Code and Functional Coverage Tutorial
May '99

Shmuel Ur
IBM Research Laboratory in Haifa

Explaining the System....
Outline

- Introduction to Coverage
  - Code Coverage
  - Regression
  - Functional Coverage
  - Class Exercise
  - Automatic Coverage Directed Generation
  - Comparing Functional Coverage to Code Coverage
  - Conclusions

What is Coverage?
Why Coverage?

- **Testing is based on samples**
  - Cannot run all possible tests
  - Need to know that all areas of design are tested
  - A test suite is not measured by its size

- **Solution: Coverage Analysis**

Definition of Coverage

Coverage is any metric of completeness with respect to a test selection criterion.

The main ideas behind coverage:

- Systematically create a list of tasks (the testing requirements)
- Check that each task is covered during the testing
Other Definitions

- Coverage Task - a Boolean function on a trace
  - Statement 534 was executed
  - A plane landed on Friday the 13th

- Coverage Model - a cohesive set of coverage tasks
  - Execution of each statement in a program
  - A plane landed every day of the year

- Illegal Task - a coverage task that should not occur

- Task Coverage List (TCL) - a list of all the legal tasks

Coverage Goals

- Measure the "quality" of a set of tests
- Supplement test specifications by pointing to untested areas
- Help create regression suites
- Provide a stopping criteria for unit testing
Risks of Using Coverage

- Using coverage without commitment to using the results
- Generating simple tests to cover specific uncovered tasks
  - The painted wall analogy:

- Some coverage models are ill-suited to deal with common problems
  - Missing code
- Low coverage goals

Automatic Test Generation and Coverage

- Main goal is to improve biasing
- Working with feedback (closed loop)
- Avoiding the risk of specific biasing
What is the Value of Code Coverage?

- Provides a quantitative measurement of the testing effort
- Can assist in directing the tester’s future efforts
- Can demonstrate redundancy in test cases
- Can be used as entry/exit criteria between test phases
- Studies have demonstrated a direct correlation between increased code coverage and defect detection
Types of Code Coverage Models

- **Control flow**
  - Check that the control flow of the program has been fully exercised

- **Data flow**
  - Models that look at the flow of data in, and between, programs

- **Mutation**
  - Models that check directly for common bugs

- **Visual coverage models**

- **Program types - OO, concurrent, distributive**

Control Flow Models

- **Routine (function entry)**
- **Function call**
- **Function return**
- **Statement (block)**
- **Branch**
- **Multi-condition**
- **Loop**
Data Flow Models

- C-Use (Computational use)
- P-Use (Predicate variable use)
- All Uses

Mutation (Relational Operation) Coverage

- Weak Mutation
- Strong Mutation

Examples:

\(<\leq\quad\geq\)
\(>\quad\leq\)
\(&\quad\quad\&\&\)
Hierarchy of Coverage Models

- All-Uses
  - P-Use
  - C-Use
- Decision
- Block
  - Function Return
  - Function Call
- Function Entry

Hardware Models

- State machine coverage - states, transitions
- All values for signals
- Stuck At
What is xSuds (from Bellcore)?

- Principally a code coverage tool suite
- Parses C and C++ source code
- Runs on AIX, Solaris, Linux, WinNT

xSuds Tool Suite Components

- xATAC - a test coverage measurement, management & test creation tool
- xVue - supports software maintenance by visualizing code features
- xSlice - a dynamic program slicing debugger that graphically localizes bugs
- xProf - an execution count-based profiler
- xDiff - displays differences between two files in a more understandable way
- xFind - uses date seeds to locate Y2K sensitivities in code
- xRegress - assists in the selection of a minimum set of tests to cover changed code
How Does xSuds Work?

**ATAC Architecture**

Static Analysis

![Diagram of ATAC Architecture]

---

**xSuds**

**File**  **Tool**  **Options**  **Summary**  **TestCases**  **Update**  **GoBack**

<table>
<thead>
<tr>
<th>File</th>
<th>Summary</th>
<th>TestCases</th>
<th>Update</th>
<th>GoBack</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp1.c</td>
<td>31.4%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp2.c</td>
<td>23.5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main.c</td>
<td>30.3%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc.c</td>
<td>33.8%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quant.c</td>
<td>84.8%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip.c</td>
<td>51.7%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sort.c</td>
<td>54.2%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total**

249 of 617  40.4%

**xATAC**

Coverage: block  Files: 7 of 7  Testcases: 3 of 3

21-22
What is Missing from xSuds?

