*: for each definition, at least one corresponding use must be exercised in at least one program execution
- A test suite T for a program P satisfies the all definitions adequacy criterion iff, for each definition d of P, there exists a DU pair (d, u) s.t. at least one test case in T exercises (d, u)
- Unsurprisingly, $C_D = \frac{\#\text{covered definition}}{\#\text{all definitions}}$
- Of course, it is subsumed by both the all DU pairs and all DU paths coverage criterion

Model Based Testing

- Kind of black-box testing, where specifications have some special form
  - or it is possible to extract a model from specifications
  - similar to model checking, but model is then used for testing
- From models of expected behavior to test case specifications
  - to detect discrepancies between actual program behavior and the model
- Used to aid black-box approaches to identify
  - meaningful values
  - (additional) constraints
  - (additional) significant combinations

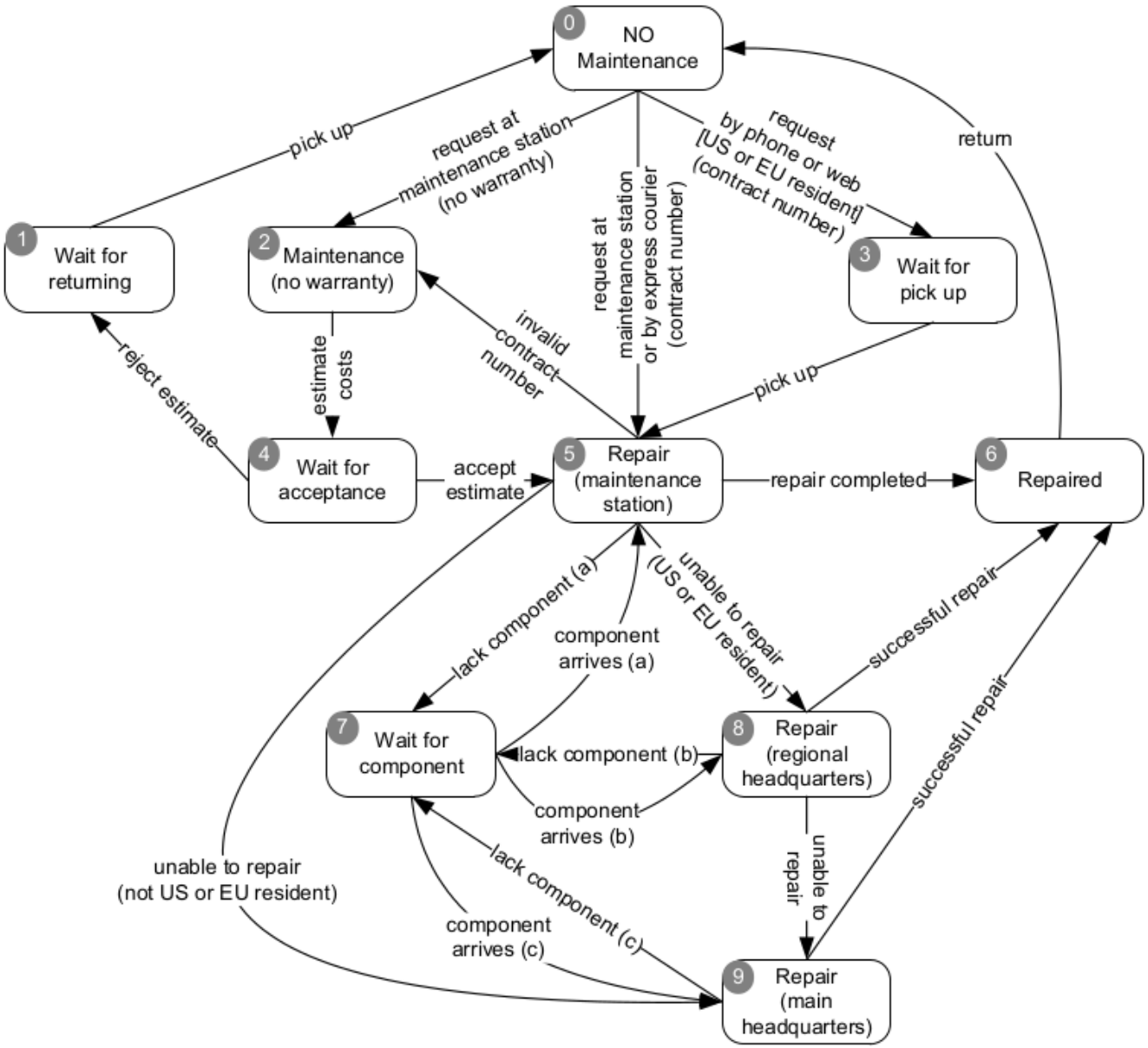
Model Based Testing

- Two types of models
  - formal, i.e., with a precise syntax: e.g., finite state machines
    - from a formal model, test cases may be automatically generated
  - semiformal: e.g., class diagrams
    - from a semiformal model, automation must be used with some care
    - note that finite state machines may be semiformal, see below
- Models (also) describe input structure
- Discrepancies from the model can be used as an implicit fault model to help identify boundary and error cases
- We will consider 4 models and see how to generate test cases from each of them

Finite States Machines

- Common for control and reactive systems, such as
  - embedded systems (StateChart), communication protocols (SDL), menu-driven applications
  - typically multiprocess or multithread
- Many systems actually have infinite states, but often are approximable with a finite state machine as well
  - for real, a port receiving a string message should have infinite states...
- Transitions are usually guarded by conditions or input events
  - conditions may be regarded as particular input events

Example


0 NO Maintenance
1 Wait for returning
2 Maintenance (no warranty)
3 Wait for pick up
4 Wait for acceptance
5 Repair (maintenance station)
6 Repaired
7 Wait for component
8 Repair (regional headquarters)
9 Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web [US or EU resident] (contract number)
return
reject estimate
estimate costs
invalid contract number
pick up
accept estimate
repair completed
lack component (a)
component arrives (a)
unable to repair (US or EU resident)
successful repair
lack component (b)
component arrives (b)
unable to repair
successful repair
unable to repair (not US or EU resident)
lack component (c)
component arrives (c)

