Advanced Topics in Combinatorial Methods for Testing

Rick Kuhn
National Institute of Standards and Technology
Gaithersburg, MD
Solutions to the oracle problem
How to automate checking correctness of output

• Creating test data is the easy part!

• How do we check that the code worked correctly on the test input?

  • **Crash testing** server or other code to ensure it does not crash for any test input (like ‘fuzz testing’)
    - Easy but limited value

  • **Built-in self test with embedded assertions** – incorporate assertions in code to check critical states at different points in the code, or print out important values during execution

  • **Full scale model-checking** using mathematical model of system and model checker to generate expected results for each input
    - expensive but tractable
Crash Testing

• Like “fuzz testing” - send packets or other input to application, watch for crashes

• Unlike fuzz testing, input is non-random; cover all t-way combinations

• May be more efficient - random input generation requires several times as many tests to cover the t-way combinations in a covering array

Limited utility, but can detect high-risk problems such as:
  - buffer overflows
  - server crashes
Built-in Self Test through Embedded Assertions

Simple example:
assert( x != 0);    // ensure divisor is not zero

Or pre and post-conditions:
/requires amount >= 0;

/ensures balance  == \old(balance) - amount &&
\result == balance;
Built-in Self Test

Assertions check properties of expected result:
- ensures balance == \old(balance) - amount && \result == balance;

- Reasonable assurance that code works correctly across the range of expected inputs

- May identify problems with handling unanticipated inputs

- Example: Smart card testing
  - Used Java Modeling Language (JML) assertions
  - Detected 80% to 90% of flaws
Using model checking to produce tests

The system can never get in this state!

Yes it can, and here’s how ...

- Model-checker test production: if assertion is not true, then a counterexample is generated.
- This can be converted to a test case.

Black & Ammann, 1999
Model checking example

-- specification for a portion of tcas - altitude separation.
-- The corresponding C code is originally from Siemens Corp.
Research
-- Vadim Okun 02/2002

MODULE main

VAR
  Cur_Vertical_Sep : { 299, 300, 601 };
  High_Confidence : boolean;

... init(alt_sep) := START_;
  next(alt_sep) := case
    enabled & (intent_not_known | !tcas_equipped) : case
      need_upward_RA & need_downward_RA : UNRESOLVED;
      need_upward_RA : UPWARD_RA;
      need_downward_RA : DOWNWARD_RA;
      1 : UNRESOLVED;
      esac;
    esac;

... SPEC AG ((enabled & (intent_not_known | !tcas_equipped) &
  !need_downward_RA & need_upward_RA) -> AX (alt_sep = UPWARD_RA))
-- "FOR ALL executions,
-- IF enabled & (intent_not_known ....
-- THEN in the next state alt_sep = UPWARD_RA"
Computation Tree Logic

- The usual logic operators, plus temporal:
  - $A \phi$ - All: $\phi$ holds on all paths starting from the current state.
  - $E \phi$ - Exists: $\phi$ holds on some paths starting from the current state.
  - $G \phi$ - Globally: $\phi$ has to hold on the entire subsequent path.
  - $F \phi$ - Finally: $\phi$ eventually has to hold
  - $X \phi$ - Next: $\phi$ has to hold at the next state
- [others not listed]

- execution paths
- states on the execution paths

SPEC $AG ((\text{enabled} \& (\text{intent\_not\_known} \mid \neg \text{tcas\_equipped}) \& \neg \text{need\_downward\_RA} \& \text{need\_upward\_RA}) \rightarrow AX(\text{alt\_sep} = \text{UPWARD\_RA}))$

- “FOR ALL executions, IF enabled $\&$ (intent\_not\_known .... THEN in the next state alt\_sep = UPWARD\_RA”
What is the most effective way to integrate combinatorial testing with model checking?

- Given $\text{AG}(P \rightarrow \text{AX}(R))$
  “for all paths, in every state, if $P$ then in the next state, $R$ holds”

- For k-way variable combinations, $v_1 \& v_2 \& \ldots \& v_k$

- $v_i$ abbreviates “$\text{var}1 = \text{val}1$”

- Now combine this constraint with assertion to produce counterexamples. Some possibilities:
  1. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow \text{AX} \neg(R))$
  2. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} \neg(1))$
  3. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} \neg(R))$
What happens with these assertions?

1. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \land P \rightarrow AX \neg (R))$

   $P$ may have a negation of one of the $v_i$, so we get $0 \rightarrow AX \neg (R)$ always true, so no counterexample, no test. This is too restrictive!

2. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg (1))$

   The model checker makes non-deterministic choices for variables not in $v_1..v_k$, so all $R$ values may not be covered by a counterexample. This is too loose!

2. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg (R))$

   Forces production of a counterexample for each $R$. This is just right!
More testing

Examples

Name: Julia
Teacher: Mrs. H.

First Grade Spelling Test

1. soft
2. lost
3. goat
4. toast
5. load
Buffer Overflows

- Empirical data from the National Vulnerability Database
  - Investigated > 3,000 denial-of-service vulnerabilities reported in the NIST NVD for period of 10/06 – 3/07
  - Vulnerabilities triggered by:
    - Single variable – 94.7%
      example: *Heap-based buffer overflow in the SFTP protocol handler for Panic Transmit … allows remote attackers to execute arbitrary code via a long ftps:// URL.*
    - 2-way interaction – 4.9%
      example: *single character search string in conjunction with a single character replacement string, which causes an "off by one overflow"*
    - 3-way interaction – 0.4%
      example: *Directory traversal vulnerability when register Globals is enabled and magic_quotes is disabled and .. (dot dot) in the page parameter*
Example: Finding Buffer Overflows

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

3.     conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

4.     pPostData = conn[sid].PostData;
5.     do {
6.         rc = recv(conn[sid].socket, pPostData, 1024, 0);

7.         pPostData += rc;
8.         x += rc;
9.     } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

......

3.     conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

......

4.     pPostData = conn[sid].PostData;
5.     do {
6.         rc = recv(conn[sid].socket, pPostData, 1024, 0);

......

7.         pPostData += rc;
8.         x += rc;
9.     } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11.   }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1.  if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

......

3.    conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

......

4.    pPostData = conn[sid].PostData;
5.    do {
6.        rc = recv(conn[sid].socket, pPostData, 1024, 0);

......

7.        pPostData += rc;
8.        x += rc;
9.    } while (((rc == 1024) || (x < conn[sid].dat->in_ContentLength)));

10.   conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }

true branch
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
   2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {
   ......
   3.         conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));
   ......
   4.         pPostData = conn[sid].PostData;
   5.         do {
   6.             rc = recv(conn[sid].socket, pPostData, 1024, 0);
   ......
   7.             pPostData += rc;
   8.             x += rc;
   9.         } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));
10.    conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';
11. }
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {

2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

3.         conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

4.             pPostData = conn[sid].PostData;

5.             do {

6.                 rc = recv(conn[sid].socket, pPostData, 1024, 0);

7.                 pPostData += rc;

8.                 x += rc;

9.             } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));

10.            conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';

11.         }

true branch

Allocate -1000 + 1024 bytes = 24 bytes
Interaction: request-method="POST", content-length = -1000, data= a string > 24 bytes

1. if (strcmp(conn[sid].dat->in_RequestMethod, "POST") == 0) {
   2.     if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {

   3.     conn[sid].PostData = calloc(conn[sid].dat->in_ContentLength + 1024, sizeof(char));

4.        pPostData = conn[sid].PostData;

5.        do {
6.            rc = recv(conn[sid].socket, pPostData, 1024, 0);

7.            pPostData += rc;

8.            x += rc;

9.        } while ((rc == 1024) || (x < conn[sid].dat->in_ContentLength));

10. conn[sid].PostData[conn[sid].dat->in_ContentLength] = '\0';

11. }

true branch
Allocate -1000 + 1024 bytes = 24 bytes
Boom!
Example: Modeling & Simulation