- Signing on coverage tasks
- Removing covered coverage tasks (to improve performance)
- Coverage models
- Operating systems
- Programming languages

Creating Your Own Code Coverage Tool

- Avoid if possible:
  - It is a lot of work
  - It is bug prone
  - It is slow

- When creating such a tool, consider:
  - Instrumentation
  - Traces
  - Off-line vs. on-line task updating
  - Reducing the size of traces
  - Integrating information
Test Coverage for Visual Programs

Visual elements to be covered:
- Connections
- Events
- Actions
- Attributes

Path Coverage - First View
A TEST!!!
From Cem Kaner "Software negligence and testing coverage"

- A program fails in the field and someone dies. At the trial, the QA Manager takes the stand and the plaintiff’s lawyer states three facts:
  - Failure occurred when the program reached a specific line of code.
  - This line had never been tested, therefore the bug had not been found before the product was released.
  - Had the QA used the coverage monitor and tested to complete coverage, this bug would have been found and the victim would be alive today.

- The plaintiff’s lawyer therefore proved that the company was negligent and that the victim’s family would win the lawsuit.
Outline

- Introduction to Coverage
- Code Coverage
  - Regression
- Functional Coverage
- Class Exercise
- Automatic Coverage Directed Generation
- Comparing Functional Coverage to Code Coverage
- Conclusions

Regression

A regression suite is a set of tests that are run on the application after every software or data change, and every night or every weekend, in order to check that no new bugs have been introduced.

This is vital because every bug fix, on the average, introduces one fifth of a bug.
Contradictory Requirements

A regression suite must be:
- Comprehensive so that it is likely to catch all the !@#$% bugs introduced
- Small so that it can economically be executed many times

How can we make our regression suite small and comprehensive?

The Set Cover Problem

Let $S = \{1 \ldots M\}$

Let $T = \{T_1, \ldots, T_n\}$ be a set of subset of $S$

The set cover problem:

Find the smallest subset of $T$ that covers $S$

The set cover problem is NP-C, however, there are a number of good algorithms for it.
The Greedy Heuristic

Find all the tests that contain unique tasks

- Loop
  - Remove all the tasks covered by selected tests
  - Choose the test that covers most remaining tasks

- Implementation issues:
  - Input type (sparse or dense)
  - Frequency of use
  - Real data vs. random data

Compaction Strategies

Simple - if a test introduces a new task, choose it.

Merge N - add every set of N new tests to the regression suite, and compact them into a new regression suite. (Size may decrease!)
Insight on Regression Suite Compaction

- An improvement of fifty percent is to be expected in Merge over Simple.
- Real information does not behave in the same way as randomly generated information.
- The real resource limitation is space.
- Incremental compaction is useful.
- The size of the regression suite does not change by much, once the set is fully covered.

Outline

- Introduction to Coverage
- Code Coverage
- Regression
  - Functional Coverage
- Class Exercise
- Automatic Coverage Directed Generation
- Comparing Functional Coverage to Code Coverage
- Conclusions
Functional Coverage

- Theater 2
- L.A. MeltDown

Central Computer

Star Tickets

Theater 1
- Godzilla goes to Town

Summary of ATAC on a Number of Tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Test Cases</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tickets.c:main</td>
<td>28 of 28</td>
<td>100%</td>
</tr>
<tr>
<td>tickets.c:GetParams</td>
<td>27 of 63</td>
<td>42.9%</td>
</tr>
<tr>
<td>tickets.c:HandleTheatre1</td>
<td>6 of 6</td>
<td>100%</td>
</tr>
<tr>
<td>tickets.c:HandleTheatre2</td>
<td>8 of 9</td>
<td>8.9%</td>
</tr>
<tr>
<td>tickets.c:HandleAgency</td>
<td>39 of 43</td>
<td>90.7%</td>
</tr>
<tr>
<td>tickets.c:GetOp1</td>
<td>7 of 8</td>
<td>87.5%</td>
</tr>
<tr>
<td>tickets.c:GetOp2</td>
<td>5 of 6</td>
<td>83.3%</td>
</tr>
<tr>
<td>tickets.c:Service2</td>
<td>2 of 8</td>
<td>25%</td>
</tr>
<tr>
<td>tickets.c:Service</td>
<td>24 of 32</td>
<td>75%</td>
</tr>
<tr>
<td>tickets.c:PrintQueue</td>
<td>38 of 41</td>
<td>92.7%</td>
</tr>
<tr>
<td>tickets.c:PrintState</td>
<td>17 of 18</td>
<td>94.4%</td>
</tr>
<tr>
<td>tickets.c:PrintIO</td>
<td>22 of 22</td>
<td>100%</td>
</tr>
</tbody>
</table>