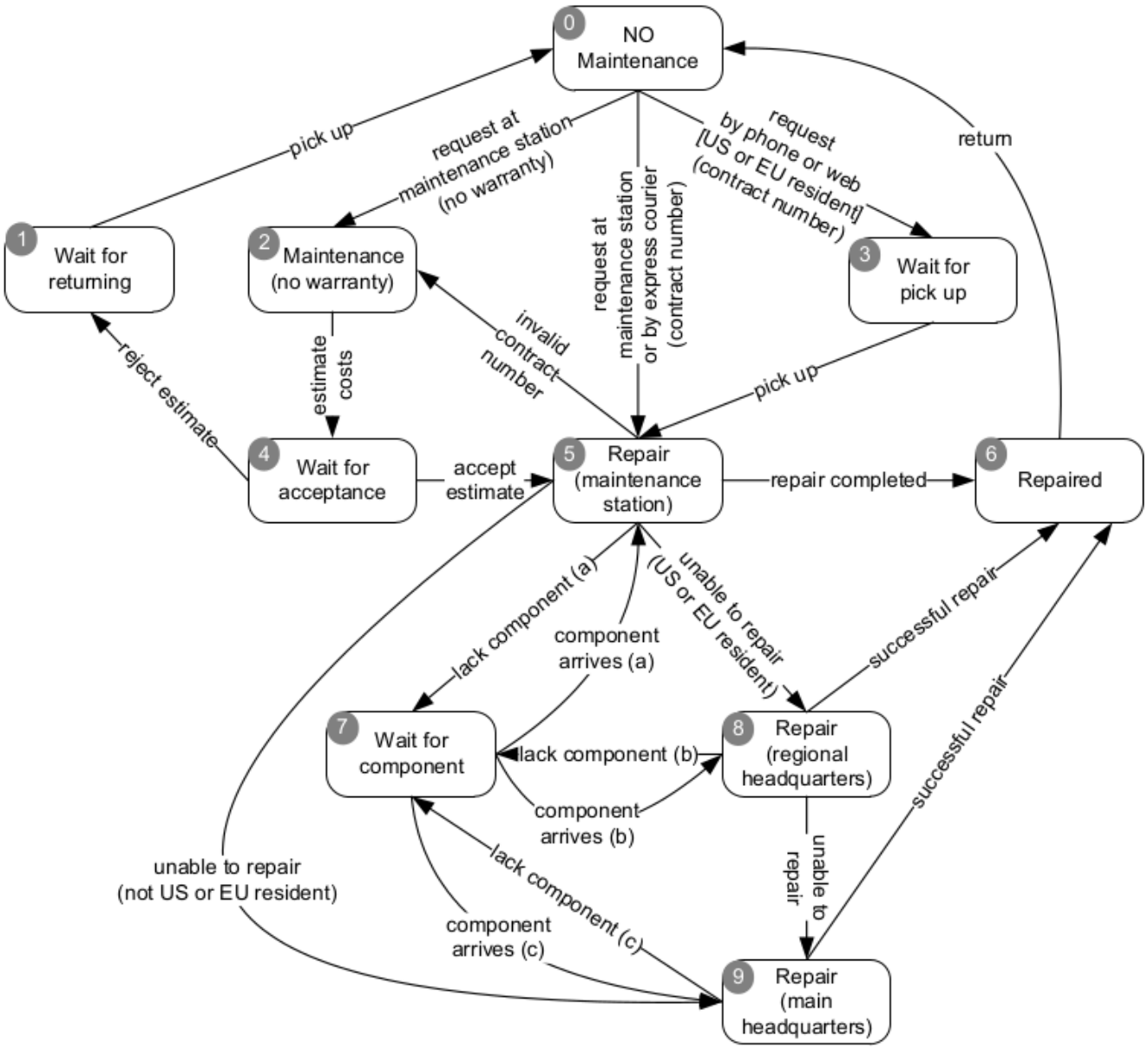
Semiformal Finite States Machines

- Sometimes they are not memoryless as they should be
  - transitions from a state must only depend on the starting state
  - instead, often it depends also on the path leading to the state (memory)
  - e.g., "Wait for component" need to remember *which* component...
- Some outgoing transitions may be missing, i.e. some input is not considered from some state
- Three possible cases:
  - don't care transition, that input is impossible in that state, e.g., because of some physical constraint
  - error transition, goes to some common error-handling procedure
  - self transition
- Though they may be "completed", also a semiformal FSS may be useful

From Finite States Machine To Test Case Specification

- Each transition should be covered
  - could be seen as a (precondition, postcondition) pair...
- Transition coverage: all transitions are covered
- Unsurprisingly, $C_T = \frac{\#\text{execd transitions}}{\#\text{all transitions}}$
- Looks similar to white-box testing, but here this is applied *in advance*
  - you decide an acceptable C_T, and generate test case specifications accordingly
- Final result is test cases specifications involving transitions
  - obtaining test cases could be not simple
  - depends on the specific program under test

Example


0 NO Maintenance
1 Wait for returning
2 Maintenance (no warranty)
3 Wait for pick up
4 Wait for acceptance
5 Repair (maintenance station)
6 Repaired
7 Wait for component
8 Repair (regional headquarters)
9 Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web [US or EU resident] (contract number)
return
reject estimate
estimate costs
invalid contract number
pick up
accept estimate
repair completed
lack component (a)
component arrives (a)
unable to repair (US or EU resident)
successful repair
lack component (b)
component arrives (b)
unable to repair
successful repair
unable to repair (not US or EU resident)
lack component (c)
component arrives (c)

Example

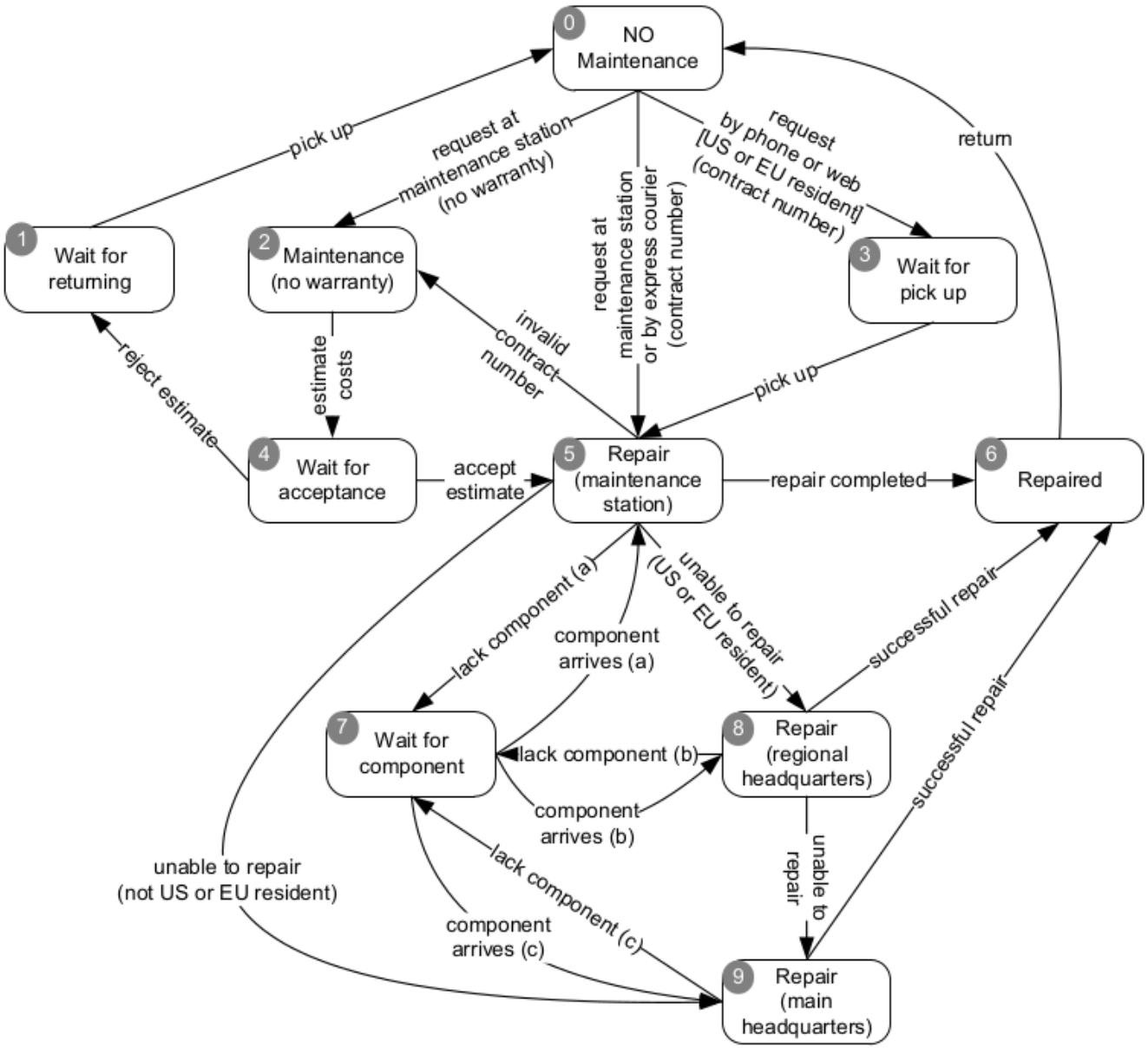
	T-Cover
TC-1	0 – 2 – 4 – 1 – 0
TC-2	0 – 5 – 2 – 4 – 5 – 6 – 0
TC-3	0 – 3 – 5 – 9 – 6 – 0
TC-4	0 – 3 – 5 – 7 – 5 – 8 – 7 – 8 – 9 – 7 – 9 – 6 – 0

States numbers refer to Figure 14.2. For example, TC-1 represents the path (0,2), (2,4), (4,1), (1,0).

From Finite States Machine To Test Case Specification

- For small FSS, also paths may be considered, especially if they are semiformal
  - single state path coverage criterion: all non-loop paths
  - single transition path coverage criterion: all paths in which each transition is taken just once
  - interior boundary loop coverage: for all loops, exercise the loop the minimum, the maximum, and some intermediate number of times
  - the corresponding coverage ratios may be defined
- Especially useful when states do not fully describe the system status

Example


0 NO Maintenance
1 Wait for returning
2 Maintenance (no warranty)
3 Wait for pick up
4 Wait for acceptance
5 Repair (maintenance station)
6 Repaired
7 Wait for component
8 Repair (regional headquarters)
9 Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web [US or EU resident] (contract number)
return
reject estimate
estimate costs
invalid contract number
pick up
accept estimate
repair completed
lack component (a)
component arrives (a)
unable to repair (US or EU resident)
successful repair
lack component (b)
component arrives (b)
unable to repair
successful repair
unable to repair (not US or EU resident)
lack component (c)
component arrives (c)

Example

	T-Cover
TC-1	0 – 2 – 4 – 1 – 0
TC-2	0 – 5 – 2 – 4 – 5 – 6 – 0
TC-3	0 – 3 – 5 – 9 – 6 – 0
TC-4	0 – 3 – 5 – 7 – 5 – 8 – 7 – 8 – 9 – 7 – 9 – 6 – 0

States numbers refer to Figure 14.2. For example, TC-1 represents the path (0,2), (2,4), (4,1), (1,0).

