Recent Issues in Software Testing:
Part A

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong
Speaker Biographical Sketch

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project: *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*)
  (http://paris.utdallas.edu/sere12)
Outline

• Testing for Software Safety
• Scalability and Integration of Techniques and Tools
• Application of Scientific Research Results in an Industrial Context
Testing for Software Safety
Software in Safety-Critical Systems

- Software safety is critical for software-dependent systems in which failure can result in property damage, injury, or death
  - Aeronautics
  - Astronautics
  - Medicine
  - Nuclear Power Generation
  - Communication
  - Transportation
  - Finance
  - etc.
**What is Software Safety**

- Software should execute within a system context without contributing to hazards
  - Software operation should not directly or indirectly lead to a hazard (and the consequences of the hazard realization could cause damage not just restricted to *financial* losses, or losses in terms of *time* or *property*, but rather may also include the loss of *life*)

- Software reliability ≠ Software safety
  - Software reliability models cannot be used for safety assurance (Parnas, 1990)

- Safety versus Security

- Good testing versus bad testing
For software-dependent and safety-critical applications, we need to test whether or not the hazardous conditions identified by design-level fault tree analysis will occur in the target implementation.

Testing for consistency between implementation and functional specifications does not provide safety assurance.

– Even if software behaves according to its functional specifications, it can still be unsafe.

– Need to examine not only the relationship between the inputs and outputs of a software system but also that between the inputs and the impact of the outputs on system behavior.
Challenges (2)

- “Selective redundancy” and “design for a single failure” do not prevent multiple-failure accidents from occurring.

- It is difficult to generate tests for safety testing by only using the hazard analysis results represented in the fault tree models, because of the lack of an explicit and common description of the relationship between a fault tree and the corresponding functional specifications.
Methodology

- Develop an efficient method for test generation using both functional specifications and safety requirements.

- Three major parts:
  - Integrate fault tree models into functional specifications so as to identify testable interactions between intended behaviors and hazardous conditions.
  - Develop a test generator that produces not only functional tests but also safety tests for a target implementation in a cost-effective way.
  - Develop a testing environment for executing generated functional and safety tests and evaluating test results against expected behaviors or hazardous conditions. It includes a test harness as well as an environment simulation of external events and conditions.
Importance and Benefits

- If there is a critical failure mode for the operation of a software system, then all possible ways that mode could occur must be discovered and tested in order to ensure the safe operation.

- A systematic approach to safety assurance of critical software can significantly reduce accidents caused by any unexpected combination of failures.

- It helps not only test software under normal conditions but also shows that unsafe states cannot be generated by the software as the result of feasible single or multiple erroneous inputs.
Integration of Safety and Functional Requirements

- Integration of results from hazard analysis in fault trees with functional specifications in UML behavior state machines
  - How to simplify a fault tree by deleting all the infeasible and non-causal events and gates?
  - Given a minimal cut set of a fault tree, how to construct a corresponding subtree which covers all necessary events leading to the top event?
  - How to transform a subtree to a minimal cut set transition?
  - How to add the minimal cut set transitions to a UML behavior state machine?
  - Identify testable interactions between intended behaviors and hazardous conditions

More discussion on the integration of UML Machines & Fault Trees, and subsequent test generation based on this integrated model
References


Scalability & Integration of Techniques and Tools and Application of Scientific Research Results in an Industrial Context
Objective

- With the increased emphasis on software quality and the pressure to deliver software *faster and cheaper*, it is imperative that we provide an *integrated solution* to help practitioners *produce high quality software at low cost*.

Solution = Methodology + Tool Support + Experience (Case Studies)

- Conduct an *effective technology transfer* to reduce the gap between state-of-art and state-of-practice.
Our Approach

• Take advantage of sophisticated analysis of the static and dynamic behavior of the design models or programs under examination for more effective and efficient coverage testing, debugging, understanding, profiling, risk analysis, etc.