Total: 223 of 284 | 78.5%

\( \chi^{ATAC} \)

Coverage: block

Functions: 12 of 12

Test cases: 7 of 7
Coverage Questions Not Answered by Code Coverage Tools

- Did every agency asked for every type of ticket?
- Did the agencies ask for service at the same time?
- How many times did Lion wait more than 10 minutes?
- Is the system fair?

Example of a Simple Functional Coverage Model

A model is composed of the following parts:
- A semantic description (story) of the functional coverage model which contains attributes
  - Has every agency {Agency name} asked for every type of show {Service type}
- A tuple of attributes <Agency name, Service type>
- Possible values for each attribute (e.g., Agency name: Disney, Fox, STAR, LION)
- Restrictions on the cross-product
What are Attributes Made Out Of?

Functional coverage attributes may come from any relevant source.
Our favorite sources include:
- Inputs
- Outputs
- Program variables
- Functions on program variables
- High level descriptions not mapped to variables

"Homemade" Functional Coverage Implementation

Instrument your code with the following statement:

```
print("Agency name = %d Service type = %d \n", agency, service);
```

Execute > trace

cat *.trace | sort -unique > output

~ What are the problems?
Simple Functional Coverage - Attributes

Simple Functional Coverage - Cases
Simple Functional Coverage - Functions

/* example of call:   traceFunc("showbiz",agency,service); */
#include <stdio.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/time.h>

FILE *trace_file_out;

void traceFunc(char *modelname, char *agency, int service) {
    struct timeval val;
    struct timezone zone;
    char filename[100];
    static int called = 0;
    if (called == 0) {
        gettimeofday(&val, &zone);
        sprintf(filename, "%s_trace_%d", modelname, val.tv_sec);
        trace_file_out = fopen(filename, "w");
    }
    called = 1;
    fprintf(trace_file_out, "%s %d \n", agency, service);
    return;
}

Elementary Report Generation Screen for the Ticket Example

Comet: Coverage measurement for model
Service type

Pick the columns to be displayed: (default is all)

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Agency Name</td>
</tr>
</tbody>
</table>

Select data where: (default is all data for the CALC_101 domain)
- Agency Names (select one or more -- selecting none gets all)
  1. Disty
  2. Ico
  3. STAR
  4. ION
- Service types (select one or more -- selecting none gets all)
  1. 4
  2. 0
  3. 1
  4. 2
  5. 4
  6. 6
A Basic Functional Coverage Report

Query Results:

<table>
<thead>
<tr>
<th>Agency Name</th>
<th>Service type</th>
<th>COVERED</th>
<th>COUNT_COV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dinky</td>
<td>1 out of 1 (100%)</td>
<td>10564</td>
<td></td>
</tr>
<tr>
<td>Dinky</td>
<td>2 out of 1 (100%)</td>
<td>3464</td>
<td></td>
</tr>
<tr>
<td>Dinky</td>
<td>5 out of 1 (100%)</td>
<td>2122</td>
<td></td>
</tr>
<tr>
<td>Dinky</td>
<td>6 out of 1 (100%)</td>
<td>1310</td>
<td></td>
</tr>
<tr>
<td>Fox</td>
<td>1 out of 1 (100%)</td>
<td>10155</td>
<td></td>
</tr>
<tr>
<td>Fox</td>
<td>2 out of 1 (100%)</td>
<td>3415</td>
<td></td>
</tr>
<tr>
<td>Fox</td>
<td>5 out of 1 (100%)</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>Fox</td>
<td>6 out of 1 (100%)</td>
<td>1308</td>
<td></td>
</tr>
<tr>
<td>STAR</td>
<td>1 out of 1 (100%)</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>STAR</td>
<td>2 out of 1 (100%)</td>
<td>3735</td>
<td></td>
</tr>
<tr>
<td>STAR</td>
<td>5 out of 1 (100%)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>LION</td>
<td>1 out of 1 (100%)</td>
<td>3872</td>
<td></td>
</tr>
<tr>
<td>LION</td>
<td>2 out of 1 (100%)</td>
<td>2201</td>
<td></td>
</tr>
<tr>
<td>LION</td>
<td>5 out of 1 (100%)</td>
<td>9901</td>
<td></td>
</tr>
<tr>
<td>LION</td>
<td>6 out of 1 (100%)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