- single state path coverage criterion: e.g., 0-2-4-5-3 is missing
- single transition path coverage criterion: e.g., 0-2-4-1-0-5-2 is missing
- interior boundary loop coverage: e.g., 0-2-4-1-0-2-4-1 is missing

Decision Structures

- From a functional specification to a *decision table*
- Possible intermediate step: from the specification, write a Boolean formula
  - first order logic: boolean combinations of propositions
- To be done when a (part of a) specification is clearly based on some complex Boolean formula

Decision Structures

Pricing: The *pricing* function determines the adjusted price of a configuration for a particular customer. The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.

There are three price schedules and three corresponding discount schedules, *Business*, *Educational*, and *Individual*. The Business price and discount schedules apply only if the order is to be charged to a business account in good standing. The Educational price and discount schedules apply to educational institutions. The Individual price and discount schedules apply to all other customers. Account classes and rules for establishing business and educational accounts are described further in [. . .].

A discount schedule includes up to three discount levels, in addition to the possibility of "no discount." Each discount level is characterized by two threshold values, a value for the current purchase (configuration schedule price) and a cumulative value for purchases over the preceding 12 months (sum of adjusted price).

Educational prices The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.

Business account discounts Business discounts depend on the size of the current purchase as well as business in the preceding 12 months. A tier 1 discount is applicable if the scheduled price of the current order exceeds the tier 1 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 1 year cumulative value threshold. A tier 2 discount is applicable if the current order exceeds the tier 2 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 2 cumulative value threshold. A tier 2 discount is also applicable if both the current order and 12 month cumulative payments exceed the tier 1 thresholds.

Individual discounts Purchase by individuals and by others without an established account in good standing is based on current value alone (not on cumulative purchases). A tier 1 individual discount is applicable if the scheduled price of the configuration in the current order exceeds the tier 1 current order threshold. A tier 2 individual discount is applicable if the scheduled price of the configuration exceeds the tier 2 current order threshold.

Special-price nondiscountable offers Sometimes a complete configuration is offered at a special, non-discountable price. When a special, nondiscountable price is available for a configuration, the adjusted price is the nondiscountable price or the regular price after any applicable discounts, whichever is less.

Decision Structures

Output is "no discount" IFF

$$
\begin{array}{llll}
 & (& & \text{individual account} \\
 & & \wedge\neg & \text{current purchase} > \text{tier 1 individual threshold} \\
 & & \wedge\neg & \text{special offer price} < \text{individual scheduled price} \quad) \\
\vee & (& & \text{business account} \\
 & & \wedge\neg & \text{current purchase} > \text{tier 1 business threshold} \\
 & & \wedge\neg & \text{current purchase} > \text{tier 1 business yearly threshold} \\
 & & \wedge\neg & \text{special offer price} < \text{business scheduled price} \quad)
\end{array}
$$

Corresponds to fourth column in the first table and second column in the second table (next slide)

Decision Structures

Abbreviations

EduAc	Educational account	Edu	Educational price
BusAc	Business account	ND	No discount
CP $>$ CT1	Current purchase greater than threshold 1	T1	Tier 1
YP $>$ YT1	Year cumulative purchase greater than threshold 1	T2	Tier 2
CP $>$ CT2	Current purchase greater than threshold 2	SP	Special Price
YP $>$ YT2	Year cumulative purchase greater than threshold 2		
SP $>$ Sc	Special Price better than scheduled price		
SP $>$ T1	Special Price better than tier 1		
SP $>$ T2	Special Price better than tier 2		

Example

	Education		**Individual**					
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

	Business											
EduAc	-	-	-	-	-	-	-	-	-	-	-	-
BusAc	T	T	T	T	T	T	T	T	T	T	T	T
CP > CT1	F	F	T	T	F	F	T	T	-	-	-	-
YP > YT1	F	F	F	F	T	T	T	T	-	-	-	-
CP > CT2	-	-	F	F	-	-	-	-	T	T	-	-
YP > YT2	-	-	-	-	F	F	-	-	-	-	T	T
SP > Sc	F	T	-	-	-	-	-	-	-	-	-	-
SP > T1	-	-	F	T	F	T	-	-	-	-	-	-
SP > T2	-	-	-	-	-	-	F	T	F	T	F	T
Out	ND	SP	T1	SP	T1	SP	T2	SP	T2	SP	T2	SP

Constraints

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

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

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Decision Structures

- A decision table directly corresponds to a Boolean formula of the form $\wedge_{i=1}^{n}((\wedge_{j=1}^{m}\beta_{i,j}) \rightarrow v = x_i)$
  - v is a variable representing the output of some part of the program, and x_i the desired outcome
    - typically, some enumerated value
  - $n+1$ columns, $m+1$ rows
    - first column lists all "basic conditions" $b_1, \ldots, b_m$
    - last row is for values of v
    - other heading are possible to ease table understanding
  - each $\beta_{i,j}$ is either
    - b_i, if the entry i, j in the table is T
    - $\neg b_i$, if the entry i, j in the table is F
    - void, if the entry i, j in the table is don't care
  - easy to transform in conjunctive normal form: $\wedge_{i,=1}^{n}(\vee_{j=1}^{k_i}\neg\beta_{i,j} \vee v_i = x_i)$
  - each column i (rule) corresponds to $\wedge_{j=1}^{m}\beta_{i,j} \rightarrow v_i = x_i)$

Decision Tables

- Tables are typically augmented with *constraints*
  - other Boolean formulas on b_i, typically excluding invalid combinations
  - most typical constraint are abbreviated, e.g., *at-most-one* or *exactly one*
- Thus, the overall formula is $\wedge_{i=1}^{n}((\wedge_{j=1}^{m}\beta_{i,j}) \rightarrow v = x_i) \wedge \wedge_{i=1}^{k} C_i$
  - e.g., if C_i is an at-most-one(B) constraint, being $B \subset \{b_1, \ldots, b_m\}$, then $C_i \equiv \wedge_{\ell=1}^{n} \sum_{j \mid b_j \in B} D_{j,\ell} \leq 1$
- A new table can be written by taking into account the additional constraints
  - essentially, this entails fixing some don't cares

Example

	Education		**Individual**					
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

	Business											
EduAc	-	-	-	-	-	-	-	-	-	-	-	-
BusAc	T	T	T	T	T	T	T	T	T	T	T	T
CP > CT1	F	F	T	T	F	F	T	T	-	-	-	-
YP > YT1	F	F	F	F	T	T	T	T	-	-	-	-
CP > CT2	-	-	F	F	-	-	-	-	T	T	-	-
YP > YT2	-	-	-	-	F	F	-	-	-	-	T	T
SP > Sc	F	T	-	-	-	-	-	-	-	-	-	-
SP > T1	-	-	F	T	F	T	-	-	-	-	-	-
SP > T2	-	-	-	-	-	-	F	T	F	T	F	T
Out	ND	SP	T1	SP	T1	SP	T2	SP	T2	SP	T2	SP

Constraints

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

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

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Example

EduAc	T	T	F	F	F	F	F	F	F	F	F	F	F
BusAc	F	F	F	F	F	F	F	F	T	T	T	T	T
CP > CT1	T	T	F	F	T	T	T	T	F	F	T	T	F
YP > YT1	F	-	F	-	-	F	T	T	F	F	F	F	T
CP > CT2	F	F	F	F	F	F	T	T	F	F	F	F	F
YP > YT2	-	-	-	-	-	-	-	-	-	-	-	-	F
SP > Sc	F	T	F	T	F	T	-	-	F	T	F	-	F
SP > T1	F	T	F	T	F	T	F	T	F	T	F	T	F
SP > T2	F	-	F	-	F	-	F	T	F	-	F	-	F
Out	Edu	SP	ND	SP	T1	SP	T2	SP	ND	SP	T1	SP	T1

EduAc	F	F	F	F	F	T	T	T	T	F	-
BusAc	T	T	T	T	T	F	F	F	F	F	F
CP > CT1	T	T	T	F	F	F	F	T	-	-	F
YP > YT1	T	F	F	T	T	T	-	-	-	T	T
CP > CT2	F	T	T	F	F	F	F	T	T	F	F
YP > YT2	F	-	-	T	T	F	-	-	-	T	F
SP > Sc	T	-	T	-	T	F	T	-	-	-	-
SP > T1	T	F	T	F	T	F	-	-	T	T	T
SP > T2	T	F	T	F	T	F	F	F	T	T	-
Out	SP	T2	SP	T2	SP	Edu	SP	Edu	SP	SP	SP

Decision Tables Adequacy Criteria

- *Basic condition adequacy criterion*: one test case specification for each column in the table.
  - don't cares replaced with any value, but without violating the additional constraints
- *Compound condition adequacy criterion*: one test case specification for each combination of truth values of basic conditions
  - entails 2^n test cases specification, only for small tables
  - table is used only to compute the corresponding output value

