Automated Combinatorial Testing for Software

Rick Kuhn and Vadim Okun

National Institute of Standards and Technology
Gaithersburg, MD
What is NIST?

National Institute of Standards and Technology

• The nation’s measurement and testing laboratory

• 3,000 scientists, engineers, and support staff including 3 Nobel laureates

• Best known for atomic clock, standard reference materials (for instrument calibration) from aluminum alloy to whale blubber

• Basic and applied research in physics, chemistry, materials, electronics, computer science
Automated Combinatorial Testing

● Project to combine automated test generation with combinatorial methods

● Goals – reduce testing cost, improve cost-benefit ratio for formal methods
Overview of useful results

- Proof of concept demo integrating combinatorial testing with model checking
- (Small) experimental result consistent with earlier interpretation of empirical data
- New combinatorial algorithms and tools, supporting development tradeoffs
Problem: the usual ...

- Too much to test
- Even with formal specs, we still need to test
- Take advantage of formal specs to produce tests also – better business case for FM
- Testing may exceed 50% of development cost

- Example: 20 variables, 10 values each
- $10^{20}$ combinations
- Which ones to test?
Solution: Combinatorial Testing

• Suppose no failure requires more than a pair of settings to trigger
• Then test all pairs – 180 test cases sufficient to detect any failure

Yes, but aren’t real-world failures more complicated?
Pairwise testing – what do we know?

- Mandl, 1985 – very effective for compiler test
- Brownlie, Prowse, Phadke - high coverage
- Cohen, Dalal, Parelius, Patton, 1995 – 90% coverage with pairwise, all errors in small modules found
- Dalal, et al. 1999 – effectiveness of pairwise testing, no higher degree interactions
- Smith, Feather, Muscetolla, 2000 – 88% and 50% of flaws for 2 subsystems,

What if finding ~90% of flaws is not good enough?
How many combinations do we need to test to find ALL errors?

• Surprisingly, no one had looked at this question when NIST studied medical device software in 1999
  • Wallace, Kuhn 2001 – medical devices
    – 98% of flaws were pairwise interactions, no failure required > 4 conditions to trigger
  • Kuhn, Reilly 2002 – web server, browser; no failure required > 6 conditions to trigger
  • Kuhn, Wallace, Gallo 2004 – large NASA distributed database; no failure required > 4 conditions to trigger
• Max failure triggering fault interaction (FTFI) number of these applications was 6
• Much more empirical work needed

Maybe the hard to find flaws weren’t reported.
FTFI numbers for 4 application domains – failures triggered by 1 to 6 conditions
Combinatorial test example: 5 parameters, 4 values each, 3-way combinations

<table>
<thead>
<tr>
<th>Test</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

All 3-way combinations of A,B,C values

But also all 3-way combinations of A,B,D; A,B,E; A,C,D; ...

B,D,E; ...

etc...

This is going to take a lot of tests!
Problem: Combinatorial Testing Requires a Lot of Tests

- Number of tests: suppose we want all 4-way combinations of 30 parameters, 5 values each: 3,800 tests – too many to create manually
- Test set to do this is a covering array
- Time to generate covering arrays: problem is NP hard
- No. of combinations: \( \binom{n}{k} v^k \)

For \( n \) variables with \( v \) values, \( k \)-way combinations
Solution: Automated Testing

Test data generation – easy
Test oracle generation – hard
Creating test oracles – model checking and other state exploration methods

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
Using model checking to produce tests

The system can never get in this state!

Yes it can, and here’s how ...

---

**System specs**

- generate mutants
- model checker
- combine scenarios
- generate test input

**System source**

- Run Tests
- Complete Tests

**Flow**

- Test cases
- Test results
- TDA
- Combine scenarios
- Counter-examples
- Mutant specs
- System specs
-- 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;
    1 : UNRESOLVED;
    esac;

...

SPEC AG ((enabled & (intent_not_known | !tcas_equipped) & !need_downward_RA & need_upward_RA) -> AX (alt_sep = UPWARD_RA))
Computation Tree Logic

The usual logic operators, plus temporal:

- \( \forall \phi \) - All: \( \phi \) holds on all paths starting from the current state.
- \( \exists \phi \) - Exists: \( \phi \) holds on some paths starting from the current state.
- \( \Box \phi \) - Globally: \( \phi \) has to hold on the entire subsequent path.
- \( \Diamond \phi \) - Finally: \( \phi \) eventually has to hold
- \( \chi \phi \) - Next: \( \phi \) has to hold at the next state

[other operators not listed]

execution paths
\[
\text{SPEC } \Box (\text{enabled } \& \ (\text{intent_not_known } \mid \ \text{!tcas_equipped}) \ & \ \text{!need_downward_RA } \ & \ \text{need_upward_RA}) \ \\
\rightarrow \Box \chi (\text{alt_sep } = \text{UPWARD_RA})
\]

“FOR ALL executions,
IF enabled \& (intent_not_known ....
THEN in the next state alt_sep = UPWARD_RA”
How can we integrate combinatorial testing with model checking?