• “Simured” network simulator
  • Kernel of ~ 5,000 lines of C++ (not including GUI)
• Objective: detect configurations that can produce deadlock:
  • Prevent connectivity loss when changing network
  • Attacks that could lock up network
• Compare effectiveness of random vs. combinatorial inputs
• Deadlock combinations discovered
• Crashes in >6% of tests w/ valid values (Win32 version only)
# Simulation Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DIMENSIONS</td>
<td>1,2,4,6,8</td>
</tr>
<tr>
<td>2 NODOSDIM</td>
<td>2,4,6</td>
</tr>
<tr>
<td>3 NUMVIRT</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>4 NUMVIRTINJ</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>5 NUMVIRTEJE</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>6 LONBUFFER</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>7 NUMDIR</td>
<td>1,2</td>
</tr>
<tr>
<td>8 FORWARDING</td>
<td>0,1</td>
</tr>
<tr>
<td>9 PHYSICAL</td>
<td>true, false</td>
</tr>
<tr>
<td>10 ROUTING</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>11 DELFIFO</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>12 DELCROSS</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>13 DELCHANNEL</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>14 DELSWITCH</td>
<td>1,2,4,6</td>
</tr>
</tbody>
</table>

\[5x3x4x4x4x4x2x2x2x4x4x4x4x4x4 = 31,457,280\] configurations

Are any of them dangerous? If so, how many? Which ones?
# Network Deadlock Detection

## Deadlocks Detected: Combinatorial

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

## Average Deadlocks Detected: Random

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0.63</td>
<td>0.25</td>
<td>0.75</td>
<td>0.50</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>10.13</td>
<td>11.75</td>
<td>10.38</td>
<td>13</td>
<td>13.25</td>
</tr>
</tbody>
</table>
Detected 14 configurations that can cause deadlock:
\[
\frac{14}{31,457,280} = 4.4 \times 10^{-7}
\]

Combinatorial testing found more deadlocks than random, including some that might never have been found with random testing.

Why do this testing? Risks:
- accidental deadlock configuration: low
- deadlock config discovered by attacker: much higher
  (because they are looking for it)
Coverage Measurement
### Combinatorial Coverage Measurement

<table>
<thead>
<tr>
<th>Tests</th>
<th>Variables</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable pairs</th>
<th>Variable-value combinations covered</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ac</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ad</td>
<td>00, 01, 11</td>
<td>.75</td>
</tr>
<tr>
<td>bc</td>
<td>00, 11</td>
<td>.50</td>
</tr>
<tr>
<td>bd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>cd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
</tbody>
</table>

100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations
Graphing Coverage Measurement

100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations

Bottom line:
All combinations covered to at least 50%
Adding a test

Coverage after adding test [1,1,0,1]
Adding another test

Coverage after adding test [1,0,1,1]
Additional test completes coverage

Coverage after adding test [1,0,1,0]
All combinations covered to 100% level,
so this is a covering array.
Combinatorial Coverage Measurement

Number of tests: 7489
Number of parameters: 82

Coverage:
- 2-way coverage: 3.321 combinations, 14,761 variance combinations, total coverage: 0.940
- 3-way coverage: 88,560 combinations, 828,135 variance combinations, total coverage: 0.831

Graph shows coverage for different combinations and values.
Using Coverage Measurement
Combinatorial Sequences for Testing
Combinatorial Sequence Testing

• We want to see if a system works correctly regardless of the order of events. How can this be done efficiently?
• Failure reports often say something like: 'failure occurred when A started if B is not already connected'.
• Can we produce compact tests such that all t-way sequences covered (possibly with interleaving events)?

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>connect flow meter</td>
</tr>
<tr>
<td>$b$</td>
<td>connect pressure gauge</td>
</tr>
<tr>
<td>$c$</td>
<td>connect satellite link</td>
</tr>
<tr>
<td>$d$</td>
<td>connect pressure readout</td>
</tr>
<tr>
<td>$e$</td>
<td>start comm link</td>
</tr>
<tr>
<td>$f$</td>
<td>boot system</td>
</tr>
</tbody>
</table>
Sequence Covering Array

- With 6 events, all sequences = 6! = 720 tests
- Only 10 tests needed for all 3-way sequences, results even better for larger numbers of events
- Example: .*c.*f.*b.* covered. Any such 3-way seq covered.