Decision Tables Adequacy Criteria

- *Modified Condition/Decision Criterion (MC/DC)*
  - similar, but not the same to the structural approach with the same name
  - indeed, it cannot be the same as the result is not a boolean...
- First, some new columns must be added to the original table
  - but if they are already present, you can skip them
- For each column c and for each b_i in c which is not don't care:
  - obtain a new column equal to c, but where b_i is negated w.r.t. c
- Finally, one test case specification for each resulting column
  - similar to the basic condition, but on the augmented table

Example

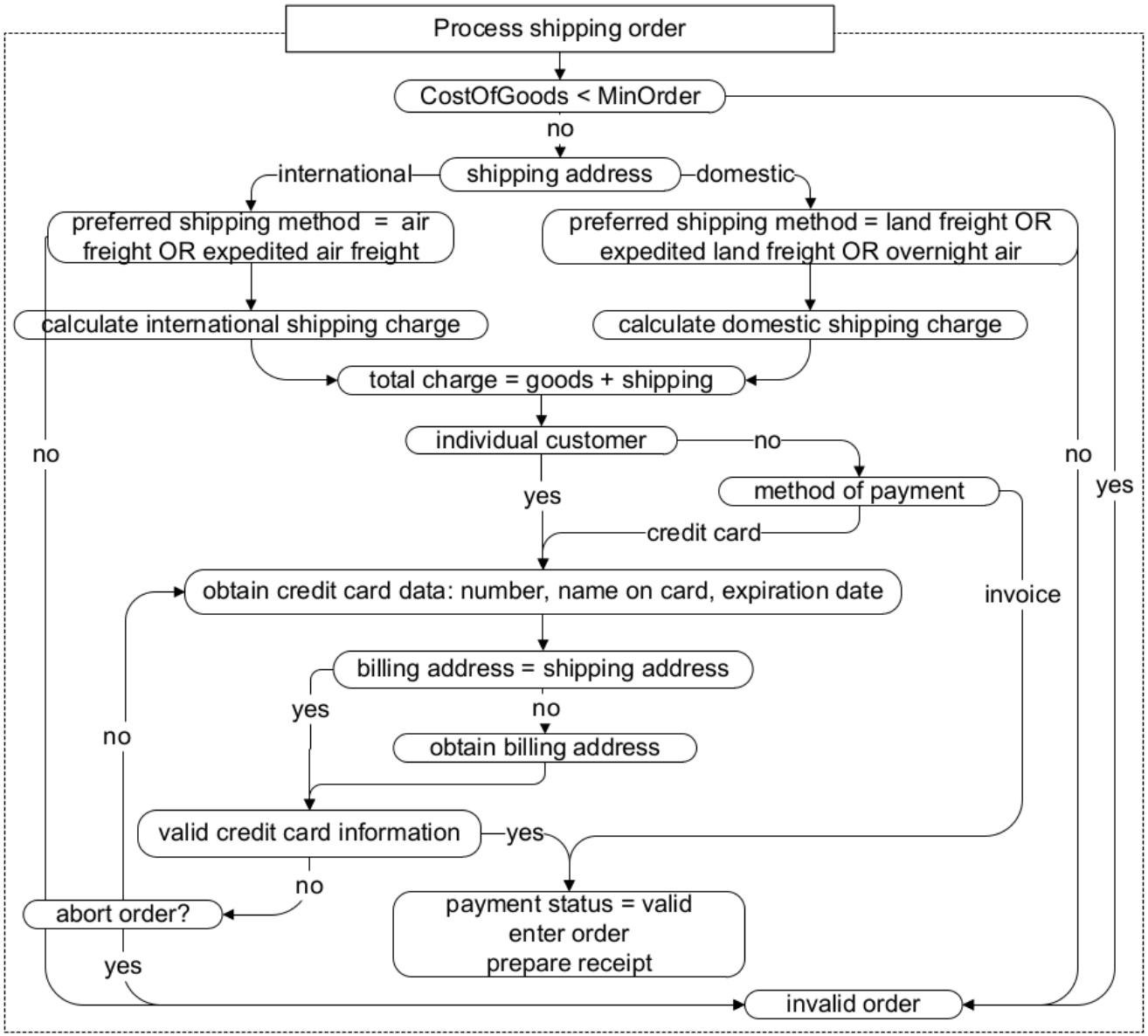
EduAc	T	T	F	F	F	F	F	F	F	F	F	F	F
BusAc	F	F	F	F	F	F	F	F	T	T	T	T	T
CP > CT1	T	T	F	F	T	T	T	T	F	F	T	T	F
YP > YT1	F	-	F	-	-	F	T	T	F	F	F	F	T
CP > CT2	F	F	F	F	F	F	T	T	F	F	F	F	F
YP > YT2	-	-	-	-	-	-	-	-	-	-	-	-	F
SP > Sc	F	T	F	T	F	T	-	-	F	T	F	-	F
SP > T1	F	T	F	T	F	T	F	T	F	T	F	T	F
SP > T2	F	-	F	-	F	-	F	T	F	-	F	-	F
Out	Edu	SP	ND	SP	T1	SP	T2	SP	ND	SP	T1	SP	T1

EduAc	F	F	F	F	F	T	T	T	T	F	-
BusAc	T	T	T	T	T	F	F	F	F	F	F
CP > CT1	T	T	T	F	F	F	F	T	-	-	F
YP > YT1	T	F	F	T	T	T	-	-	-	T	T
CP > CT2	F	T	T	F	F	F	F	T	T	F	F
YP > YT2	F	-	-	T	T	F	-	-	-	T	F
SP > Sc	T	-	T	-	T	F	T	-	-	-	-
SP > T1	T	F	T	F	T	F	-	-	T	T	T
SP > T2	T	F	T	F	T	F	F	F	T	T	-
Out	SP	T2	SP	T2	SP	Edu	SP	Edu	SP	SP	SP

Control (or Data) Flow Graphs

- Sometimes it could be possible to derive a control flow graph from specifications
  - often with coarse granularity, e.g., nodes are single computations or computation steps
  - example: interactions with a database
  - as opposed to before, where CFG is extracted from code
- The statement adequacy criterion becomes node adequacy criterion
- The branch adequacy criterion does not change name
- Other criteria seen for code control flow graphs may be applied as well

Example


Process shipping order
CostOfGoods < MinOrder
no
yes
shipping address
international
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight OR expedited land freight OR overnight air
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
yes
method of payment
credit card
invoice
obtain credit card data: number, name on card, expiration date
billing address = shipping address
yes
no
obtain billing address
valid credit card information
yes
no
abort order?
yes
no
payment status = valid
enter order
prepare receipt
invalid order

Example

T-node

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Air	Ind	CC	–	No (abort)

Abbreviations:

Too small	CostOfGoods < MinOrder ?
Ship where	Shipping address, Int = international, Dom = domestic
Ship how	Air = air freight, Land = land freight
Cust type	Bus = business, Edu = educational, Ind = individual
Pay method	CC = credit card, Inv = invoice
Same addr	Billing address = shipping address ?
CC Valid	Credit card information passes validity check?

Example

T-branch

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	–	–	–	–
TC-3	Yes	–	–	–	–	–	–
TC-4	No	Dom	Air	–	–	–	–
TC-5	No	Int	Land	–	–	–	–
TC-6	No	–	–	Edu	Inv	–	–
TC-7	No	–	–	–	CC	Yes	–
TC-8	No	–	–	–	CC	–	No (abort)
TC-9	No	–	–	–	CC	–	No (no abort)

Abbreviations:

Too small	CostOfGoods $<$ MinOrder ?
Ship where	Shipping address, Int = international, Dom = domestic
Ship how	Air = air freight, Land = land freight
Cust type	Bus = business, Edu = educational, Ind = individual
Pay method	CC = credit card, Inv = invoice
Same addr	Billing address = shipping address ?
CC Valid	Credit card information passes validity check?

Grammars

- Sometimes it could be possible to derive a grammar from specifications
  - regular expressions or annotated context-free grammars (in BNF, Backus-Naur Form)
  - it could be already present as such: e.g., to describe a search pattern
  - XML schema may be easily translated into BNF
- Very good to represent inputs of varying and unbounded size, with recursive structures
- Want a test case? simply generate a string from the grammar!
- Similar to a walk in a graph: how to choose from many possible grammar productions?