• An integrated solution to address these concerns simultaneously.
  – It is also crucial to have a set of effective tools with user-friendly interfaces available to software practitioners to apply this solution in real-life contexts.
Our Experience (1)

- Telcordia Technologies (formerly Bellcore or Bell Communications Research)
  - **Suds** (Software understanding and diagnosis systems): a set of software testing, analysis, and understanding tools for C and C++ programs
    - **ATAC**
    - **Slice**
    - **Regress**
    - **Vue**
    - **Prof**

- UTD/Avaya Labs Research
  - **eXVantage** (eXtreme Visual-aid novel testing and generation): a tool suite for code coverage prioritization, test generation, test execution, debugging, and performance profiling of Java, C, and C++ programs
  - eXVantage is based on the **JBT** (Java Bytecode Testing) tool suite developed at UTD since 2002
Our Experience (2)

• UTD/Texas Instruments
  – **BATS** (a Bug Analysis and Tracking System) for improving software quality by effective defect analysis
  – **TestBED** for coverage Testing emBEDded software (OMAP/Symbian)

• UTD
  – **χSuds/SDL** for testing and diagnosis of software design specification

• UTD/Raytheon/NSF
  – **RBT/FPA**: Risk-based Testing and Fault-Proneness Analysis
Based on our previous experience, our ongoing effort is the creation of an integrated solution, with a suite of tools, to support software development on the Android platform.
The $\chi$Suds Toolsuite
http://xsuds.argreenhouse.com
**Telcordia Software Visualization and Analysis Toolsuite**

Chapter 2

**ATAC: A Tutorial**

The chapter illustrates how the basic features of ATAC work.

In this tutorial, we describe how the basic features work using small examples, a program that creates a random binary tree, and a program that sorts a list of numbers.

1. Random Binary Tree
2. Sorting a List of Numbers

Before using ATAC, make sure the most recent code from the appropriate web site or mirror is installed.

[Image of ATAC interface]

Recent Issues in Software Testing (© 2012 Professor W. Eric Wong, The University of Texas at Dallas)
Use of χSuds

- χSuds is renamed as *Telcordia Software Visualization and Analysis Toolsuite*

- It is maintained and supported by the Applied Research Division and used internally at Telcordia
  - A new version sponsored by the US Army Research Lab has been implemented and deployed to the customer

- It is also used by IBM as *IBM C and C++ Maintenance and Test Toolsuite*

- In addition, institutions such as Johnson Space Center/NASA, Chunghwa Telecom (Taiwan), etc. also have χSuds in-house

- The toolsuite is also used by more than 25 universities worldwide for teaching software testing at both undergraduate and graduate levels

---

(1) χSuds User’s Manual, Telcordia Technology
\textbf{\textit{KA\textsc{TA}C}:}

Greatly Improves Testing through
Targeted Test Generation and Precise Coverage Measurement
**Efficient Coverage Testing (1)**

- How much of the code is currently tested?
  
  What is missing?
  
  - Decompose coverage *based on users’ needs*
  
  - Which statements, decisions, def-use pairs were exercised?

- Need help in creating tests?
  
  - Which statement should I try to cover next in order to achieve high coverage in an effective way?
  
  - Prioritizes uncovered code in different colors

Analyzing the controlflow graph of the program to find the dominant blocks, decisions, and def-use pairs.

When a test covers highly dominant blocks it will cover many other blocks.
Efficient Coverage Testing (2)

Use prioritization and visualization to provide hot spots that give the most value in coverage. Each color represents a different weight determined by a control flow analysis using the concept of superblocks and dominators.

Code in white has already been covered by a test case and covering it again will not add new coverage.

Covering this red block guarantees the execution of at least 8 additional blocks.
Efficient Coverage Testing (3)

Covering either true or false branch guarantees the execution of at least another 8 branches.
χATAC Demo: Coverage Testing of C Code

Compile code with χATAC

Initial display

χSuds User’s Manual

Source display after executing wordcount.1

Source display after executing wordcount.2

100 % block coverage after executing wordcount.5

Source display after executing wordcount.6

100 % block & decision coverage after executing wordcount.9
χSlice:
Pinpoints the Location of Faults from Failures

• Do you know how to locate bugs quickly?
• Do you spend too much time finding faults in your program?
• Can you narrow down bugs to files, then functions, then lines of code?
χSlice for Fault Localization (1)

Different colors indicate whether a file contains code that is executed by the failed test and/or the successful test.

χSlice

Coverage: block

Files: 7 of 7

Passed Tests: 1 of 4

Failed Tests: 1 of 4

Summary TestCases Update GoBack Help

<table>
<thead>
<tr>
<th>File</th>
<th>Options</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>by-type</th>
<th>by-file</th>
<th>by-function</th>
<th>Disable</th>
<th>Sort by</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>69 of 178</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc.c</td>
<td>18 of 101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort.c</td>
<td>3 of 33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp1.c</td>
<td>0 of 102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp2.c</td>
<td>0 of 112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip.c</td>
<td>0 of 29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sort.c</td>
<td>0 of 59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total 90 of 614 14.7%
\textbf{χSlice for Fault Localization (2)}

\begin{itemize}
  \item [\textbf{Code in blue}] is executed by the failed test \textbf{AND} the successful one.
  \item [\textbf{Code in red}] is executed by the failed test \textbf{BUT NOT} the successful one.
  \item [\textbf{Code in white}] is \textbf{NOT} executed by the failed test.
\end{itemize}

\begin{verbatim}
p = (struct msg *)lsp; 
    j = 0;
for(i=a; i < b; i++) {
    p = setfil(i);
    if(i == 0)
      p->b = stdin;
    else if((p->b = fopen(t, "r")) == NULL)
      cant(f);
    ibuf[] = p;
    if(!rline(p)) j++;
    p++;}
do { 
  i = j;
  qsort((char **)ibuf, (char **)ibuf+i);
  l = 0;
  while( l < i )
    cp = ibuf[i]->l;
    if(*cp == '\n' ) {
      l = l + 1;
      if(rline(ibuf[i])) {
        k = i;
        while(++k < i)
          ibuf[k-1] = ibuf[k];
        j--;}
    }
}
\end{verbatim}

\textbf{χSlice}

- File: main.c
- Line: 152 of 240
- Coverage: block
- Highlighting: all prioritized

---

Recent Issues in Software Testing (© 2012 Professor W. Eric Wong, The University of Texas at Dallas)
χRegress:
Significantly Reduces Regression Testing Costs

- Do you spend excessive resources in regression testing?
- Do you know how to select effective regression tests?
- Is your regression test suite too large to manage?
### Re regress for Effective Regression testing

<table>
<thead>
<tr>
<th>File</th>
<th>Tool</th>
<th>Options</th>
<th>Summary</th>
<th>TestCases</th>
<th>Update</th>
<th>GoBack</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>function_entry</td>
<td></td>
<td>block</td>
<td>decision</td>
<td>c_use</td>
<td>p_use</td>
<td>Disable</td>
<td>Minimize_in</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cumulative coverage summary by testcase over selected coverage types</th>
</tr>
</thead>
<tbody>
<tr>
<td>wordcount.9</td>
</tr>
<tr>
<td>wordcount.3</td>
</tr>
<tr>
<td>wordcount.4</td>
</tr>
<tr>
<td>wordcount.14</td>
</tr>
<tr>
<td>wordcount.5</td>
</tr>
<tr>
<td>wordcount.8</td>
</tr>
<tr>
<td>wordcount.16</td>
</tr>
<tr>
<td>wordcount.2</td>
</tr>
<tr>
<td>wordcount.11</td>
</tr>
<tr>
<td>wordcount.12</td>
</tr>
<tr>
<td>wordcount.13</td>
</tr>
<tr>
<td>wordcount.6</td>
</tr>
<tr>
<td>total</td>
</tr>
</tbody>
</table>

**χ Regress**

Coverage: block

Test cases: 5 of 17
**χVue:**

Allows the Programmer to See Where User Functions Are Implemented in the Software

- Do you know where features are implemented?
- Do you spend excess time in resolving trouble reports?
- Can you visualize features in code?
Requirement changes and enhancement requests are usually specified in terms of features affected, not in terms of software components that must be modified.

A feature is an abstract description of a functionality given in the specification.
- Example: the ATM software has three features
  - Withdraw
  - Deposit
  - Balance

Software developers must locate and understand the code associated with the affected features before they can translate change requests into code change.
**Program Comprehension (2)**

- **Well-designed Software Systems**
  - A high degree of cohesion
    - A cohesive module should ideally do just one thing
  - A low degree of coupling
  - Each module addresses a specific subfunction of the requirements and has a simple interface when viewed from other parts of the program structure
  - A clear mapping between each feature and its corresponding code segments

- **Software systems in the real world**
  - Low cohesion & high coupling
  - Program features are mixed together in the code – across modules which are seemingly unrelated
**Program Comprehension (3)**

- In a complex software system it is not unusual to find that modifications made to one feature, which can be viewed as a functionality of the system, have *adverse impacts* on other *seemingly unrelated features*.

- Such impacts can subsequently change the behavior of those features and *cause a system failure*.

- Need a good understanding of the system.
**Program Comprehension (4)**

- Locate program code relevant to a particular feature in order to provide software programmers and maintainers with a good *starting point* for quick program understanding
  - Develop novel heuristics and experiment with them to identify
    - Code unique to the given feature
    - Code common to the given feature and others
  - Examine factors which affect the code so identified

More discussion …..
**Program Comprehension (5)**

- Three different approaches
  - **Systematic**
    - Provides a good understanding
    - Impractical for large complicated systems
  - **As-needed**
    - Less expensive and less time-consuming
    - Miss some non-local interactions between features
  - **Execution Slice-based**
    - An execution slice is the set of program components executed by a test input
      
      ```plaintext
      Failed test (fault localization) ➔ Invoking test (program comprehension)
      Successful test (fault localization) ➔ Excluding test (program comprehension)
      ```

- **Qualitative description** versus **quantitative measurement**
χ Vue for Programming Understanding (1)
\( \chi \text{Prof}: \)

**Identifies Performance Bottlenecks Visually**

- Do you need to improve the performance of your program?
- Do you know which part of your program slows down the execution?
- Can you visualize the most frequently executed pieces in code?
- Are your performance measures repeatable?
### ΧProf for Performance Profiling (1)

**Table 1: Block Profiling Summary by File Over Selected Testcases**

<table>
<thead>
<tr>
<th>File</th>
<th>Options</th>
<th>Summary</th>
<th>Testcases</th>
<th>Update</th>
<th>GoBack</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc.c</td>
<td>by-type</td>
<td>2024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main.c</td>
<td>by-file</td>
<td>393</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: Block Profiling Summary by Function Over Selected Testcases**

<table>
<thead>
<tr>
<th>File</th>
<th>Options</th>
<th>Summary</th>
<th>Testcases</th>
<th>Update</th>
<th>GoBack</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc.c:count</td>
<td>by-type</td>
<td>2024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main.c:main</td>
<td>by-file</td>
<td>247</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main.c:print</td>
<td>by-function</td>
<td>146</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
χProf for Performance Profiling (2)

Code in red is executed 107813 times

Code in green is executed 30805 times
Design Philosophy for Testing/Maintenance Tools

• When we develop a testing/maintenance tool, we should consider
  – Ease of use
  – Visualization
  – Prioritization
  – Granularity
  – Incrementability
  – Extensibility
  – Portability
  – etc.
χSuds/SDL:
Coverage Testing Your Architectural Design in SDL
Graphical and Textual Representations

- The graphical representation is called GR and the textual representation is called PR (Phrase Representation).
  - Automatic translation between GR and PR can be done

```plaintext
process Game;
dcl Count Integer;

start;
    task Count := 0;
    nextstate Losing;

state Losing:
    input Probe;
    output Lose;
    task Count := Count - 1;
    nextstate -;
    input Bump;
    nextstate Winning;
    endstate;

state Winning:
    input Bump;
    nextstate Losing;
    input Probe;
    output Win;
    task Count := Count + 1;
    nextstate -;
    endstate;

state * :
    input Result;
    output Score(Count);
    nextstate -;
    input GameOver;
    stop;
    endstate;
    endprocess Game;
```
Life of an SDL Specification

- The same concepts are applied to SDL (Specification and Description Language) which is an object-oriented, formal language for designing complex, real-time and communicating systems.
**Alternating Bit Protocol**

- The alternating bit protocol, which is a simple form of the “sliding window protocol” with a window size of 1, is used as the example.

- It can be used to provide *reliable communication* over *non-reliable network channels* through a *one-bit sequence number* (which alternates between 0 and 1) in each message.

- The alternating bit protocol is constituted by a sender and a receiver who exchange messages through two channels, medium1 and medium2.
**Black Box-based Simulation**

```
bit: 0

sender
- start
- wait_put
- wait_am

medium1
- start
- wait_dm

medium2
- start
- wait_dm

receiver
- start
- wait_dm

[put]

put('world') from: env

[bit: 0]

[medium1, medium_error]
[medium2, medium_error]

[am, medium_error]
[am, medium_error]

put('hello') from: env
put('world') from: env

dm('hello', 0) from: sender
dm('hello', 0) from: sender
dm('world', 1) from: sender
dm('world', 1) from: sender

medium_error from: medium2
am(1) from: medium2
am(1) from: medium2

[get]

get('world') from: receiver

[bit: 0]

[am(1)]
[am(1)]
[am(1)]
[am(1)]

get('hello') from: receiver
get('world') from: receiver
```
**Question**

- Do you know how much of your SDL code has been tested?
- What is still missing?
- How to generate additional test cases to execute the uncovered code in an effective way?
**Research Issues on Coverage Testing (1)**

- **How to reduce the amount of instrumentation?**
  That is, how to reduce the number of probes inserted in the source code?
  - One probe per line of code
  - One probe per block
  - One probe per superblock
  - What are the pros and cons?

- **What kind of information should be collected at run time?**
  - Executed or not executed
  - Execution counts of each testable attribute
  - Execution sequence

- **What kind of information should be collected at parsing?**
  - Number of testable attributes
  - Locations of each testable attribute

- Can the techniques used for coverage testing software on Windows be applied to coverage testing embedded software? Real-time applications?
Research Issues on Coverage Testing (2)

- In addition to reporting what has been tested and what is still missing, we would also like to
  
  - Decompose coverage information based on users’ needs
    (do not just report a single number with respect to all the tests for the entire program)
  
  - Provide useful hints for efficient test generation to effectively increase the coverage of the program being tested
  
  - Use coverage as a filter to show how the number of test cases can be reduced significantly without sacrificing the overall code coverage
  
  - Test cases in the reduced subset have a higher priority to be executed when revalidating the program during the regression testing
  
  - Conduct effective fault localization based on how the program is executed by each test
Source Code-Based Risk Analysis
(Identify Fault-Prone Software Modules)
Challenges

- To reduce the risk of software operations, code which may cause problems must be identified early so that actions can be taken to prevent such problems from occurring.
  - A risk can be viewed as a potential problem.
  - A problem is a risk that has manifested.

- Not all code can be exhaustively reviewed and tested.

- Pareto’s principle applies to software.
  - Approximately 20% of a software system is responsible for 80% of the faults.

- An efficient and effective analysis is required to help identify only a small percentage of code as fault-prone.
  - The more complex a software system is, the more likely it is that programmers will make mistakes that introduce faults which in turn can lead to execution failures.

  - How should the complexity of each software module be determined?
    - Using a risk analysis-based approach
Methodology (1)

• A risk analysis-based model supported by tools to identify fault-prone software modules
  – Model Construction
    - Static metrics: can be collected at compile time (e.g., lines of code, McCabe complexity)
    - Dynamic metrics: cannot be collected until run-time (e.g., test coverage)
    - Internal metrics: focus on the internal structure of a software module (e.g., number of lines of code)
    - External metrics: emphasize the interactions between a software module and the rest of the system
    - Special metrics: unique to multi-threaded embedded software (e.g., number of threads through each module)
    - Process-based metrics: how a software system is built and operated
  – Model Validation and Refinement
    - Evaluate the model against real defect data
    - Refine it, if necessary, to improve its accuracy on fault-proneness prediction
    - Use open-source applications and software developed at our member companies, as available
  – Model Application
    - Repeat the risk analysis at different stages of the software development lifecycle
    - Our model is flexible as it uses various data depending on its availability
Methodology (2)

- Problems of current approaches
  - Only static and/or process-based metrics are used
  - No dynamic information such as how each module has been tested is considered
  - Results of many studies are inconsistent or contradictory
  - Not aimed at multi-threaded embedded software which requires consideration of concurrent operations

- Our solution overcomes existing drawbacks by calibrating the static metrics-based complexity using dynamic metrics, and also incorporating special metrics unique to subject software.
Methodology (3)

- Principal component analysis
- Statistical modeling
- Appropriate weight factors

Methodology

- Code-based
  - Generic
    - Static
      - Internal
      - External
    - Dynamic
      - Internal
      - External
  - Special
  
- Process-based
Quiz

• Given two modules $C_1$ and $C_2$
• $C_1$ has higher static metrics values but also is significantly more tested than $C_2$

• Is the probability of $C_1$ containing any faults still higher than that of $C_2$?

• We propose

The more complex the static structure of the code, the higher risk it has.
The more thoroughly the code is tested, the less risk it has.
Effective Defect Analysis
Welcome to the Bug Tracking and Analysis System

BATS: Bug Analysis and Tracking System

--For producing more reliable products

Overview of the defect analysis and tracking system

BATS is a Bug Analysis and Tracking System for effectively extracting semantic information from software defects in order to help determine the state of the software development process and suggest possible ways of improving the process. For inquiry, please send email to Professor Eric Wong.
**Objective**

- Develop a novel approach to effectively *extract information from defects* in order to help determine the state of the software development process and suggest *possible ways of improving the process*.

  - An early stage analysis of the defects can indicate actions to be taken in later stages to ensure the quality of a given product.
  
  - The final product that is shipped should have *the fewest possible defects*. 
An EDA/ODC-based Defect Analysis (1)

- An EDA/ODC-based defect analysis can bridge the gap between *quantitative statistical defect models* and *qualitative causal analysis.*
  - EDA: Efficient Defect Analyzer developed in the Applied Research Division at Telcordia Technologies (formerly Bellcore)
  - ODC: Orthogonal Defect Classification which was first used at IBM
An EDA/ODC-based Defect Analysis (2)

- The basic idea is to classify each defect by using several orthogonal categories which describe different aspects of the defect
  - Although any one category captures a limited amount of information, when all are considered together, a meaningful and rich characterization emerges.

- This process is similar to measuring the dimensions of an object
  - When a cube is measured, it is measured along an x, y, and z axis.
  - In measuring defects we measure along axes such as Defect Type, Defect Source, and Defect Impact.

- Once the defect data have been classified, the corresponding defect patterns are to be analyzed to uncover opportunities for process improvement and consequently reduce the number of defects in the final product
Defect Analysis by One Category

- A *pie chart* shows the *defect percentage distribution* with respect to a given category for a set of defects.

- Place the mouse on a segment to get the corresponding information about the defect category, defect class, its description, count, percentage, and the total number of defects.

---

**Defect Type: Interface**

Communication problems between modules, components, device drivers via macros, call statements, control blocks, parameter lists, etc.

**Count**: 13 (12.4%)

**Total number of records displayed** = 105
Drill Down of Pie Charts (1)

- This is the original pie chart.

Click on Reliability
Drill Down of Pie Charts (2)

- This is the *drilled down pie chart*.

When a particular class is selected as an additional criterion for drill down, the corresponding category is fixed to this class. It will reduce the set of defects which is currently under consideration. The reduced set of defects is then classified again along all the categories.
Defect Analysis by Two Categories: Bar Charts

• A bar chart shows the *defect counts* with respect to two selected categories for a set of defects.

• When the cursor is placed on a bar, the corresponding classes of the two selected categories are shown along with the number of defects.