More Functional Models for the Simple Example

1. <Agency name, Reply>
2. <Agency name, Reply, Service>
3. <Agency name, Reply, Service, Time>
4. <Agency1 calling, Agency2 calling, Agency3 calling, Agency4 calling>
5. <Agency1 reply, Agency2 reply, Agency3 reply, Agency4 reply>
6. <Service, Service>
7. Round Robin Bits
Arbitration Data

Query Results:

<table>
<thead>
<tr>
<th>Agency</th>
<th>Service</th>
<th>Response</th>
<th>Cycles</th>
<th>COVERED</th>
<th>COUNT_COV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disney</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4 out of 4 (100%)</td>
<td>9275</td>
</tr>
<tr>
<td>Fox</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4 out of 4 (100%)</td>
<td>8059</td>
</tr>
<tr>
<td>Fox</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4 out of 4 (100%)</td>
<td>1013</td>
</tr>
<tr>
<td>STAR</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4 out of 4 (100%)</td>
<td>7576</td>
</tr>
<tr>
<td>STAR</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4 out of 4 (100%)</td>
<td>1765</td>
</tr>
<tr>
<td>STAR</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4 out of 4 (100%)</td>
<td>686</td>
</tr>
<tr>
<td>LION</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4 out of 4 (100%)</td>
<td>5084</td>
</tr>
<tr>
<td>LION</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4 out of 4 (100%)</td>
<td>2048</td>
</tr>
<tr>
<td>LION</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4 out of 4 (100%)</td>
<td>1016</td>
</tr>
<tr>
<td>LION</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4 out of 4 (100%)</td>
<td>603</td>
</tr>
</tbody>
</table>

Rows Printed: 10
Total Rows Printed So Far: 10

Improvement Over Time
How Restrictions are Created

- The designer creates a list of restrictions for a model.
- Coverage feedback modifies the restrictions.
- Sub-models are used to economically check and refine the restrictions.

Restrictions are Self-correcting
Restrictions are Created Hierarchically

Car Assembly Line Example
Simple Coverage Models

< Car type, engine size, color >

< Car type, total work time, total elapse time, day >

< # cars in line, # of workers in line >

White Box Model -
Cars Location on the Assembly Line

(CL1, CT1, CL2, CT2, CL3, CT3, CL4, CT4, CL5, CL6, CL7, Problem)

<table>
<thead>
<tr>
<th>Car</th>
<th>Location</th>
<th>Car Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0-9</td>
<td>Accord-Civic</td>
</tr>
<tr>
<td>2</td>
<td>0-8</td>
<td>Accord-Civic</td>
</tr>
<tr>
<td>3</td>
<td>0-7</td>
<td>Accord-Civic</td>
</tr>
<tr>
<td>4</td>
<td>0-5</td>
<td>Accord-Civic</td>
</tr>
<tr>
<td>5</td>
<td>0-4</td>
<td>Accord</td>
</tr>
<tr>
<td>6</td>
<td>0-4</td>
<td>Accord</td>
</tr>
<tr>
<td>7</td>
<td>0-2</td>
<td>Accord</td>
</tr>
</tbody>
</table>

Problem: (0 = none, 1 = full stop, 2 = electrical shortage)
So there are: 10·9·8·6·5·5·3·2·2·2·2·3 = 15,552,000 tasks ?!
Model Restrictions

- \( CL_1 \geq CL_2 \geq CL_3 \geq CL_4 \geq CL_5 \geq CL_6 \geq CL_7 \)
  
  No car may pass another in the assembly line

- If \( CT_2 == \text{Civic} \) then \( CL_2 > 4 \) or \( CL_2 = 0 \)
- If \( CT_3 == \text{Civic} \) then \( CL_3 > 4 \) or \( CL_3 = 0 \)
- If \( CT_4 == \text{Civic} \) then \( CL_4 > 4 \) or \( CL_4 = 0 \)
  