Example

$$\langle search \rangle ::= \langle search \rangle \ \langle binop \rangle \ \langle term \rangle \mid \boxed{\texttt{not}} \ \langle search \rangle \mid \langle term \rangle$$

$$\langle binop \rangle ::= \boxed{\texttt{and}} \mid \boxed{\texttt{or}}$$

$$\langle term \rangle ::= \langle regexp \rangle \mid \boxed{(} \ \langle search \rangle \ \boxed{)}$$

$$\langle regexp \rangle ::= Char \ \langle regexp \rangle \mid Char \mid \boxed{\{} \ \langle choices \rangle \ \boxed{\}} \mid \boxed{*}$$

$$\langle choices \rangle ::= \langle regexp \rangle \mid \langle regexp \rangle \ \boxed{,} \ \langle choices \rangle$$

Grammars

- Productions may be different and guide the choice
  - want many short test cases? choose production with many non-terminals first
  - want few long test cases? choose production with few non-terminals first
  - of course, there are intermediate cases
- Production adequacy criterion: each production must be exercised at least once in generating the test case
  - of course, $C_P = \frac{\#\text{execd productions}}{\#\text{all productions}}$

Grammars

- Boundary condition grammar-based adequacy criterion: each production must be exercised at least m and at most M times in generating the test case
  - productions must be labeled by bounds
- Probabilistic grammar-based adequacy criterion: see paper

Testing for Objected-Oriented Software

- Conceptually, same as procedural software
  1. generate functional tests from specification
  2. add selected structural test cases
  3. work from unit testing and small-scale integration testing toward larger integration
  4. system testing
- However, some techniques are tailored for OO software, to tackle OO software peculiarities
  - short methods (e.g., getters and setters), no statement coverage etc.
  - sequences of same-class methods calls are important
  - polymorphism, dynamic binding, generics, overloading...

Objected-Oriented Software Characteristics

State-dependent behaviour testing techniques based on control structure only may be ineffective

Encapsulation testing oracles may need to access class private fields

Inheritance must distinguish between:

- new and overridden inherited methods
- methods that require new test cases
- ancestor methods that can be tested by reexecuting existing test cases
- methods that do not need to be retested

Objected-Oriented Software Characteristics

Polymorphism and dynamic binding testing must exercise different bindings of the same method

- polymorphism: calls to superclass methods bound to subclass methods
- dynamic binding: not only with superclasses

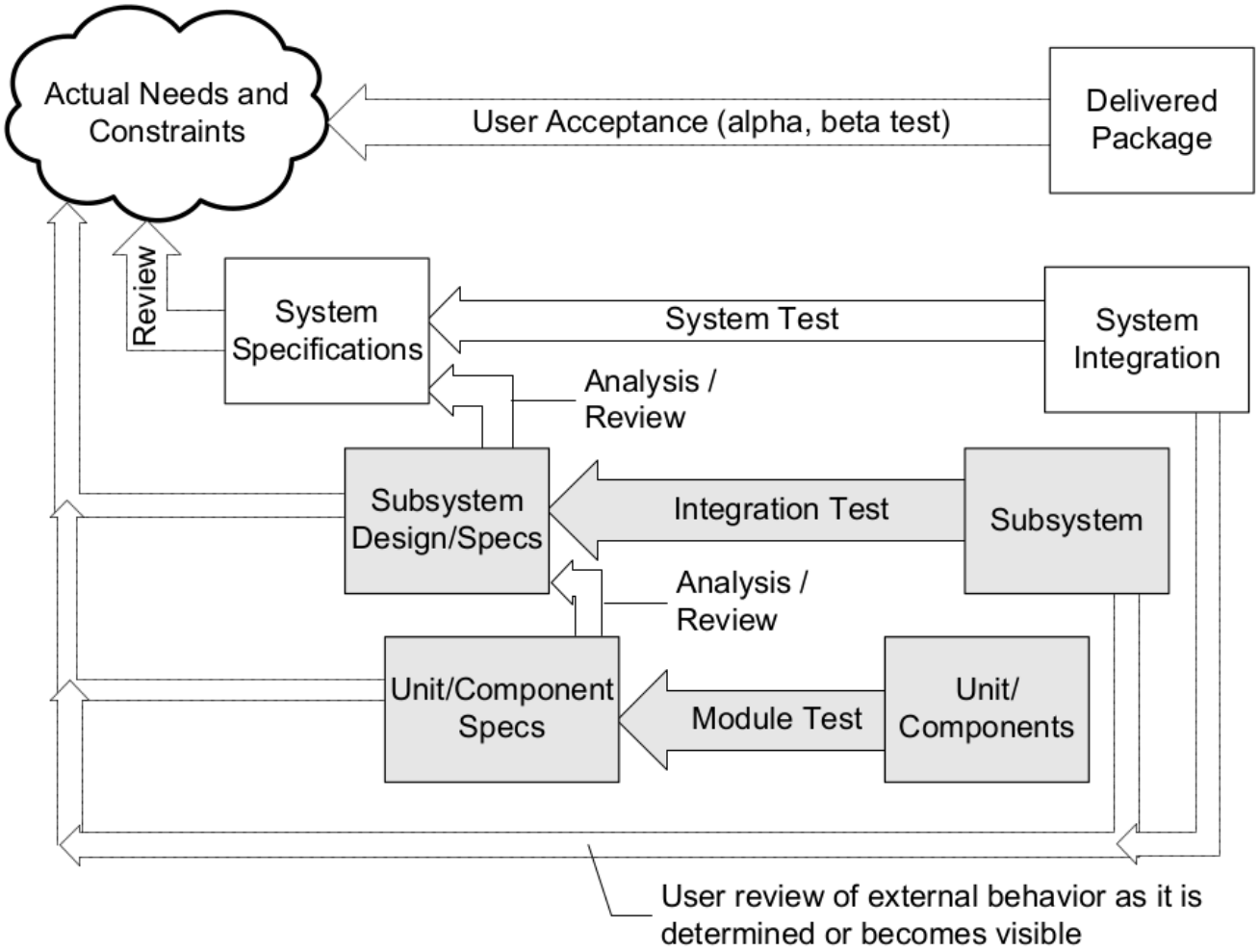
Abstract classes need to be somehow tested

- also without knwoledge of actual implementation
- important for interfaces

Exception handling test both exceptional as well as normal control flow

Concurrency failures may depend on scheduler, which is not under testing control

Impact of Objected Oriented Tailored Techniques



A&T Activities that require specific approaches for OO SW

Objected-Oriented Software Testing

Mainly 3 stages, from single class to system integration

- intraclass: testing each class in isolation
  - also called unit testing
- interclass: testing class integration
  - also called integration testing
- system and acceptance: apply standard functional and acceptance testing techniques to larger components and the whole system
  - independent of design

Objected-Oriented Software Testing Stages: Intraclass

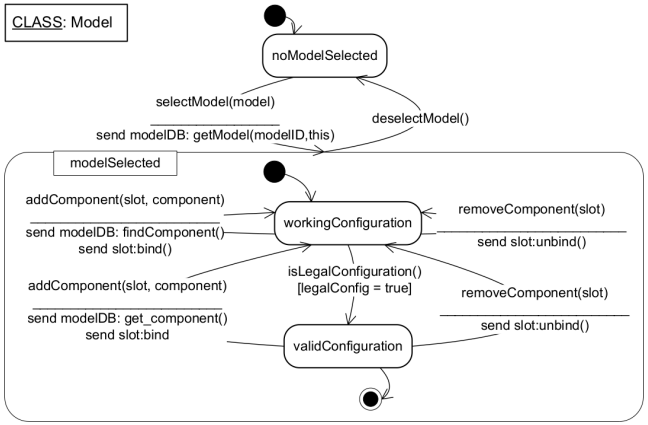
- For an abstract class, derive a set of instantations
  - if not available, do this for testing only
- Derive test cases for constructors and inherited (or overriden) methods
  - if the superclass has already been tested, determine which methods need not to be retested
  - for methods to be retested, determine if the same test cases of the superclass may be reused
- Derive test cases basing on a state machine model of the class behaviour
  - state and transition coverage for testing

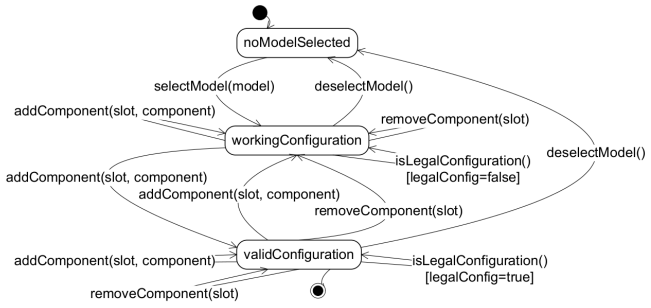
Objected-Oriented Software Testing Stages: Intraclass

- Generate additional test cases considering structural testing techniques
- Derive test cases for exception handling
  - both exceptions which are only thrown and exceptions which are caught and handled
- Derive test cases for polymorphic methods
  - instantiate a superclass in all possible ways

Intraclass Testing: State Machines

- Rather than a generic state machine, we may have a UML statechart
- UML statecharts are somewhat more complicated than standard state machines
  - superstates: to be replaced with a flattening
  - considering all ingoing and outgoing transitions...