1. 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 “var1 = val1”
   - Now combine this constraint with assertion to produce counterexamples. Some possibilities:
     - $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow \text{AX} \neg(R))$
     - $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} \neg(1))$
     - $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} \neg(R))$
What happens with these assertions?

1. \( AG(v_1 \land v_2 \land \ldots \land v_k \land P \rightarrow AX \neg(R) ) \)
   
   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. \( AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(1) ) \)
   
   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. \( AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(R) ) \)
   
   Forces production of a counterexample for each \( R \).
   This is just right
Proof-of-concept experiment

- Traffic Collision Avoidance System module
  - Small, practical example – 2 pages of SMV
  - Used in other experiments on testing
    - Siemens testing experiments, Okun dissertation
    - Suitable for model checking
- 12 variables: 7 boolean, two 3-value, one 4-value, two 10-value
- Tests generated w/ Lei “In Parameter Order” (IPO) algorithm extended for >2 parameters
### Combinations /tests generated

<table>
<thead>
<tr>
<th>$t$</th>
<th>Comb.</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way:</td>
<td>100</td>
<td>156</td>
</tr>
<tr>
<td>3-way:</td>
<td>405</td>
<td>461</td>
</tr>
<tr>
<td>4-way:</td>
<td>1,375</td>
<td>1,450</td>
</tr>
<tr>
<td>5-way:</td>
<td>4,220</td>
<td>4,309</td>
</tr>
<tr>
<td>6-way:</td>
<td>10,902</td>
<td>11,094</td>
</tr>
</tbody>
</table>

*(more “don’t care” conditions at lower interaction levels)*
Results

- Roughly consistent with data on large systems
- But errors harder to detect than real-world examples
What do we need to make this practical?

- This approach would not have been practical 10 years ago
- Now we have high performance model checkers, better covering array algorithms, and cheap processors
- Generating $\sim 10^6 - 10^7$ tests can be done
- Proof of concept experiment completed

So what? Finding covering arrays is an NP hard problem!
Solution: new covering array algorithms

- **Tradeoffs to minimize calendar/staff time:**
  - FireEye (extended IPO) – Lei – roughly optimal, can be used for most cases under 40 or 50 parameters
    - Produces minimal number of tests at cost of long run time
    - Currently integrating algebraic methods
  - Adaptive distance-based strategies – Bryce – dispensing one test at a time w/ metrics to increase probability of finding flaws
    - Highly optimized covering array algorithm
    - Variety of distance metrics for selecting next test
  - Paintball – Kuhn – for more variables or larger domains
    - Randomized algorithm, generates tests w/ a few tunable parameters; computation can be distributed
    - Better results than other algorithms for larger problems
Will automated combinatorial testing work in practice?

The usual potential pitfalls:

- Faithfulness of model to actual code
  - Always a problem
  - Being able to generate tests from specification helps make formal modeling more cost effective

- Time cost of generating tests
  - Model checking very costly in run time
  - Inherent limits on number of variable values even with ideal covering array generation: need at least \( C(n,k) \times v^k \)

- Abstraction needed to make this tractable
  - Equivalence classes for variable values may miss a lot that matters
  - Not all software is suited to this scheme – e.g., good for code with lots of decisions, not so good for numerical functions.
Scaling up

• Two real-world trials planned – US Govt Personal Identity Verification (PIV) card, machine tool specification exchange software

• Plan to experiment with both SMV model checker and TVEC

• Generate $10^5$ to $10^6$ tests per module, probably up to 5-way combinations

How is this going to work in the real world?
Summary and conclusions

• Proof of concept is promising – integrated w/ model checking
• Appears to be economically practical
• New covering array algorithms help make it more tractable
• Cluster implementation of covering array algorithm
• Many unanswered questions
  • Is it cost-effective?
  • What kinds of software does it work best on?
  • What kinds of errors does it miss?
  • What failure-triggering fault interaction level testing is required? 5-way? 6-way? more?
• Large real-world example will help answer these questions

Please contact us if you are interested!

Rick Kuhn          Vadim Okun
kuhn@nist.gov    vadim.okun@nist.gov

http://csrc.nist.gov/acts