  Civic may only be at stages 5 to 9 for painting

More Model Restrictions

- Only one car can be at stages 3 and 7
- Only one car can be at stages 5 and 6
- Only one car can be at stages 6 and 7
- Only one car can be at stages 1, 2 and 3
- Only two Accords can be at stages 4, 5 and 6

Only 1415 legal tasks!!!
Restriction Implementation

Simulation Outputs

Query Properties:

<table>
<thead>
<tr>
<th>Database Name</th>
<th>MICA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Name</td>
<td>Assembly line</td>
</tr>
<tr>
<td>Model Name</td>
<td>MODEL2</td>
</tr>
<tr>
<td>TCL Name</td>
<td>TCL_MODEL2</td>
</tr>
</tbody>
</table>

Selected Values:

- CT1: CIVIC
- CT2: CIVIC
- CT3: CIVIC
- CT4: CIVIC

COUNT_COV Range: 0 and above

61-62
Simulation Outputs

Query Results:

<table>
<thead>
<tr>
<th>CL1</th>
<th>CL3</th>
<th>CL4</th>
<th>CL5</th>
<th>CL6</th>
<th>CL7</th>
<th>COVERED</th>
<th>COUNT COV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Not creating enough of the right kind of scenarios

Performance related bugs

Illegal scenarios found
Functional Coverage Models

Model Classification Based on Information Sources

- **Black Box** - based on interface details
- **White Box** - based on internal details
- **Broken Box** - combined internal and external information
Model Classification Based on Temporal Properties

**Snapshot** - a single point in time
Could be abstract. For example, if two routines are called sequentially, a model containing the values of variables from both routines is considered a snapshot model even though the variables are not "alive" at the same time.

**Temporal** - a coverage model that checks scenarios
Usually the type where event 1 happens first, then event 2, and then event 3. Coverage is measured on the events (event 1 could have a few legal types), on time between events, and on the way the events interact.

Special Functional Coverage Models

**Stress** - Coverage model that checks how many things happen within a short interval of time, or at the same time. Also a model that checks that resources have been used to their full capacity.

**Statistically oriented** - Coverage model used to measure statistics such as how many times each task has been covered. The main motivation is usually performance. (Do we need a stack that large?)
### Invent Your Own Model

<table>
<thead>
<tr>
<th></th>
<th>Ticket office</th>
<th>Car assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistically oriented</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Invent Your Own Model - cont.

<table>
<thead>
<tr>
<th></th>
<th>Snapshot</th>
<th>Temporal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Box</td>
<td></td>
<td></td>
</tr>
<tr>
<td>White Box</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broken Box</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Model Hierarchy

- Building a model hierarchically, out of sub-models, is desirable for a number of reasons
  - Errors can be found earlier
  - Implementation is cleaner and faster
  - Restrictions are more easily debugged
- Sub-models should be made out of semantically related variables
- High level models can be reused many times

Comet - COverage MEasurement Tool

A generic functional coverage tool for use in a heavy industrial setting
Development leaders: Amir Eliaz and Raanan Grinwald
Overview of Comet

- Measures coverage of a given set of event traces over user-defined models
- Based on a relational database as the repository for coverage data
- Outputs various types of coverage reports

System Overview

```
Trace ----> COMET ----> Reports
          |          |          |
          v          v          v
Model ----> COMET ----> Regression Suites
```

```
COMET

Relational Database
```

73-74
Comet Tool Components

- GUI components
  - Model Definition
  - Coverage Measurement Control
  - Coverage Report Engine

- Insertion and measurement of coverage data engines

Preparing the Domain

- The Comet domain is a set of attributes with which coverage data is expressed
- Based on the domain attributes, Comet creates database tables in the designated database
- Event trace for storing traces
- Trace table for storing trace history information
- ...
Populating the Database

- The trace analyzer is written, for every domain, to extract the attribute values out of the raw data.
- The resulting traces are inserted into the database. Every line in the trace is an event.

```
HOST ppc .haifa.ibm.com
FILE default.tst elia
DOMAIN FP
TRACE
    ATTR NAMES cluster processor count mnemonic opcode s sign s fraction
s bin frac s exponent s type s sign s fraction s bin frac s exponent
s type s sign s fraction s bin frac s exponent s type t sign t fraction
 t bin frac t exponent t type fpscr before round mode fpscr after
    "stub" "" " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " 
```
Functional Coverage Models in the Domain Paradigm

Comet models are composed of:

- A semantic description
  - A query on the trace
- A tuple of attributes
  - Model attributes are not necessarily domain attributes
- Restrictions
  - Using SQL to mark some of the tasks as illegal

Model Definition Dialog
Coverage Measurement Control

- Monitoring the coverage process
- Controlling the insertion of traces into the database and which TCLs to measure them against
- Controlling various aspects of the measurement, such as collecting illegal task information, optimizing, etc.