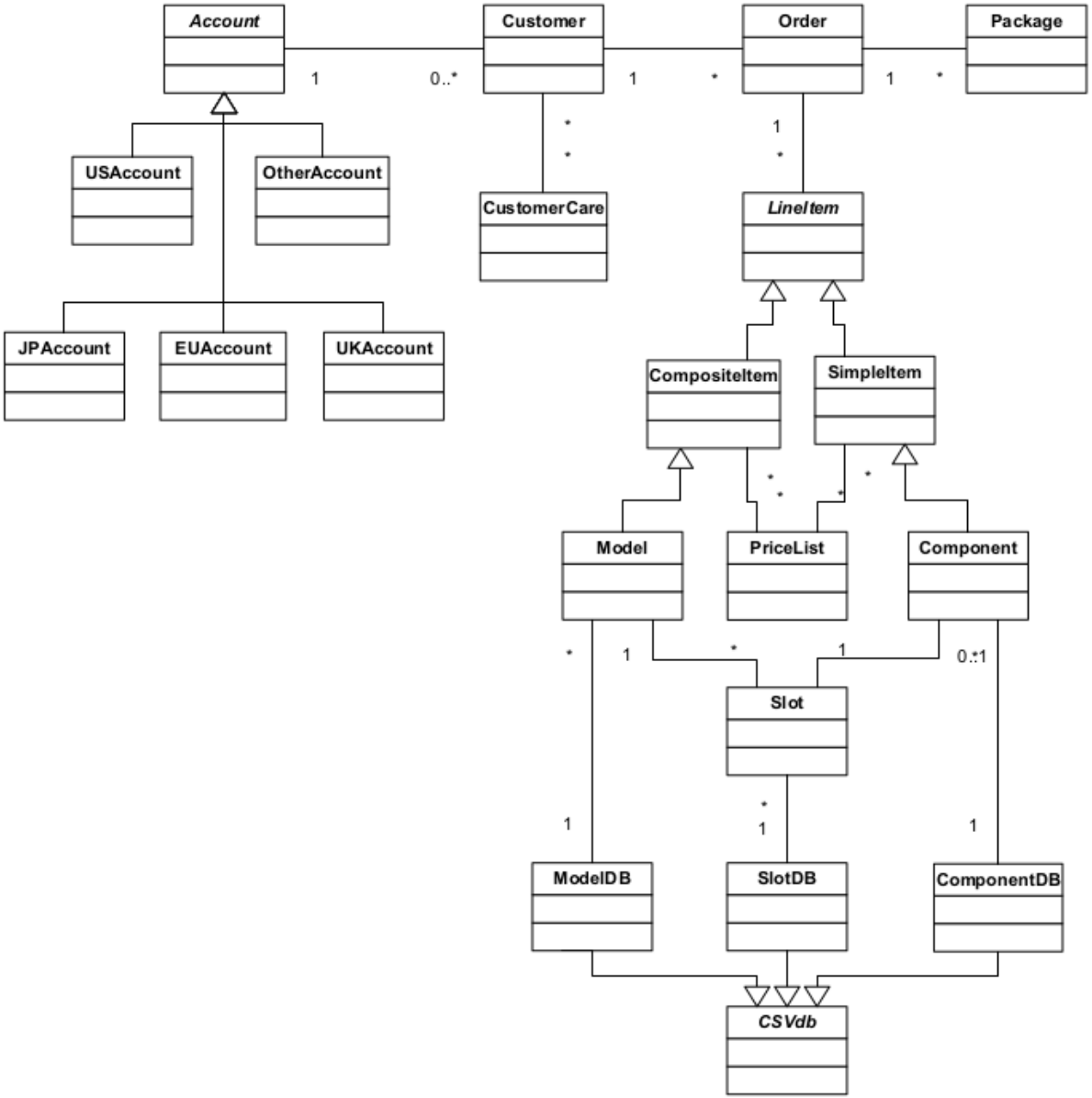




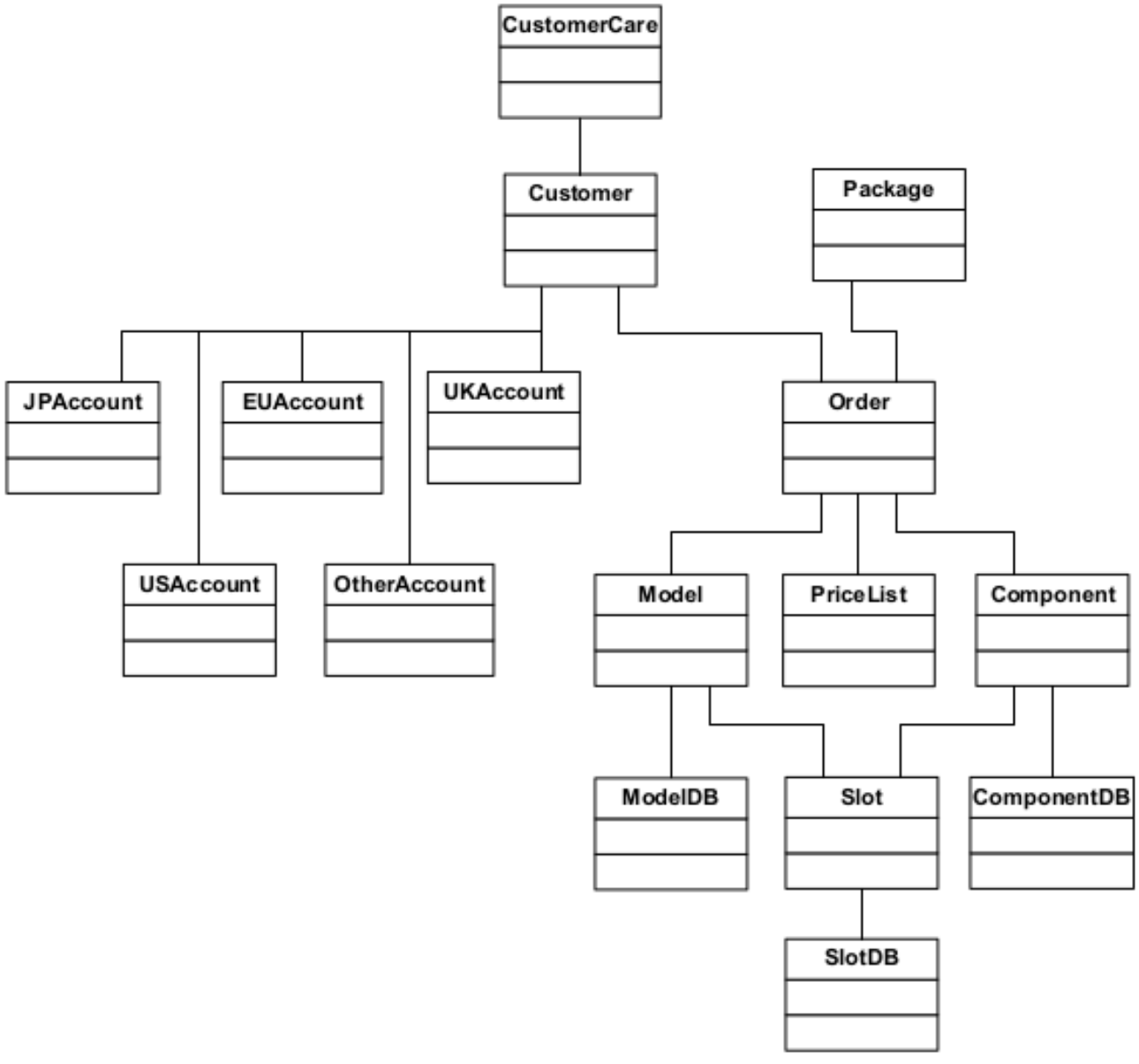
Interclass Testing

- Two main workhorses: use/include relation and sequence/collaboration diagrams
- Use/include is very simple to derive from UML Class Diagrams
  - typically drawn with simple vertical lines: the class on the top depends on the one on the bottom
- Once you have it, simply start testing from the bottom
  - classes which does not depend on any other else with classes which only depend on those
  - difficult to generalize, experience helps
  - need to select a subset of interactions among the possible combinations of method calls and class states

From UML...


