Pseudo-Exhaustive Testing for Software

Rick Kuhn  
kuhn@nist.gov

Vadim Okun  
vadim.okun@nist.gov

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

• Basic and applied research in physics, chemistry, materials, electronics, computer science
Automated Testing Using Covering Arrays

- Project to combine automated test generation with combinatorial methods
- Goals – reduce testing cost, improve cost-benefit ratio for formal methods
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

- How many interactions required to detect flaws in real-world software?
- If we know, can conduct “pseudo exhaustive” testing
Example: 5 parameters, 4 values each, 3-way combinations

<table>
<thead>
<tr>
<th>Test</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C D E</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>1 1 2 2 2</td>
</tr>
<tr>
<td>3</td>
<td>1 1 3 3 3</td>
</tr>
<tr>
<td>4</td>
<td>1 1 4 4 4</td>
</tr>
<tr>
<td>5</td>
<td>1 2 1 2 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2 2 1 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2 3 4 1</td>
</tr>
<tr>
<td>8</td>
<td>1 2 4 3 2</td>
</tr>
<tr>
<td>9</td>
<td>1 3 1 3 4</td>
</tr>
<tr>
<td>10</td>
<td>1 3 2 4 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....
Failure-Triggering Fault Interactions – data suggest few variables involved

- Dalal, et al. 1999 – effectiveness of pairwise testing, no higher degree interactions
- Smith, Feather, Muscetolla 2000 – NASA Deep Space 1 software – pairwise testing detected 88% and 50% of flaws for 2 subsystems, no higher degree interactions
- Wallace, Kuhn 2001 – medical device s/w – 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
FTFI numbers for 4 application domains – failures triggered by 1 to 6 conditions
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
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
• Modeled in SMV, model checked w/ NuSMV
• Test cases produced/tests generated:
  – 2-way: 100 / 156
  – 3-way: 405 / 461
  – 4-way: 1,375 / 1,450
  – 5-way: 4,220 / 4,309
  – 6-way: 10,902 / 11,094
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;
  1 : UNRESOLVED;
  esac;
...
SPEC AG ((enabled & (intent_not_known | !tcas_equipped) &
  !need_downward_RA & need_upward_RA) -> AX (alt_sep = UPWARD_RA))
The usual logic operators, plus temporal:

- \( A \varphi \) - All: \( \varphi \) holds on all paths starting from the current state.
- \( E \varphi \) - Exists: \( \varphi \) holds on some paths starting from the current state.
- \( G \varphi \) - Globally: \( \varphi \) has to hold on the entire subsequent path.
- \( F \varphi \) - Finally: \( \varphi \) eventually has to hold
- \( X \varphi \) - Next: \( \varphi \) has to hold at the next state

[others not listed]

```
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"
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. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow \text{AX} ! (R))$
  2. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} ! (1))$
  3. $\text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow \text{AX} ! (R))$
What happens with these assertions?

1. \( AG(v_1 \land v_2 \land \ldots \land v_k \land P \rightarrow AX ! (R) ) \)

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

2. \( AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX ! (1) ) \)

   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!

3. \( AG(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX ! (R) ) \)

   Forces production of a counterexample for each R.
   This is just right!
Results

- Roughly consistent with data on large systems
- But errors harder to detect than real-world examples studied
Problem: the usual - scaling up

- Tests for Personal Identity Verification (PIV) smartcard to be used by all US Government employees and contractors
- Three modules: 25, 29, 43 parameters
- Tests currently being produced with TVEC test generator
- Plan to experiment with both TVEC and SMV model checker
- Generate $10^5$ to $10^6$ tests per module, probably up to 5-way combinations
- Producing optimal covering arrays is NP hard!
Solution: new covering array algorithms

• Tradeoffs:

• FireEye (extended IPO) – Lei – optimal, can be used for most cases under 50 parameters
  • Produces minimal number of tests at cost of long run time
  • Outperforms other algorithms in minimizing tests
  • Run time exponential in interaction level
  • Not parallelized

• Paintball – Kuhn – suboptimal, can be used for very large arrays (>50 parameters) or higher interaction levels
  • Simple algorithm, generates random tests w/ a few tunable parameters
  • Suboptimal in number of tests by 3% to 40% depending on run time
  • Extremely fast and can be parallelized (still hits a wall, but the wall is further away than for other algorithms)
  • Reduces test generation time from 40-60 days to one or two, at cost of redundant tests
Number of tests vs. runtime, 4-way interactions, 4-value variables
No optimization, single processor for paintball algorithm

This increment getting smaller
This increment getting way bigger!

Will pseudo-exhaustive 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) * \(v^k\)

- 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.
Summary and conclusions

- Proof of concept is promising – integrated w/ model checking
- Appears to be economically practical
- New covering array algorithms help make this approach more tractable
- Working on 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
kuhn@nist.gov

Vadim Okun
vadim.okun@nist.gov