Coverage Measurement Dialog
Coverage Report Engine Tool

- Querying the coverage data and getting specific cuts, views, statistics, graphs

- Examples (in the FP domain):
  - Did we see all FP instructions?
  - Did we see all rounding modes?
  - Source and Target operand types
  - Exponent difference for addition and subtraction
  - Uncovered subspaces (holes in the coverage)

CRE Application - Basic Dialog

![CRE application - Basic Dialog](image)
Basic Report Results

CRE Application - Progress Graph
CRE Application - 2D Scatter

Scatter Report Results
CRE Application - Summaries

Comet Report Engine - Summary Report
Two types of Coverage Directed Generation

- **Based on functional coverage**
  - Create a functional coverage model
  - Generate all the tasks
  - Transform the model into part of a test plan
  - Transform the tasks into tests

- **Based on an abstract model**
  - Describe the application using a formal model (automata)
  - Derive coverage tasks from the formal model
  - Derive tests from the coverage tasks
  - Expected results are built-in

- Usually easier to apply the first method, some tools are available

The Problem

- The test effort is a software project that requires design and implementation.

- High level test design documents are required for effective reviews and maintainability.

- Because of a lean test budget, the complete test design cycle is not implemented and quality is hindered.
A Solution for API Testing

- Functional models provide a simple language that enables high level description of the test design.
  - This level of description is easily reviewed and maintained, thus increasing quality.
- Low level test documents are produced automatically.
- Test implementation is straightforward, using low level test documents. Automatic generation of tests is also possible.
- Time saved by the automation is used to define stronger (e.g., time dependent) models.

An Example

- A file system being tested has migrated to support files that are bigger than 2 GB:
  - There are two types of file open: open and open64

- The functional model is defined as:
Test Automation

The attribute 'open file' is mapped to the class method:

```java
void open(StringBuffer apiName){
    System.out.println(apiName+ "("+"file1", O_RDWR, S_IRWXU)";}
```

open("open64") produces the call:

```java
open64("file1", O_RDWR, S_IRWXU);
```

The tool produces the following tuples to be tested:
(open, less than 2 GB), (open64, less than 2 GB), etc.

When automatically producing tests from the tuples, open64 is mapped to the method call
open("open64") thus producing the required code.
Model Based Automatic Testing - GOTCHA Testing Methodology

- Input is a functional specification
- Used to produce the model
- Generates abstract test case suite involving:
  - Sequences of rules and states (assertions and post conditions)
- Runs tests
- Has the following benefits:
  - 100% coverage of model
  - Early defect prevention:
    - Inconsistency
    - Ambiguity
    - Incompleteness

TCBeans / GOTCHA Software Testing Methodology - Test Design

Functional Specification

Early Defect Discovery: Inconsistency Ambiguity Incompleteness

100% Model Coverage

GOTCHA Engine

GOTCHA Abstract Test Suite

TCBeans Test Object(s)
TCBeans GOTCHA Software Testing Methodology - Test Execution

GOTCHA Abstract Test Suite

TCBeans GOTCHA Factory

TCBeans Test Object(s)

Application Under Test

Start/Stop Test
Customize Test Objects
Monitor Results

Defect Discovery/Resolution
Model Adjustment

The GOTCHA Definition Language

States

Start Final

101-102
GOTCHA Suite Generation

- **Input**
  - GOTCHA Definition Language model
  - Coverage model - projected state/transition
  - Definition of start and final states for tests

- **Output**
  - Test suite guaranteed to cover all coverable tasks
  - List of reachable, but uncoverable, tasks

- **Test**
  - Execution path from a start state to a final state

- **By-product**
  - Violated invariance encountered

Model Based Automatic Testing - TC Beans GOTCHA Test Case Factory

- **JavaBeans Test Case Component**
- **Input:** GOTCHA abstract test suite (text)
- **Output:** concrete test case execution and results
  - No additional source files
  - No explicit compile and link phase
  - One hundred percent coverage of model