Account
Customer
Order
Package
USAccount
OtherAccount
CustomerCare
LineItem
JPAccount
EUAccount
UKAccount
CompositeItem
SimpleItem
Model
PriceList
Component
Slot
ModelDB
SlotDB
ComponentDB
CSVdb
1
0..*
*
0..1

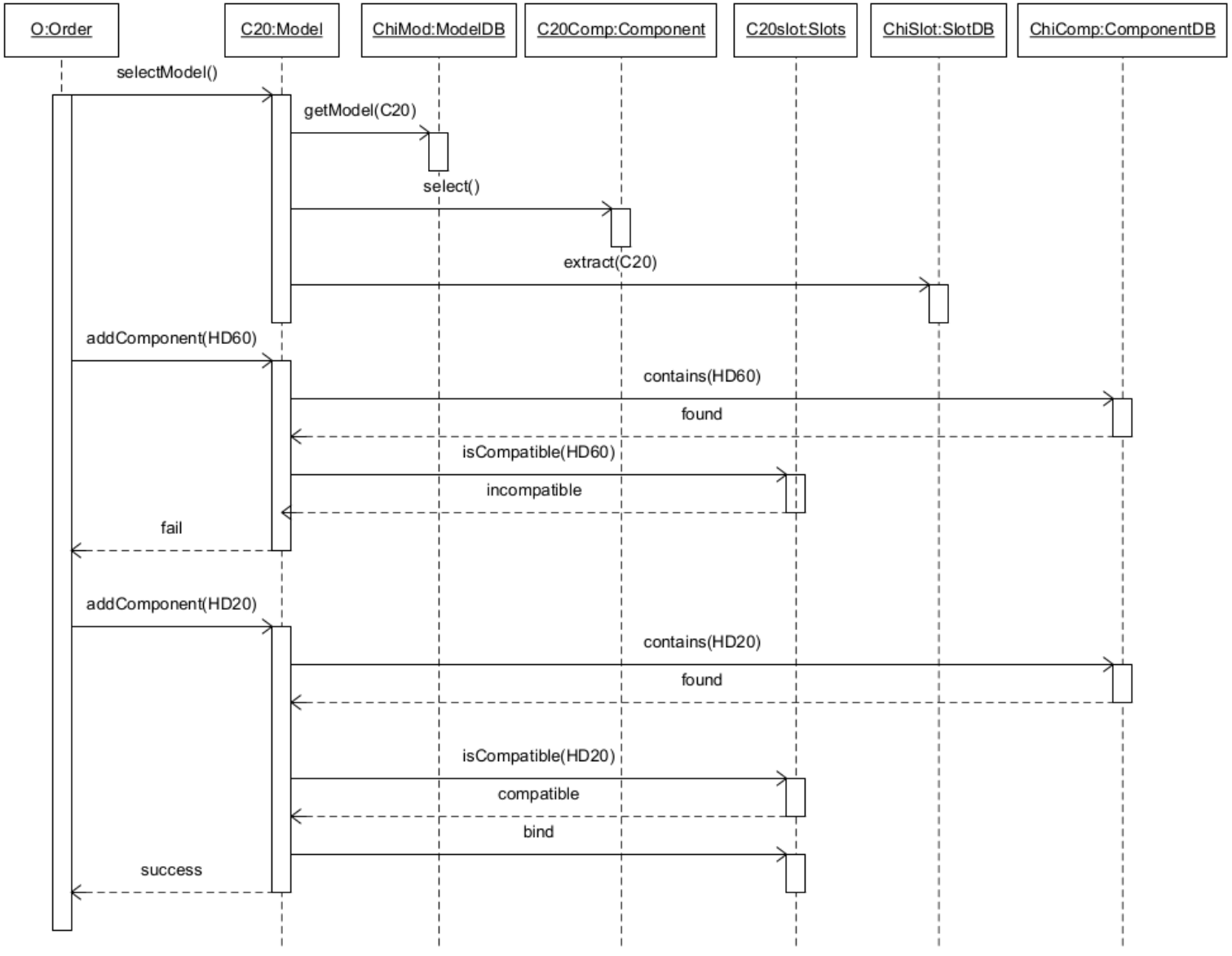
... to Use/Include Diagram



Interclass Testing

- Sequence diagrams may be used to design test which cover all possible exchanges
- Furthermore, some interaction in the sequence diagram may be replaced by another, to check if errors are handled correctly
- State diagrams should cover all possible behaviours
- Instead, sequence diagrams are a selection made by designers
- Thus, they are very valuable for testing

Example



Functional and Structural Testing for OO Software

- For single methods, the same function (and structural) techniques we used for procedural programs are available
- With one major difference: *sequences* of methods calls are typically more important
  - testing a getter without considering the corresponding setter is useless
- Another way of seeing it: test the methods with different values for the state
  - state = member variables
  - one integer variable is enough to have an impractible number of tests
  - choose some meaningful values
  - if specifications tells which ok, otherwise random

Functional and Structural Testing for OO Software

- Another way of seeing it: definition-use pairs
  - but now, setters must be taken into account for definitions
  - getters may be hidden inside other methods
  - an intraclass CFG is needed
- To exercise a DU pair we need a sequence of method invocations that:
  - starts with a constructor
  - pass through the definition
  - end with the use without going through any other definition
- Given this, the all DU pairs adequacy criterion is defined as above

Interclass Structural Testing

- Classification of methods as:
  - inspectors: only read the class state
  - modifiers: only write the class state
  - inspectors/modifiers: both
  - getters are inspectors, setters are modifiers
  - "class state": at least one variable in the class state
- By going bottom-up in the whole class dependencies:
  - invocations of modifier methods and inspector/modifiers of leaf classes are considered as definitions
  - invocations of inspectors and inspector/modifiers in other classes are treated as uses
- Proceed as in definition-use pairs

Oracles, Drivers and Stubs

- Stubs only needed if we want to test a part of the program when some other related part is not ready
- Drivers actually launch the experiments
  - not different from procedural testing
- Oracles are more difficult than procedural testing due to incapsulation
- Technique one: allow oracle to read private members
  - not a good idea: tested and delivered software would be different!
  - exploit language features: friend classes in C++
  - or package visibility in Java (also putting oracles in the same package)

Techniques for Building Oracles

- Technique two: consider equivalence between objects
  - especially useful for arrays and similar
  - an array is similar to a linked list: important is that they have the same values...
  - the oracle uses the equivalent data structure, if the original one is not available bacause private
- One sequence of method invocations is equivalent to another if the two sequences lead to the same object state
- This does not necessarily mean that their concrete representation is bit-for-bit equal

Techniques for Polymorphism and Dynamic Binding

- If the possible morphs or binding are just a few, simple flatten them
- If not, data flow analysis may be used
  - identify potential interactions: where a variable is modified w.r.t. where is used
- This does not necessarily mean that their concrete representation is bit-for-bit equal

Techniques for Inheritance

- Inheritance cannot introduce errors per se
  - though it is the base for polymorphism
- However, it can be exploited to understand when tests can be reused
- To this aim, distinguish methods of a subclass in:
  - new: not present in the superclass
    - present = same name & same parameters (implies same return)
  - recursive: present in the superclass and left unchanged
  - recursive: present in the superclass and body changed
- Also distinguish if the method was abstract in the superclass
  - abstract new: abstract in the subclass

Techniques for Inheritance

- Thus, recursive methods need not to be retested
  - abstract recursive only tested with stubs, as an implementation is lacking
- (Abstract) redefined and new methods always need to be retested
- To keep track of what to retest, a testing history table may be used
- When a new subclass is considered, the table is scanned to understand which methods needs retesting
- Note that abstract methods cannot be really tested, so the corresponding tests cannot be executed

Techniques for Inheritance

- "Local" methods are those which only call other methods of the same class
- Testing history tables are organized as follows:
  - rows are methods
  - columns are of 4 types:
    - intraclass functional
    - intraclass structural (not for abstract)
    - interclass functional (not for local)
    - interclass structural (not for local or abstract)
  - entries are the corresponding test set, plus a flag
    - executable/not executable

Techniques for Parameterized Types

- "Generics" in Java, "templates" in C++
  - typical example: an array where the type of each entry may be decided when instantiating the object
- Only class instantiations can be tested
  - as for abstract methods
  - cannot know in advance which type will be used
- Testing is split in two parts:
  - showing that some instantiation is correct
  - showing that all permitted instantiations behave identically

Techniques for Exceptions

- Exceptions are by themselves an help to testing
  - in procedural programming, overlooking error codes returned by functions ofter occurs
  - exceptions handling mitigates this problem, as an exception is certain to interrupt normal control flow
  - cost: implict control flows are added
- Building a CFG with exceptions is impractible
- Testing technique one: to test custom code for handling some exception, a stub that raises that exception must be built
- Testing technique two: exercise each point at which an exception is explicitly thrown, and each custom handler, but not necessarily all their combinations.