<table>
<thead>
<tr>
<th>Test</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b c d e f</td>
</tr>
<tr>
<td>2</td>
<td>f e d c b a</td>
</tr>
<tr>
<td>3</td>
<td>d e f a b c</td>
</tr>
<tr>
<td>4</td>
<td>c b a f e d</td>
</tr>
<tr>
<td>5</td>
<td>b f a d c e</td>
</tr>
<tr>
<td>6</td>
<td>e c d a f b</td>
</tr>
<tr>
<td>7</td>
<td>a e f c b d</td>
</tr>
<tr>
<td>8</td>
<td>d b c f e a</td>
</tr>
<tr>
<td>9</td>
<td>c e a d b f</td>
</tr>
<tr>
<td>10</td>
<td>f b d a e c</td>
</tr>
</tbody>
</table>
Sequence Covering Array Properties

• 2-way sequences require only 2 tests
  (write events in any order, then reverse)
• For > 2-way, number of tests grows with \( \log n \), for \( n \) events
• Simple greedy algorithm produces compact test set

![Graph showing growth of tests with number of events for 2-way, 3-way, and 4-way sequences.](image-url)
Example: Laptop application

Problem: connect many peripherals, order of connection may affect application
# Connection Sequences

<table>
<thead>
<tr>
<th></th>
<th>Boot</th>
<th>P-1 (USB-RIGHT)</th>
<th>P-2 (USB-BACK)</th>
<th>P-3 (USB-LEFT)</th>
<th>P-4</th>
<th>P-5</th>
<th>App</th>
<th>Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boot</td>
<td>App</td>
<td>Scan</td>
<td>P-5</td>
<td>P-4</td>
<td></td>
<td>P-3 (USB-RIGHT)</td>
<td>P-2 (USB-BACK)</td>
</tr>
<tr>
<td>2</td>
<td>Boot</td>
<td>P-3 (USB-RIGHT)</td>
<td>P-2 (USB-LEFT)</td>
<td>P-1 (USB-BACK)</td>
<td></td>
<td>App</td>
<td>Scan</td>
<td>P-5</td>
</tr>
<tr>
<td>3</td>
<td>etc...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3-way sequence covering of connection events
Results

• Tested peripheral connection for 3-way sequences
• Some faults detected that would not have been found with 2-way sequence testing; may not have been found with random
  • Example:
  • If P2-P1-P3 sequence triggers a failure, then a full 2-way sequence covering array would not have found it
    (because 1-2-3-4-5-6-7 and 7-6-5-4-3-2-1 is a 2-way sequence covering array)
Research Questions
Given: a set of tests that the SUT fails, which combinations of variables/values triggered the failure?

variable/value combinations in **passing** tests

These are the ones we want

variable/value combinations in **failing** tests
Fault location – what's the problem?

If they're in failing set but not in passing set:
1. which ones triggered the failure?
2. which ones don't matter?

out of $v^t \binom{n}{t}$ combinations

Example:
30 variables, 5 values each
= 445,331,250
5-way combinations

142,506 combinations in each test
Integrating into Testing Program

• Test suite development
  • Generate covering arrays for tests
  OR
  • Measure coverage of existing tests and supplement

• Training
  • Testing textbooks – Mathur, Ammann & Offutt,
  • Combinatorial testing “textbook” on ACTS site
  • User manuals
  • Worked examples
Industrial Usage Reports

- Work with US Air Force on sequence covering arrays, submitted for publication
- World Wide Web Consortium DOM Level 3 events conformance test suite
- Cooperative Research & Development Agreement with Lockheed Martin Aerospace - report to be released 3rd or 4th quarter 2011
Technology Transfer

• Tools obtained by 700+ organizations; NIST “textbook” on combinatorial testing downloaded 9,000+ times since Oct. 2010


• We are always interested in working with others!
Please contact us if you would like more information.

Rick Kuhn
tuhn@nist.gov

Raghu Kacker
raghu.kacker@nist.gov

http://csrc.nist.gov/acts

(Or just search “combinatorial testing”. We’re #1!)