- **Target and Expected object (JavaBeans)**
  - Rules = target method invocation
  - States = expected property setting

- **Process:**
  - Parse test suite
  - Execute
  - Compare target and expected properties

- **Interactive customization with BDK**
Model Based Automatic Testing -
A Use Case Scenario

1. Tester reviews functional specification.
2. Tester codes a behavior model in GOTCHA.
3. Tester uncovers defects in specification.
4. Tester runs GOTCHA to create an abstract test suite.
5. Tester codes a Java class (JavaBeans):
   - Wrapper methods to application methods
   - Wrapper properties to application state variables
   - Static properties for interactive customization
6. Tester runs TCBeans GOTCHA Test Case Factory.
7. Tester customizes target properties.
8. Tester runs test, uncovers lots of bugs, celebrates.
9. Tester analyzes structural coverage, adjusts GOTCHA model.

GOTCHA Hardware Test Study
Availability

**Code Coverage vs. Functional Coverage**

- **Code coverage**
  - Many tools are available for most operating systems/languages
    - Tools are hard to find or are very expensive for custom made/embedded environments
    - Tools are hard to implement

- **Functional coverage**
  - Limited functionality tool available (Comet)
    - Works in any environment
    - Preparing a simple tool or script is easy
Learning Curve
Code Coverage vs. Functional Coverage

- **Code coverage**
  - Users can benefit from the tools almost immediately
  - The tools are easy to use; usually plug and play

- **Functional coverage**
  - Need some time to learn how to define and implement models
  - Tools are not as easy to use
  - Need to have at least one person in the organization with a working knowledge of functional coverage methodology
  - Takes about 3 days to acquire knowledge

Cost
Code Coverage vs. Functional Coverage

- **Code coverage**
  - Cost of tools is usually low
  - Almost no education cost
  - Slowdown of 10 - 100 percent

- **Functional coverage**
  - Higher cost to develop tools and models
  - Higher education costs
  - Slowdown of a few percent (variable in # of models)
Experience

Code Coverage vs. Functional Coverage

- **Code coverage**
  - Used by many projects and in many standards
  - Papers on correlation between good coverage at unit level and the number of integration bugs
  - Refinements in tools as a result of experience

- **Functional coverage**
  - New field, not a lot of experience
  - Harder to transfer experience from one project to another

So Why Use Functional Coverage?
Focus
Code Coverage vs. Functional Coverage

- Code coverage
  - Uniformly spread over the entire program
  - Hard-coded coverage model hierarchy
  - All models are local

- Functional coverage
  - Risk-driven
  - Models can be local or global

Adapting to Testing Resources
Code Coverage vs. Functional Coverage

- Code coverage
  - Hard to adapt; hard-coded models
  - Hard to adapt; very primitive focusing options

- Functional coverage
  - Easy to adapt
  - Can use the right number of models with the desired granularity
  - Built-in model hierarchy
Impact on Project

Code Coverage vs. Functional Coverage

- Code coverage
  - Impacts testing process and testing requirements

- Functional coverage
  - Impacts the design itself from early design stages
  - Enables better understanding of programs and environments
  - Helps with test specifications and requirements

Outline

- Introduction to Coverage
- Code Coverage
- Regression
- Functional Coverage
- Class Exercise
- Automatic Coverage Directed Generation
- Comparing Functional Coverage to Code Coverage
- Conclusions
Which Coverage Method to Use?

- **Functional coverage**
  - High risk areas
  - Complex, error prone areas
  - Changes to existing code (maintenance)
  - Used when cheaper than manual test inspection
  - Used for global coverage models

- **Code coverage**
  - On all code that is written, given sufficient resources
  - As a criterion for finishing unit testing

Conclusions

- Code coverage is easier to use, cheaper, and more available than functional coverage.
- Given the resources, code coverage should be used on the entire code base.
- Functional coverage is an excellent way to improve testing of risky, error prone areas.
- The quality of functional coverage depends directly on the quality of the created functional coverage models
Conclusions - cont.

- Functional coverage can help at all design stages. Functional coverage models should be created at the high level design stage.

- Combining code based coverage and functional coverage can lead to high quality testing.

Any Questions?