Tests Execution

- Executing tests is not always straightforward
- General goal: once we have test cases, we should perform the last steps as automatically as possible
  - not always possible, e.g., if the result is visually checking a Web page
  - in other cases, once organized, it is not different from compiling
- Mastering the following techniques is important:
  - run-time support for generating and managing test data
  - creating scaffolding for test execution
  - automatically distinguishing between correct and incorrect results
- Run-time support for testing enables frequent "easy" reexecution of a test suite

From Test Case Specifications to Test Cases

- That is: we want the raw inputs
- Keeping specification and generation separate aids in reducing impact of small changes in the software development
- Some test case specifications are extremely simple to be completed
  - e.g., those coming from partition/category method
- Others may require some other effort
  - e.g. "a sorted sequence, length greater than 2, with items in ascending order with no duplicates"
- Two or more test cases specifications may be dependent on each other

From Test Case Specifications to Test Cases

- A very general case specifications may be difficult to be implemented
  - e.g., "traverse these transition in this program state machine" $\rightarrow$ find the data which cause that transitions
  - model checking may be used!
  - say the transitions are impossible and collect the counterexample...

Tools Which Generate Test Cases

- Though it is indecidable, some tools are actually able to output test cases for code
  - typically, for C code
  - typically, incomplete tools, but better than nothing
- CBMC: given some *small* C code, generates test cases for all types of coverage
  - free, for any platform
- Reactis: given some *small* C code, generates test cases for all types of coverage
  - commercial, for Windows
- Unit testing only, with some support for call coverage

Scaffolding

- Literally, the temporary structures erected around a building during construction or maintenance
- The problem is that we want to test from early stages: very few components will be available
- Thus, we need to create code to only for the purpose of testing:

  drivers for calling programs

  stubs for functions called by the program

  test harnesses for parts of the deployment environment
  - external signals generation, different execution platforms/hardware, ...

  other program instrumentation and support for recording and managing test execution

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Scaffolding

- Could require an high cost, up to half of the code developed for the entire project
  - of course, it depends on the application domain and on the project itself
- Cost lowered by paying attention in early stages
  - well-designed build plan
  - provide necessary functionality in a rational order

Scaffolding

- Even if stubs and drivers are not necessary, instrumentation is (nearly) always needed
  - controllability: launching the test in a deterministic way
    - may be an issue for thread-based programs
  - observability: if there is a failure, we want to know it
    - or if the result of a computation is not correct

Scaffolding Techniques

- Writing a driver for a single test case is usually easy
  - exercise a given sequence of calls
- If we have to write a driver for each single test case, it would be impractible
- Better write some generic driver, if the project budget allows to
  - kind of: take the description of the test case and execute it
- Typical scaffolding support:
  - execute (generic) test case
  - log test execution
  - check results

Scaffolding Techniques

- Also scaffolding: how to replace (unavailable) portions of the program to be tested? We need stubs
- Easiest form of stub: *mock*
  - replace a function with another function taking the same parameters and outputting a fixed value
  - "output" in a broad sense: also print a string or set global variables
  - in many cases, mock can be generated automatically from source code
- Also scaffolding: test harnesses like a network traffic generator to test some distributed system
- Deciding to create a new software for scaffolding or not depends budget

Test Oracles

- Again part of scaffolding: how to check test results?
  - human intervention to be avoided whenever possible: expensive and unreliable
- A *test oracle* is something distinguishing correct from uncorrect test results
  - automated test oracles: it is a software
- Partial oracle: one with false positives
  - sometimes good because cost-effective, especially for early testing
- Oracle with false negatives: to be avoided
  - requires manual inspection to understand it is a true or false negative

Test Oracles Techniques

- Comparison-based oracles
  - test cases are (input, output) pairs, so oracle simply check if result is equal to expected output
  - may be not bit-to-bit equal and still be ok
  - this is typically the case for test harness
- How to have correct (input, output) pairs?
  - could seem a self-referencing solution...
  - depends on the application and the problem
- Solution 1: create an output and produce a corresponding input
  - sorting: create a sorted list and permute it...
- Solution 2: use some other software
  - may be already available, but not usable in production code
    - e.g., due to intellectual rights, or because it is too slow
  - may be written by test engineers
  - the important thing is that it is independent by the program to be tested
    - not necessarily better

Test Oracles Techniques: Capture and Replay

- Solution 3: use humans, but only once
- Especially ok for visual responses of graphical interfaces
- The human judges the output, and its evaluation is recorded together with the input
- Starting from that point, any other re-execution of the test remembers the human evaluation
- Not always simple or cost-effective: small differences in a graphical interface should be ok but may trigger a false negative

Test Oracles Techniques

- Oracles need not to be comparison-based: we may implement some software able to check the output
  - using specifications, of course
- Checking if an output is correct is often easier than producing it
  - e.g., routing problem...
- Partial oracles: drop optimality
  - e.g., in optimal routing problem, we only check if the output route is correct, not if it is optimal
  - often combined with comparison-based oracle:
    - cheap, partial oracle for large test suite
    - heavy test suite with precomputed outputs for a subset of the suite

Besides Testing: Inspection

- Have a human inspect the code
  - of course, the code must be inspectable
  - something could be done also automatically, see, e.g., lint and purify
- Not too much, it is boring
  - two hours a day
  - valuable for juniors, they see production code
  - reinspection is as hard as the first one
- Organize the work, perform the work, speak with developers
- Perform the work: typically with checklists
- Pair programming in Agile method: inspection included in implementation phase

Example

Java Checklist: Level 1 inspection (single-pass read-through, context independent)

FEATURES (where to look and how to check): Item (what to check)			
FILE HEADER: Are the following items included and consistent?	yes	no	comments
Author and current maintainer identity			
Cross-reference to design entity			
Overview of package structure, if the class is the principal entry point of a package			
FILE FOOTER: Does it include the following items?	yes	no	comments
Revision log to minimum of 1 year or at least to most recent point release, whichever is longer			
IMPORT SECTION: Are the following requirements satisfied?	yes	no	comments
Brief comment on each import with the exception of standard set: java.io.*, java.util.*			
Each imported package corresponds to a dependence in the design documentation			
CLASS DECLARATION: Are the following requirements satisfied?	yes	no	comments
The visibility marker matches the design document			
The constructor is explicit (if the class is not *static*)			
The visibility of the class is consistent with the design document			
CLASS DECLARATION JAVADOC: Does the Javadoc header include:	yes	no	comments
One sentence summary of class functionality			
Guaranteed invariants (for data structure classes)			
Usage instructions			
CLASS: Are names compliant with the following rules?	yes	no	comments
Class or interface: CapitalizedWithEachInternalWordCapitalized Special case: If class and interface have same base name, distinguish as ClassNameIfc and ClassNameImpl			
Exception: ClassNameEndsWithException			
Constants (final): ALL_CAPS_WITH_UNDERSCORES			
Field name: capsAfterFirstWord. name must be meaningful outside of context			
IDIOMATIC METHODS: Are names compliant with the following rules?	yes	no	comments
Method name: capsAfterFirstWord Local variables: capsAfterFirstWord. Name may be short (e.g., i for an integer) if scope of declaration and use is less than 30 lines.			
Factory method for X: newX			
Converter to X: toX			
Getter for attribute x: getX();			
Setter for attribute x: void setX			

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Example

Java Checklist: Level 2 inspection (comprehensive review in context)

FEATURES (where to look and how to check):
Item (what to check)

DATA STRUCTURE CLASSES: Are the following requirements satisfied?	yes	no	comments
The class keeps a design secret			
The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface			
Methods are correctly classified as constructors, modifiers, and observers			
There is an abstract model for understanding behavior			
The structural invariants are documented			
FUNCTIONAL (STATELESS) CLASSES: Are the following requirements satisfied?	yes	no	comments
The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface			
METHODS: Are the following requirements satisfied?	yes	no	comments
The method semantics are consistent with similarly named methods. For example, a "put" method should be semantically consistent with "put" methods in standard data structure libraries			
Usage examples are provided for nontrivial methods			
FIELDS: Are the following requirements satisfied?	yes	no	comments
The field is necessary (cannot be a method-local variable)			
Visibility is protected or private, or there is an adequate and documented rationale for public access			
Comment describes the purpose and interpretation of the field			
Any constraints or invariants are documented in either field or class comment header			
DESIGN DECISIONS: Are the following requirements satisfied?	yes	no	comments
Each design decision is hidden in one class or a minimum number of closely related and co-located classes			
Classes encapsulating a design decision do not unnecessarily depend on other design decisions			
Adequate usage examples are provided, particularly of idiomatic sequences of method calls			
Design patterns are used and referenced where appropriate			
If a pattern is referenced: The code corresponds to the documented pattern			

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica