Taxonomies of Testing Techniques

• Conventional taxonomies
• Based on operational characteristics
  - Static vs. Dynamic
• For example,

An Example Taxonomy

• Two dimensions
  - Types of documents
  - Static/dynamic

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static analysis of design documents</td>
<td></td>
<td>Design-based testing</td>
</tr>
<tr>
<td>Informal checklists</td>
<td>Functional testing</td>
<td></td>
</tr>
<tr>
<td>Formal modeling</td>
<td>Testing by classes of input data</td>
<td></td>
</tr>
<tr>
<td>Structural testing</td>
<td>Testing by classes of output data</td>
<td></td>
</tr>
<tr>
<td>General information</td>
<td>Structural testing</td>
<td></td>
</tr>
<tr>
<td>Static error analysis</td>
<td>Expression testing</td>
<td></td>
</tr>
<tr>
<td>Symbolic execution</td>
<td>Data-flow testing</td>
<td></td>
</tr>
</tbody>
</table>
Why Taxonomy?

• Well suited to planning a series of validation activities
  - Identifies the type of documents required
  - Allows a manager to identify where a technique may fit into the product’s lifecycle

• Helps cost estimation
  - Identify resources/needs/documents
  - Static analysis is computationally cheaper than dynamic
    • May be misleading

Observations

• No single testing technique is capable of finding all faults

• Every technique involves a tradeoff
  - Between accuracy and completeness on one hand
  - And tractability on the other

• Various software validation schemes have been defined
  - Combine several techniques by applying them in sequence

• Limited success
**Drawback of Operational Characterization**

- Limited success because of static/dynamic analysis orientation
- Predisposes one to view each technique in isolation
- Obscures the important issues of technique interaction
- Dimensions of tradeoff are orthogonal to the issue of program execution
- Conventional taxonomies do not adequately address tradeoffs between accuracy and computational effort/cost

**Practical Testing**

- Sampling subset of program behaviors
  - Execute a few program paths
- Folding states together
  - Abstracting away details to create a model
    - Control-flow model
    - Data-flow model

- Discussion
Sampling

- Explore few states
  - Statement
  - Branch
  - Path
    - All feasible paths
  - Exhaustive
    - All inputs

- Threshold of tractability!
- Threshold of decidability!

Inaccuracy

- Can we fail to reject an incorrect program?
  - Optimistic inaccuracy
- Can we fail to accept a correct program?
  - Pessimistic inaccuracy
- For practical techniques
  - Admit at least one inaccuracy
- Conservative techniques
  - Pessimistic inaccuracy
  - But no optimistic inaccuracy
Proving Correctness

- Impossible in general
- Construct proofs for some programs by abstracting away details
  - E.g., Flow-graphs, “virtual coarsening”
  - Data-flow
  - Static type checking?
  - If successful, then program is (may be) correct
  - Failure to find a proof?
    - Program “may” or “may not” be incorrect
    - Pessimistic inaccuracy

Folding

- Abstracting away details
  - Structural properties
  - Dataflow analysis
  - Reachability analysis
  - Infallible proof finder

Merely Hopeless

Truly Impossible
Combining Folding & Sampling

- First fold states to get a “smaller” state-space
- The sample a part of this state-space
- For example, create a Petri net model and execute it

Summary
Symbolic Evaluation

- **Symbolic execution**
  - **Program flow-graph**
    - Nodes for each program statement
    - "If" statement node has two out-edges
    - "While" statement node has two out-edges
  - **Execution representation**
    - Token represents a thread of control
    - Path expression
      - Program variables <-> symbolic values
    - Path conditions
      - Predicates describing the conditions

Symbolic Execution

- **Initialize execution**
  - Token on edge leading to first node
  - Path condition is TRUE
  - Path expression: associate each program variable with a unique symbol

- **Execute**
  - Advance token from in-edge to out-edge
    - Assignment statement: modify path expression
    - "If" and "while" statements: add a term to the path condition
**Symbolic Execution**

- **Program state**
  - Path expression & path condition

- **State space**
  - For a program without loops, what does the state space look like?
    - A tree: can (in principle) be exhaustively explored to check for problems
  - With loops, the state space is infinite. Only some paths may be checked, i.e., Explore a sample of the state space

- **Symbolic testing**
  - Start from the initial state to a terminal state

**Symbolic Execution**

- **Can we fail to reject an incorrect program?**
  - How about unexplored paths? What if a fault lies on one of them!!
  - Optimistic inaccuracy