In our example there are 15 defects whose defect source is “Old Function” and defect type is “Assignment”
**Defect Analysis by Two Categories: Mat Plots**

- A mat plot also shows the *defect counts* with respect to two selected categories for a set of defects.

![Defect Analysis Diagram]

There are 15 defects whose defect type is “Assignment” and defect source is “Old Function”.

---

Recent Issues in Software Testing (© 2012 Professor W. Eric Wong, The University of Texas at Dallas) 68
Example

- Suppose the following figure gives the classification based on the defects from the field for Product X

The first pie shows that design and specification defects account for 54% of the total defects.
- This seems too high.
- It suggests that there need to be better design reviews.

The second pie shows that among the coding defects, which account for 44% of the total defects, 41% of those are assignment problems and 35% are algorithmic problems.
- These numbers indicate that developers should be especially careful in these areas.
Challenges for Testing, Analysis and Debugging Multi-Threaded Embedded Software on Android
**Approach for Reproducible Execution**

- Execution sequence → Partial order of synchronous events
- Preserve the order and apply the same IO events → reproducible execution

---

**Diagram:**

- **T1**
  - $send_{1,1}$
  - $send_{1,2}$
- **T2**
  - $recv_{2,1}$
  - $send_{2,2}$
  - $b$
- **T3**
  - $recv_{3,1}$
  - $recv_{3,2}$
  - $send_{3,2}$
  - $recv_{3,3}$
  - $recv_{3,3}$
  - $recv_{2,3}$

- Additional nodes and edges are present in the diagram, indicating the execution sequence and event order.
A Framework for Recording & Replay

• Recording
  – Recording is optimized to reduce instrumentation overhead
    □ Considered in the schedulability analysis
    – A partially ordered thread-level interaction is used

• Replay
  – The replay is based on Lamport’s clock
  – Dynamic data collected from execution traces at the replay stage will be used for subsequent analysis and debugging

Reduce instrumentation during recording & include additional instrumentation and perform data collection during replay
The eXVantage Tool Suite
INTRODUCTION

eXVantage is a product line of eXtreme Visual Aid Novel Testing and Generation tools. The eXVantage family of test tools focuses on providing code coverage information to software developers and testers on a variety of platforms which may include various resource and performance constraints, i.e. embedded and real-time systems. The primary capability of the tools in the family is to execute tests and show how much of the code was executed during the tests, both as a percentage of the total code and as a display that shows which individual lines of code were or were not executed. Members of the eXVantage family have some, if not all, of the following capabilities:

Program Structure Recovery and Analysis
What are the dependencies among classes or other invokable program elements? (Class dependency graph)
What is the control flow for the program? (Control flow graph)
Which lines of code should have highest priority for testing so as to maximize coverage? (Priority analysis)
Is the program consistent with established rules of style? (Style checker)

Coverage
Which lines of code are executed as a result of running tests?
Which methods are executed as a result of running tests?
Which packages are executed as a result of running tests?

Slicing
What lines of code were executed by failed tests? By successful tests?

Testing Embedded Software
**TestBED for Coverage Testing of Embedded Software** (1)

- **a project specification** (.mmp file)
  - TestBED
  - source files
    - instrumentation by χSuds/ATAC
    - • extract static structures
    - • replace original probes by customized probes
    - instrumented source files
      - .atac files
      - temporary files with static structures & original probes
      - revised instrumented source files
  - Build
    - Host Machine
      - a log file
    - OMAP platform
      - • executing test cases using the Automatic Testing Framework (ATF)
      - • each customized probe sends a message via the debugger port to a log file on the host machine
      - Symbian OS components running on OMAP

Recent Issues in Software Testing (© 2012 Professor W. Eric Wong, The University of Texas at Dallas) 76
**TestBED for Coverage Testing of Embedded Software (2)**

- **Host Machine**
  - temporary files with static structures & original probes
  - log files
  - trace files
  - source files
  - .atac files

- **TestBED**
  - \( \chi \text{Suds} \)
    - coverage analysis/report
    - source code display
    - test set minimization
    - etc

- **Debug port**
  - Target platform (OMAP 1710)
  - Serial port

Communication between the debug port and the host machine