Input Validation Testing:
A Requirements-Driven, System level,
Early Lifecycle Technique

Jane Hayes                               Jeff Offutt
SAIC                           George Mason University
jane.e.hayes@cpmx.saic.com      ofut@gmu.edu

Support from U.S. National Science Foundation and U.S. Navy
is greatly appreciated.
Motivation

- Many older generation language applications exist
- Many of these applications depend on user keyboard input
- Keyboard input highly error prone
- Interfaces are acknowledged problem areas
Motivation

- IVT can help build test suites for older as well as modern language applications

- IVT can improve analysis and testing of:
  - Transaction control languages
  - Communications protocols
  - User/Operator commands
  - Inter- and Intra-system interfaces
Test Specifications & Requirements

■ Software
  – Spec/Req -- What the software does
  – Design -- How the software does it

■ Tests
  – Requirements: Specific things that must be satisfied or covered (statements, branches, etc.)
  – Criterion: A set of rules that imposes test requirements
  – Coverage: Extent to which a criterion is satisfied
Software Testing Levels

- **Unit and Module Testing**: Testing individual procedures and groups of related procedures.

- **Integration Testing**: Testing for interface problems and incompatibilities between objects.

- **System Testing**: Testing a complete system from an external perspective.
Input Validation Definitions

- **Command Language Interface**
  - Language having complete, finite set of actions
  - Entered textually via keyboard
  - Used to control execution of software system

- **Syntax-directed Software**
  - Software system with command language interface

- **Input Validation Testing**
  - Choosing test data for specific input tolerance faults

- **Test Obligation**
  - Ensure that a static defect is not in the system
Input Validation Testing (IVT)

- System testing is primarily about finding faults in the system structure.

- IVT is about finding faults in the input handling.

- IVT can yield good system level tests.
Four IVT Steps

1) Specifying Input Format

2) Analysis of User Command Specification

3) Generating Valid Test Cases

4) Generating Invalid Test Cases
Specifying Input Format

IVT expects a minimum of three fields for data elements:

1) Data Element Name

2) Data Element Size

3) Expected/Allowable Values
Static Analysis: Requirements Quality Criteria

1) **Completeness**
   - Ensure data values for every column and row of input table
   - Static analysis of input specification table

2) **Consistency**
   - Analyze command language tables
   - Analyze input / output tables

3) **Correctness**
   - Not addressed in this research
Static Analysis:
Other Checks on Input Format

1) Potential Catenation
   Two data elements of the same type are adjacent

2) Ambiguous Grammar
   If a data element is a type of another element,
   input elements are ambiguous

3) Overloaded Token
   Same expected value for different data elements

In all cases, generate a test obligation.
Dynamic Analysis: Valid Test Cases

■ Based on Input Syntax Graph
  – Nodes are data elements
  – Edges represent ordering of data elements

■ Apply All-edges Coverage Criterion

■ Loop Heuristics
  – 0 times
  – 1 times
  – N times
  – N+1 times

■ Automatic Generation of Test Values
Dynamic Analysis: Invalid Test Cases

Two Sources:

1) Error Condition Rule Base
   - Top, intermediate, field-level syntax errors
   - Delimiter errors
   - Violation of expected values

2) Test Obligation Database
   - Overloaded token/ambiguous grammar
   - Catenation
MICASA:
Method for Input Cases and Static Analysis

- Interface Spec Tables
- Import Spec Tables
- Database of Tables
- Database of Tables
- Perform Static Analysis
- Generate Covering Test Cases
- Generate Error Cases
- Warnings/Error Msgs, Test Oblig. DB
- All-Edges Test Cases
- Error Test Cases

Visual C++
Accepts flat files and MS Word

ICSEA, 12/98
MICASA (Cont’d)
Validation

Validated MICASA against senior testers

Specifications for large, real-world systems
  1) Three Navy systems
  2) One commercial system
  3) One FBI system

Three part experiment:
  1) Statically analyzing specifications for defects
  2) Generating adequate test cases for specifications
  3) Executing test cases
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>MICASA</th>
<th>Testers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Spec. Defects Found</td>
<td>524</td>
<td>21</td>
</tr>
<tr>
<td>Total Spec. Defects Found</td>
<td>524</td>
<td>106</td>
</tr>
<tr>
<td>Number of Test Cases</td>
<td>48</td>
<td>7</td>
</tr>
<tr>
<td>Software Faults Found</td>
<td>20</td>
<td>27</td>
</tr>
<tr>
<td>Defect Detection Rate</td>
<td>7.4</td>
<td>4.6</td>
</tr>
<tr>
<td>Minutes Per Fault Found</td>
<td>8.4</td>
<td>72.2</td>
</tr>
</tbody>
</table>
Experimental Results

- **MICASA outperformed senior testers:**
  - Statically found more total and syntactic defects
  - Statically found defects not found by senior testers
  - Generated test cases with higher syntactic coverage
  - Required less time to develop and execute test cases
  - Generated test cases found defects not found by senior testers

- **Senior testers outperformed MICASA:**
  - Found more defects per test case
Observations / Conclusions

- Poor quality tables
- Overlap
- Early lifecycle analysis can facilitate late lifecycle testing
- Robust interfaces should be specified and designed
- Technique in use
Contributions

- Interface specification defects map to SW failures
- Defects drive effective input validation testing
- System coverage criterion
- General, multi-domain
- Early lifecycle
- Automated
- Implemented and validated
- Currently used in practice
Future Research

- Sequencing of commands
- Handling masks (DD-MM-YYYY)
- Handling automatically entered data elements
- Handling data element dependencies
- Adapting to GUIs
- Investigate individual defects